컴퓨터 공학 기초 실험2 보고서

실험제목: Carry Look-ahead Adder (CLA)

실험일자: 2023년 09월 27일 (수)

제출일자: 2023년 10월 1일 (월)

학 과: 컴퓨터정보공학부

담당교수: 이준환 교수님

실습분반: 수요일 0, 1, 2

학 번: 2022202075

성 명: 우나륜

1. 제목 및 목적

A. 제목

Carry Look-ahead Adder (CLA)

B. 목적

계산이 완료될 때까지 시간이 많이 소요되는 Ripple Carry Adder Carry (RCA)의 단점을 보안하기 위해 등장한 Look-ahead Adder (CLA)의 작동 원리를 이해하고 이를 사용하여 4-bit CLA와 32-bit CLA를 Verilog를 사용하여 설계하고 RCA와 CLA의 module에 Clock을 인가하여 유효한 결과가 나오는 데 필요한 clock 주기가 나오는 지 확인하는 데 목적을 둔다.

2. 원리(배경지식)

A. Ripple Carry Adder (RCA)

N비트의 Ripple Carry Adder는 더하고자 하는 두 입력의 비트의 개수 N만큼 full adder를 연결하여 구현할 수 있다. 하지만 RCA는 이전 비트에서 캐리가 계산되어 carry-in으로 입력될 때까지 상위 비트 가산기는 아무런 수행없이 기다려야 하므로 비트 수가 늘어날수록 연산 속도가 느려진다. 또한, N의 개수가 무한히 커짐에 따라 연결해야 하는 full adder의 개수도 늘어나므로 면적 또한 넓어진다는 단점을 가진다.

B. Full Adder for CLA

CLA에 사용할 full adder는 carry out을 필요로 하지 않는다. 해당 full adder의 진리표는 아래의 표1에서 확인할 수 있다.

Input			Output
Ci	a	b	s
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

표1. Full Adder for CLA truth table

위 진리표에서 SOP를 사용하여 s를 구하면 다음과 같다. $s = \overline{ci}\,\overline{a}b + \overline{ci}a\overline{b} + ci\overline{ab} +$

$ciab = ci(\overline{ab} + ab) + \bar{ci}(\bar{a}b + a\bar{b}) = ci \oplus a \oplus b$. 따라서 carry-out이 필요하지 않은 해당 full adder는 하나의 3-to-1 XOR 게이트 또는 두 개의 2-to-1 XOR 게이트를 사용하여 구현할 수 있다.

C. Carry Look-ahead Adder (CLA)

위에서 설명한 Ripple Carry Adder의 단점을 보완하기 위해 등장한 가산기로, 캐리가 입력될 때까지 기다리지 않고 캐리를 미리 보고 계산한다는 작동원리에 의해 Carry Look-ahead라는 이름이 붙여졌다. Input으로 데이터 a, b, carry-in이 입력되면, 모든 자리올림이 동시에 계산이 되어 계산시간이 단축된다.

D. Carry Look-ahead Block (CLB)

CLA에서 carry-out의 값을 미리 계산하기 위해 Carry Look-ahead Block를 사용한다. CLB에서 자리올림을 예측하기 위해 Generate와 Propagate라는 신호를 새롭게 정의하여 사용하게 된다. 이를 full adder의 carry-out에 적용하면, carry-in (C_i)의 값을 모른 채 carry-out (C_{i+1})을 계산하고자 할 때, $C_{i+1} = A_iB_i + (A_i + B_i)C_i$로 계산할 수 있다. 이때, $G_i = A_iB_i \; and \; P_i = (A_i + B_i)$로 정의하고 $C_{i+1} = G_i + P_iC_i$로 나타낸다.

E. Timing Analysis

본 과제에서 32-bit RCA 또는 32-bit CLA에 앞뒤로 flip-flop을 추가하여 clock과 연결하는 module을 만들어 타이밍을 분석할 수 있다. Timing analysis를 통해 회로의 delay를 분석할 수 있는데, 이는 구현한 회로가 어떠한 조건에서 제대로 작동하는지 확인하기 위함이다. Flip-flop의 동작속도를 결정하고 안정하게 동작할 수 있는 최대 동작 주파수(Fmax)를 찾고 최대 동작 주파수 이하에서 회로를 동작해야 한다.

F. Verilog always와 parameter

Verilog에서 사용하는 always 구문은 한 번만 실행되는 것이 아닌, 조건이 만족할 때마다 동작한다. 예를 들어, always @ (posedge clk); 문장에서 @는 at으로 "~일 때"라는 뜻이며 posedge은 rising edge를 뜻한다. 즉, always @ 뒤의 조건이 참일 때만 그 아래의 명령어들을 실행을 하게 된다. 반대로, falling edge는 negedge로 사용한다.

4-bit CLA에서 결과값 tb_s와 tb_co의 값을 한 번에 확인하기 위해 signal을 묶는 결합 연산자를 사용하였다. 중괄호({ })를 사용하여 result에 {tb_co, tb_s}를 묶어 할당해준다. 이를 testbench에서 확인하면 두 값이 이어져서 나타나며 result에서 MSB는 tb_co가 된다.

Verilog에서도 사용하고자 하는 상수 값을 저장하는 parameter는 <parameter 파라미터이름 = 상수값>으로 선언한 뒤 사용할 수 있다. Parameter는 변수나 wire가 아니며, 변수나 net으로 선언한 것을 다시 parameter로 선언할 수도 없다. 이후에 파라미터의 값을 변경하고 싶다면, 파라미터 이름 뒤에 # (새로운 값)을 통해 변경할 수 있다.

3. 설계 세부사항

A. Reversion of full adder

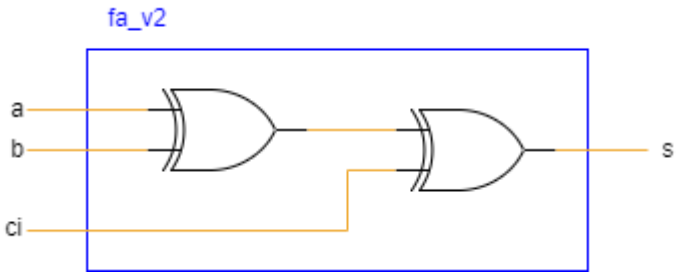

그림1. Fa_v2 Design

위의 그림1은 carry-out을 계산하지 않는 full adder이다. Input으로 a, b, ci를 입력받고 output으로 s를 출력한다. 본 과제에서는 2개의 2-to1 XOR 게이트를 사용하여 fa_v2를 설계하였다. 먼저, a와 b를 XOR 연산하고, 그 결과값을 ci와 XOR 연산하여 s의 값을 계산하게 된다.

B. 32-bit Ripple Carry Adder

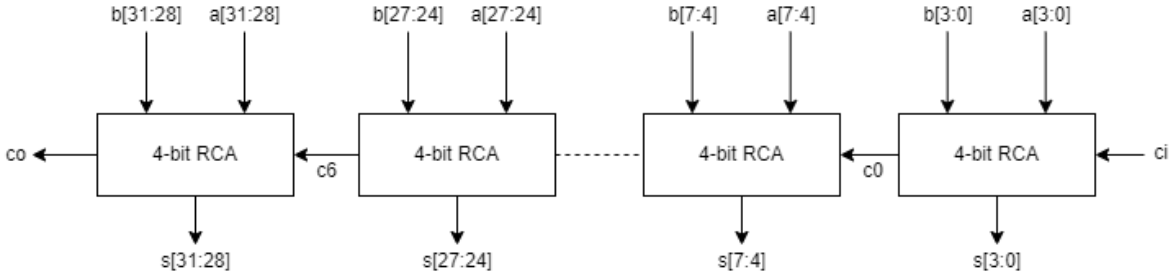

그림2. 32-bit Ripple Carry Adder Design

이전 과제에서 구현한 4-bit RCA를 인스턴스화하여, 첫번째 4-bit RCA가 하위 4비트를 연산하고 그 다음 RCA가 다음 4비트를 연산하는 방식으로 각 4-bit RCA가 4비트씩 맡아 총 8개를 연결하여 32-bit RCA를 구현할 수 있다.

C. 4-bit Carry Look-ahead Block

4-bit CLB는 input으로 4-bit data a, b와 carry-in을 입력받는다. 그리고 4-bit Generate와 Propagate wire를 생성한다. 또한, 각 Ci 연산에 필요한 wire를 선언한다.

C1의 경우 가장 아래 단계의 carry-out이므로 w0_c1 하나만 선언한다. C2, C3, Cout는 각각 2개, 3개, 4개가 필요하게 된다. 그리고 Generate Signal인 Gi와 Propagate Signal인 Pi를 G0~G3, P0~P3까지 연산한다. C1, C2, C3, Cout를 위에서 구한 식을 사용하여 차례대로 계산한다.

$$G_i = A_i B_i$$
$$P_i = A_i + B_i$$
$$C_1 = G_0 + P_0 C_0$$
$$C_2 = G_1 + P_1 C_1 = G_1 + P_1(G_0 + P_0 C_0) = G_1 + P_1 G_0 + P_1 P_0 C_0$$
$$C_3 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0$$
$$Cout = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_0$$

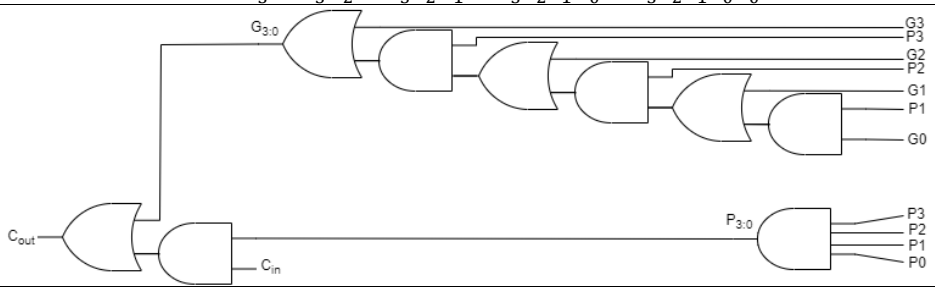

그림3. 4-bit Carry Look-ahead Block Design

D. 4-bit Carry Look-ahead Adder

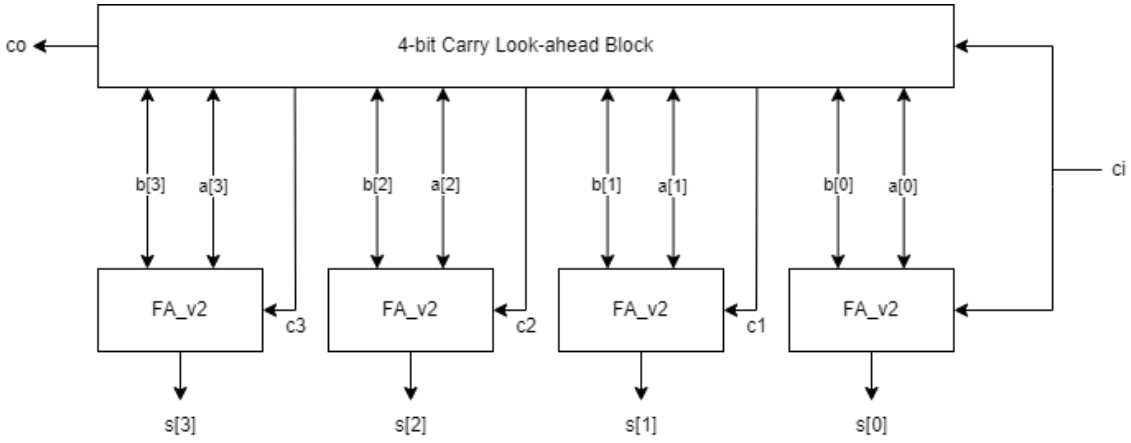

그림4. 4-bit Carry Look-ahead Adder Design

4-bit CLA는 앞서 구현한 4-bit CLB에 carry-out이 없는 full adder FA_v2를 4개 연결하여 구현할 수 있다. 4-bit CLB에서 계산한 C1, C2, C3와 Carry-in, a[3:0]과 b[3:0]을 각각 FA_v2에 입력하면 s[0], s[1], s[2], s[3]을 얻을 수 있고, 4-bit CLB를 통해 carry-out을 얻을 수 있다. 4-bit CLA 설계는 위의 그림4와 같다. 결과적으로 4-bit CLA 하나를 구현하기 위해 4-bit CLB하나와 4개의 FA_v2가 요구된다.

E. 32-bit Carry Look-ahead Adder

32-bit Carry Look-ahead Adder는 위에서 설계한 4-bit CLA를 직렬로 8개를 연결하여 구현하였다. 4-bit CLA module을 인스턴스화하고, 8개를 연결하면 하나의 4-bit

CLA가 4비트를 담당하여 총 32비트를 연산할 수 있다. 각 4-bit CLA의 연산결과로 산출된 Carry-out은 c1~c8을 선언하여 차례대로 저장하고, 다음 4-bit CLA의 carry-in으로 입력하였다. 마지막 carry-out인 c8은 32-bit CLA의 Carry-out이므로 co에 c8을 할당한다.

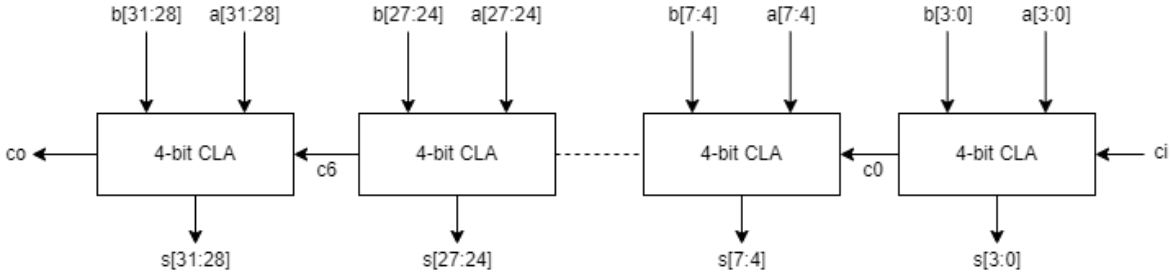

그림5. 32-bit Carry Look-ahead Adder Design

F. Clock

RCA와 CLA의 타이밍을 분석하기 위해 rca_clk, cla_clk module을 생성하였다. 각 clock module은 input으로 clk, a, b, ci를 갖고, output으로 s, co를 갖는다. 타이밍을 분석하기 위해 각 input과 output port에 flip-flop을 추가하여 clock과 연결해 준다. Flip-flop은 cla_clk module의 input과 cla32 module의 input, cla_clk output과 cla32 output사이에 register인 reg_a, reg_b, reg_ci, reg_s, reg_co과 clk를 각각 연결하여 구현하였다. 구현한 모습은 아래의 그림과 같다.

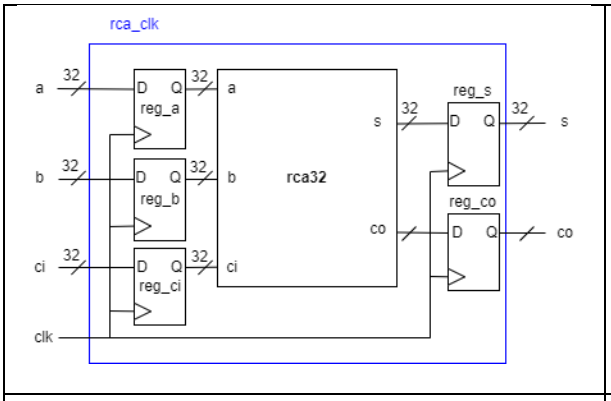

그림6. RCA Clock design

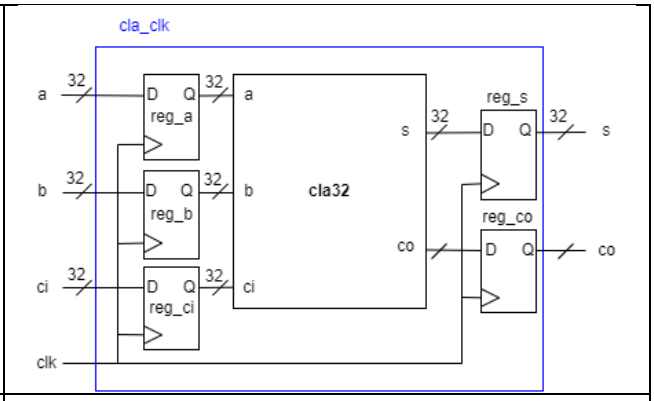

그림7. CLA Clock design

4. 설계 검증 및 실험 결과

A. 시뮬레이션 결과

1) 32-bit RCA

아래의 그림8은 tb_rca_clk의 waveform이다. 그림에서 확인할 수 있듯이 clk_rca의 rising edge에 data의 값이 제대로 입력되고 output의 값이 알맞게 계산이 되는 것을 확인할 수 있다. 그리고 clock의 주기를 저장한 parameter STEP_rca의 값을 10ns로

설정하였고, STEP의 절반은 clock의 falling edge로 설정하였다. 따라서 5ns마다 clk_rca가 rising 또는 falling하는 것을 확인할 수 있다.

첫번째 rising edge에서 데이터 tb_a_rca, tb_b_rca, tb_ci_rca의 값이 초기화된다. 두번째 rising edge에서 이전의 입력 데이터들의 결과값인 tb_s_rca와 tb_co_rca의 값이 나타나고, 데이터 tb_a_rca, tb_b_rca, tb_ci_rca에 새로운 값이 설정된다. 세번째 rising edge에서는 tb_s_rca의 값이 이전의 값과 같아 값이 명시적으로 표시되지 않고 이전의 결과값에서 carry-out이 발생하여 co의 값이 rising된 것을 확인할 수 있다. 이러한 방식으로 값들이 각각 4번씩 바뀌고 마지막 결과값을 보여주기 위해 (STEP*2)ns 후에 testbench를 종료하였다.

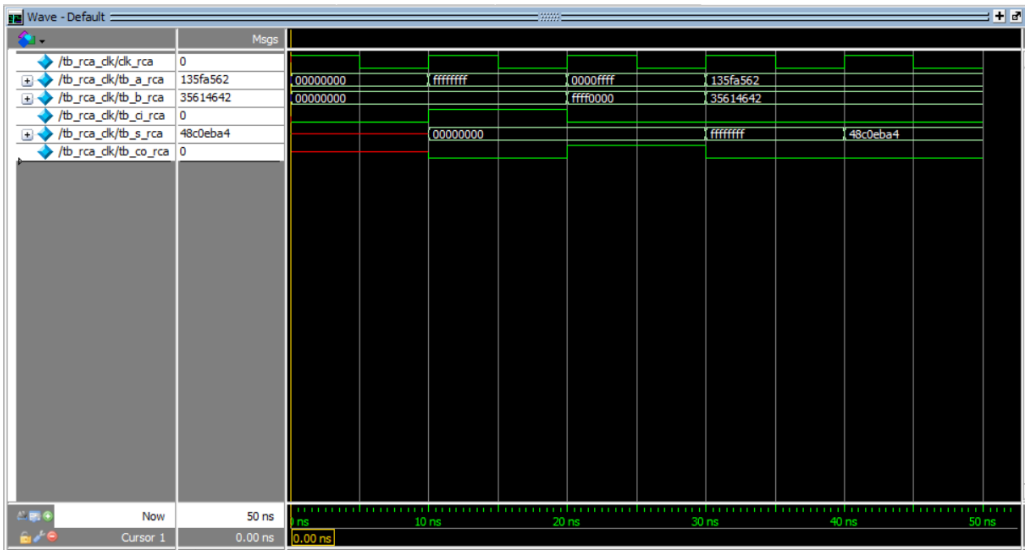

그림8. tb_rca_clk waveform

2) 4-bit CLA

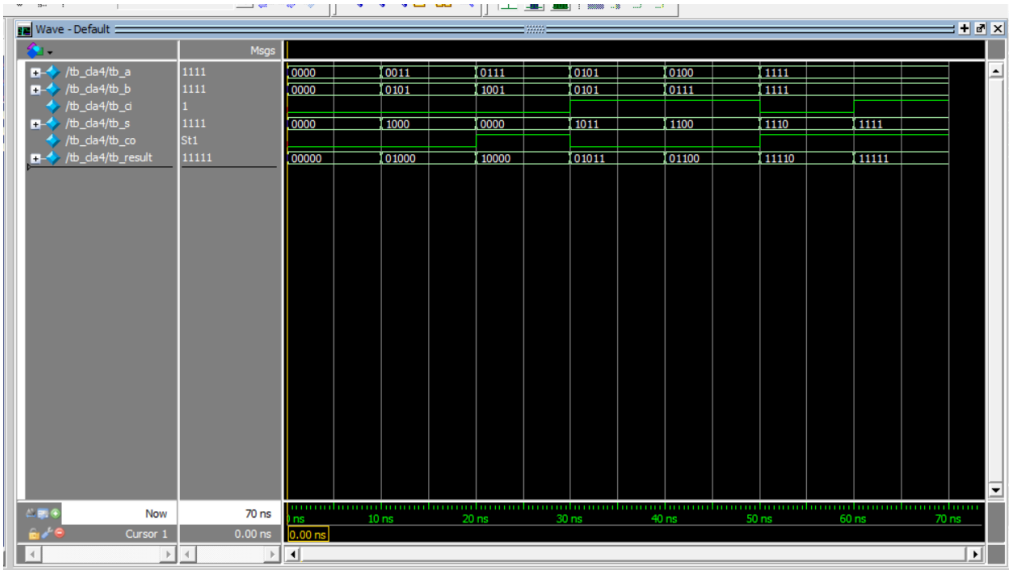

그림9. tb_cla4 waveform

위의 그림9는 4-bit Carry Look-ahead Adder testbench의 waveform이다. 처음 모든 변수들의 값을 0으로 초기화하고 10ns마다 각 입력들의 값을 설정해주었다. 두번째 결과에서 0011과 0101을 더한 값은 1000이고 carry out은 0이므로 알맞게 계산이 되었다. 또한, tb_result에 {tb_co, tb_s}를 할당하여 서로를 묶어주었다. 두번째 결과값을 예로 들면, tb_co=0, tb_s=1000으로 tb_result의 값은 01000이 된다. 이후에 값들도 알맞게 나오는 것을 확인할 수 있다.

3) 32-bit CLA

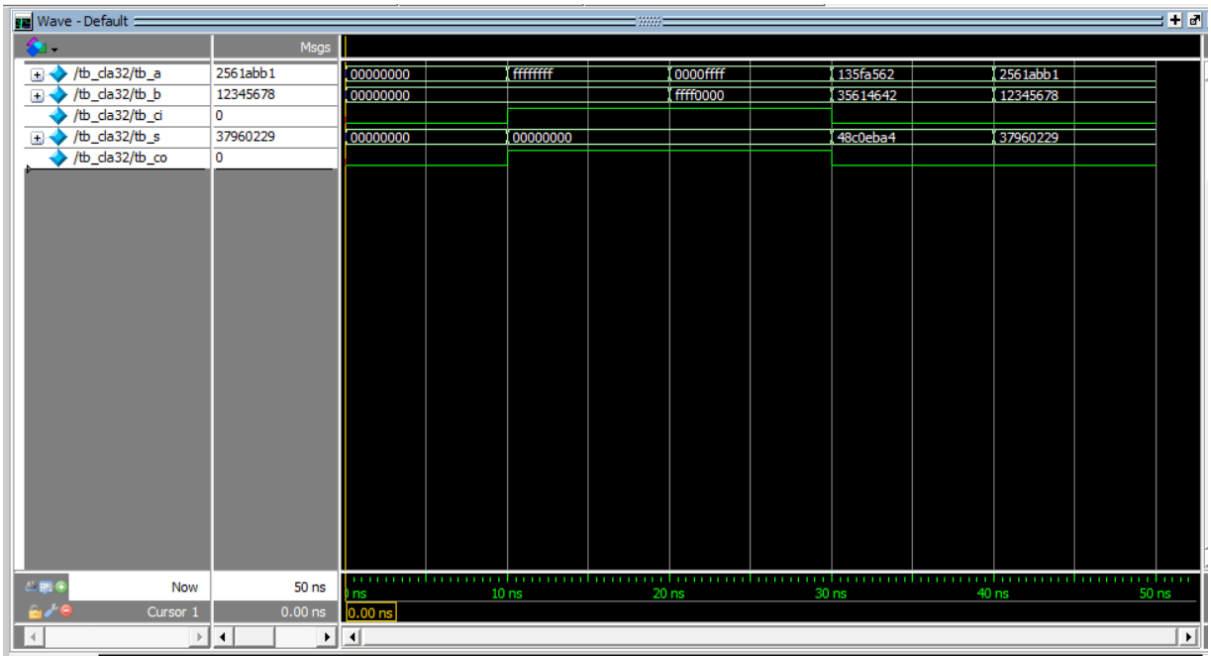

그림10. tb_cla32 waveform

위의 그림10은 32-bit CLA의 testbench waveform이다. 처음 input과 output의 값을 0으로 초기화하고, 이후에 차례대로 값을 대입하였다. 10ns마다 input의 값들을 바꾸며 output의 값을 확인하였다. 10ns에서 tb_a=32'hFFFF_FFFF, tb_b=32'h0000_0000, tb_ci=1'b1을 대입하였다. 세 입력값들의 결과값 tb_s=00000000, tb_co=1로 알맞게 계산된 것을 확인할 수 있다. 20ns에서 tb_a, tb_b, tb_ci를 더한 값이 tb_s=00000000, tb_co=1로 이전의 값과 같다는 것을 확인할 수 있다. 이후 30ns, 40ns에도 마찬가지로 값을 할당하고 그에 맞는 결과가 계산되었음을 확인할 수 있다. 40ns의 결과를 10ns만큼 유지하여 나타낸 후 testbench를 종료하였다.

4) cla_clk

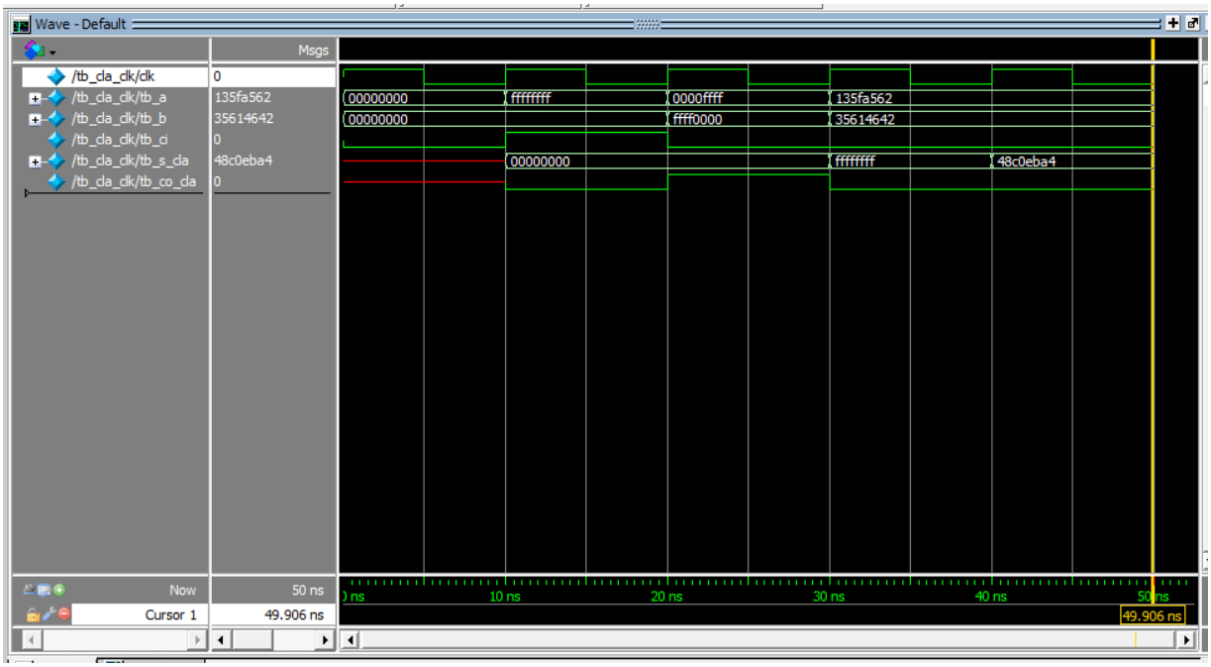

그림11. tb_cla_clk waveform

위의 그림11은 cla_clk의 testbench waveform이다. 그림에서 확인할 수 있듯이 clk의 rising edge에 data의 값이 입력되고 output의 값이 알맞게 계산이 되며 clock의 주기를 저장한 parameter STEP의 값을 10ns로 설정하여 STEP의 절반은 clock의 falling edge로 설정하였다. 따라서 5ns마다 clk_rca가 rising 또는 falling하는 것을 확인할 수 있다.

0ns때, 모든 input들의 값을 0으로 초기화하였고 output인 tb_s와 tb_co의 값은 알 수 없다. 10ns때, 이전에 입력받은 input의 결과값인 tb_s와 tb_co의 값을 확인할 수 있고, tb_a=32'hffffffff, tb_b=23'h00000000, tb_ci=1'b1으로 새로운 값을 할당하였다. 20ns때, 이전 입력값들의 합의 결과인 tb_s=0, tb_co=1로 제대로 계산된 것을 확인할 수 있다. 30ns도 위와 같은 방식으로 입력되고 그에 따른 값이 알맞게 계산되었다. 40ns~50ns까지 이전의 결과값을 나타낸 후 testbench를 종료하였다. 위 tb_cla_clk는 tb_rca_clk와 같은 testbench이며 두 덧셈기의 결과값이 같다는 것을 확인할 수 있다.

B. 합성(synthesis) 결과

1) RCA Clock

아래의 그림12는 rca_clk의 flow summary이다. 그림에서 확인할 수 있듯이, compilation이 성공적으로 완료되었다. Logic utilization은 31로 1%미만이고, total registers는 98개이며, total pins는 99개로 20% 사용하였다.

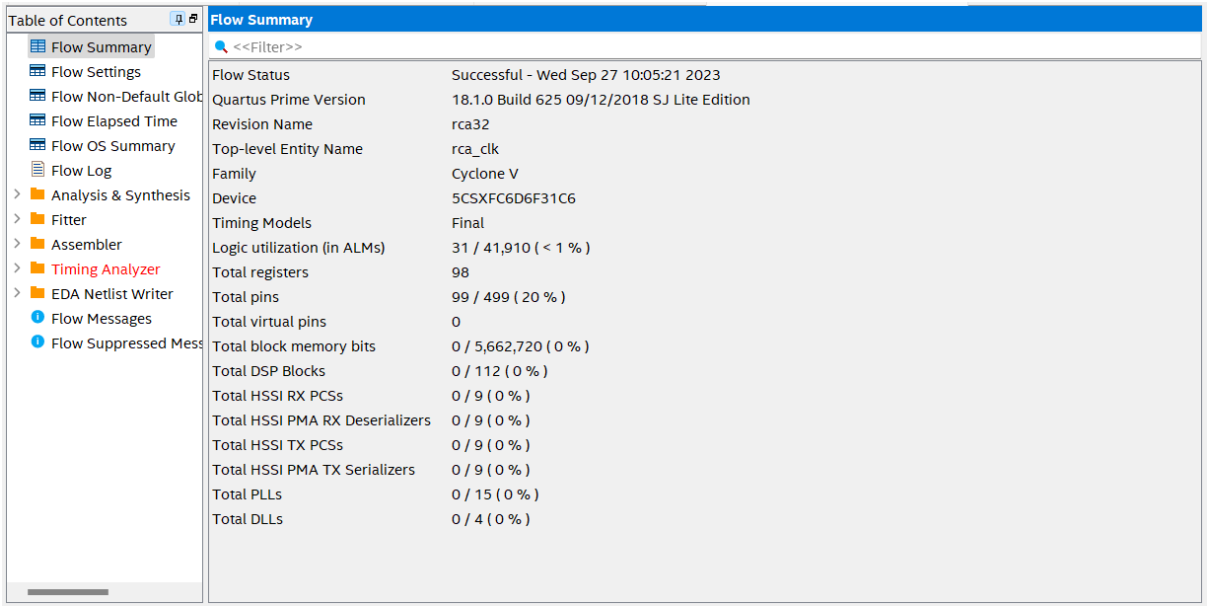

그림12. rca_clk Flow Summary

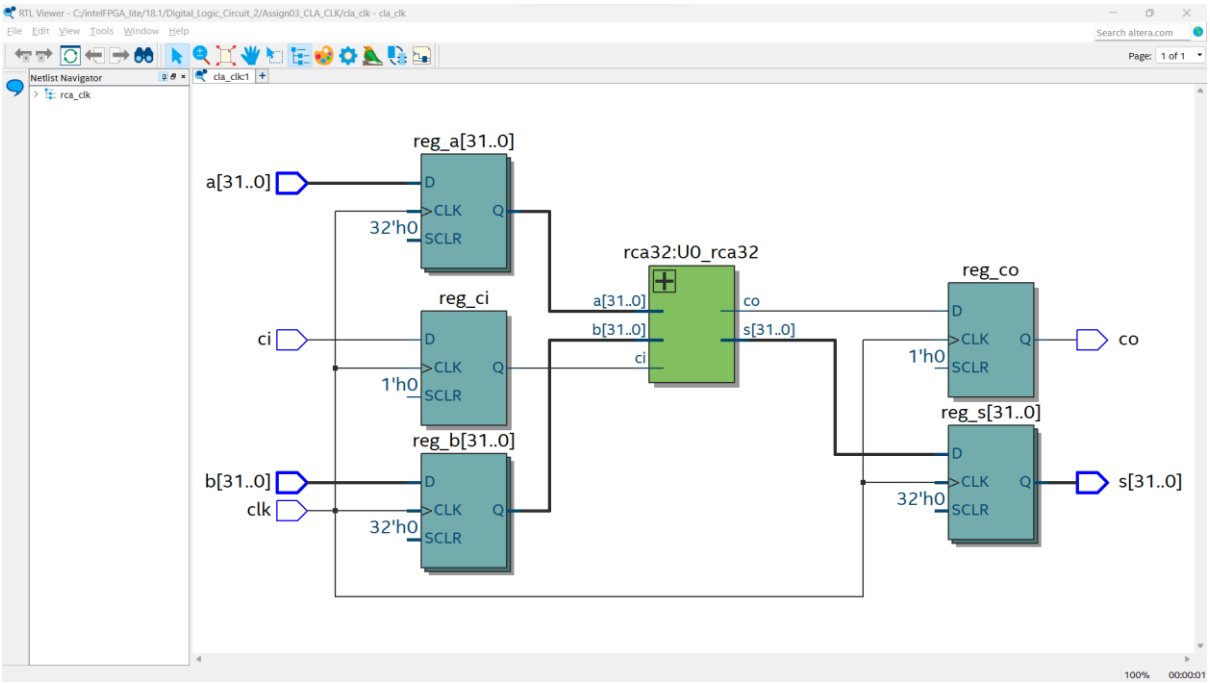

그림13. rca_clk RTL viewer

RCA clock은 clock의 input과 output, rca32의 각 input과 output의 사이에 register를 연결하여 구현하였다.

2) 4-bit CLA

아래의 그림 14는 4-bit CLA의 flow summary이다. Compilation 결과는 성공적이며, 사용한 Logic utilization은 5로 1%미만이고, 사용한 총 레지스터의 개수는 0개, total pins는 14개로 3%미만인 것을 확인할 수 있다.

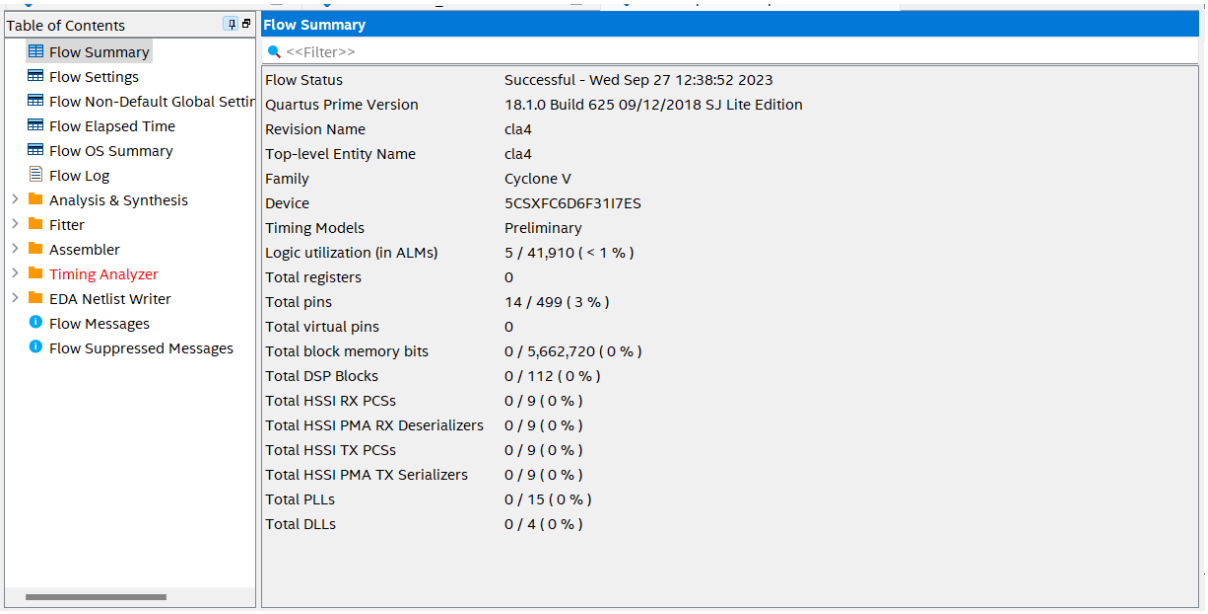

그림14. cla4 Flow Summary

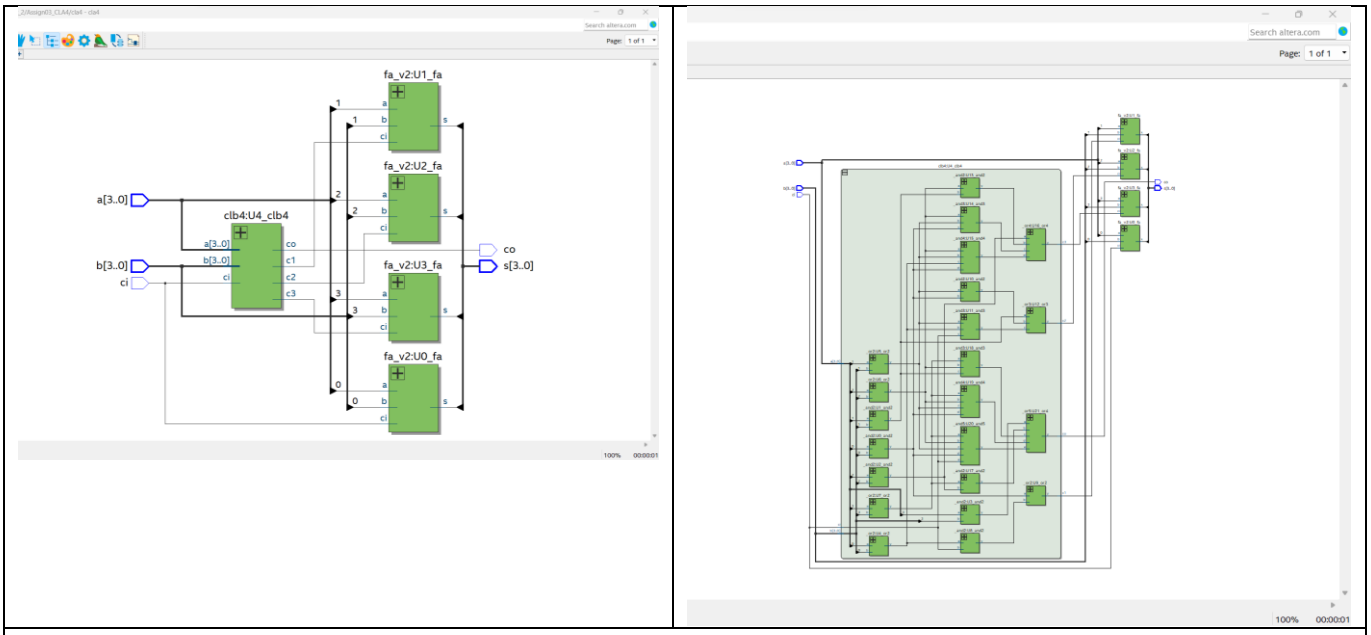

그림15. cla4 RTL viewer

RTL viewer를 통해 cla4는 한 개의 clb4와 4개의 fa_v2를 사용하여 구현하였음을 확인할 수 있다. 또한, clb4를 펼쳐보면 여러 개의 OR gates, AND gates가 사용된 것을 알 수 있다.

3) CLA Clock

아래의 그림16은 cla_clk의 flow summary이다. 그림에서 확인할 수 있듯이, compilation이 성공이고, 사용한 Logic utilization은 38로 1%미만이지만 rca_clk보다는 조금 더 사용하였다. Total registers는 98개이고 total pins는 99개로 20% 정도 사용하였다.

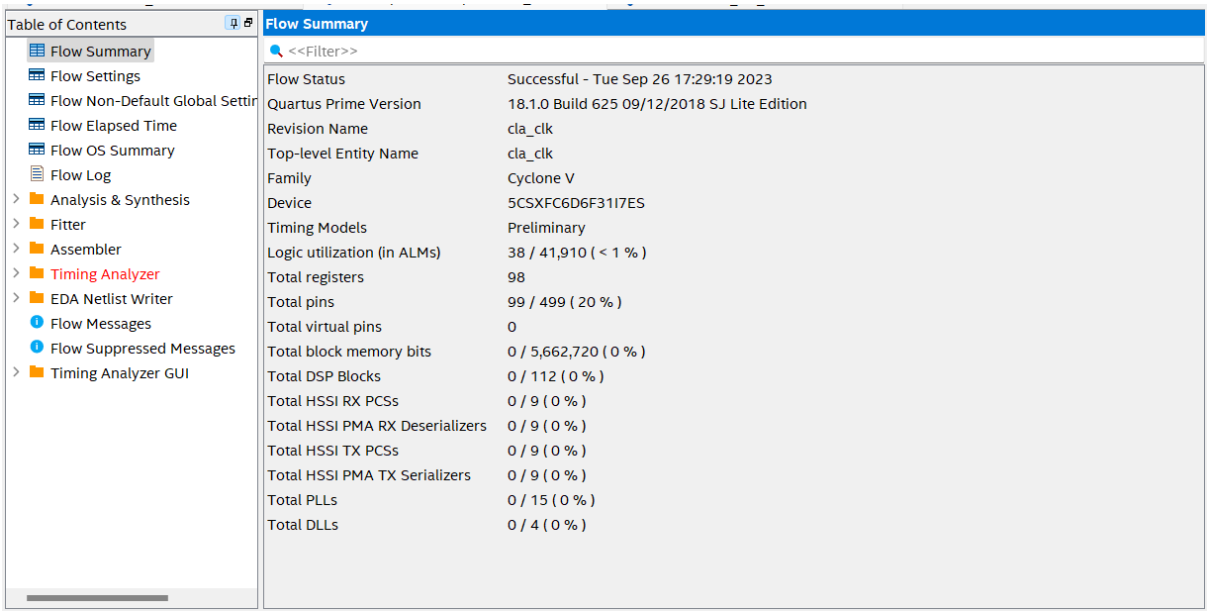

그림16. cla_clk Flow Summary

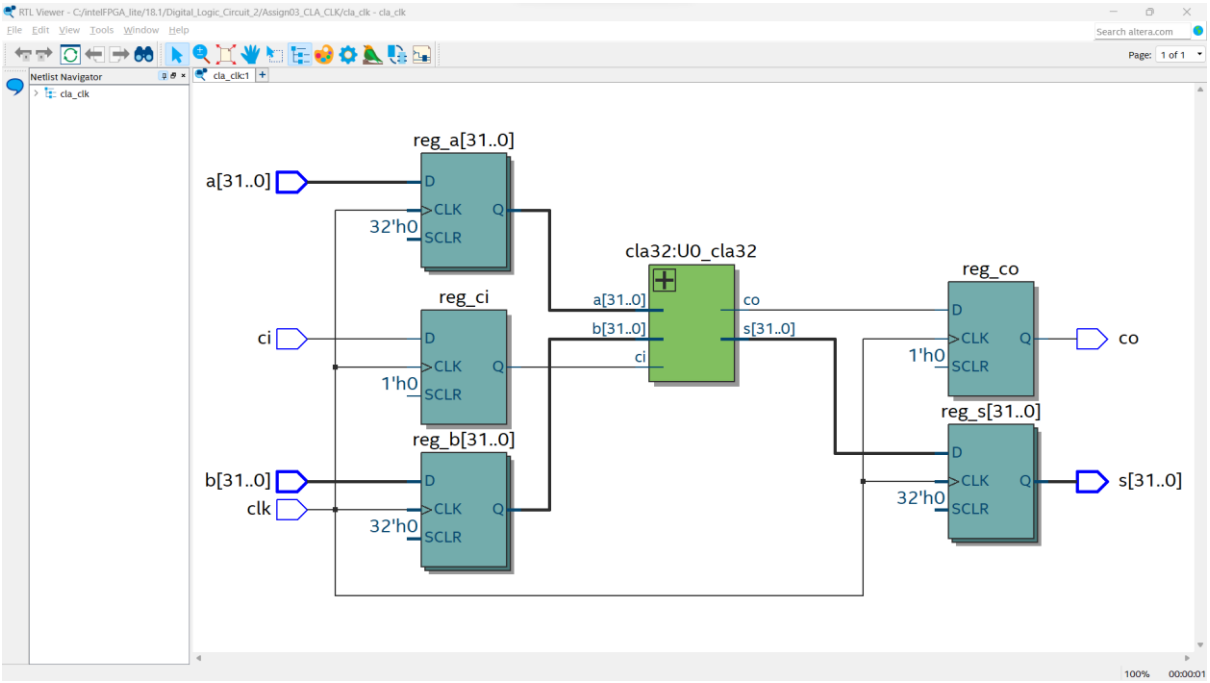

그림17. cla_clk RTL viewer

CLA clock은 cla32의 앞뒤로 register로 flip-flop을 추가하여 구현하였다.

C. Timing Analysis

1) RCA Clock

아래의 그림들은 rca_clk의 timing을 조정하기 전과 후를 나타낸다. 그림18과 그림20을 보면, slack의 값이 음수로 clock을 조절할 필요가 발생한다. 다음의 그림19과 그림 21은 clock을 조절한 이후의 모습으로, slack의 값이 양수로 violation이 발생하지 않았음을 확

인할 수 있다. 그림22에서 rca_clk 모듈의 최대 주파수(Fmax)는 109.81MHz인 것을 확인할 수 있다.

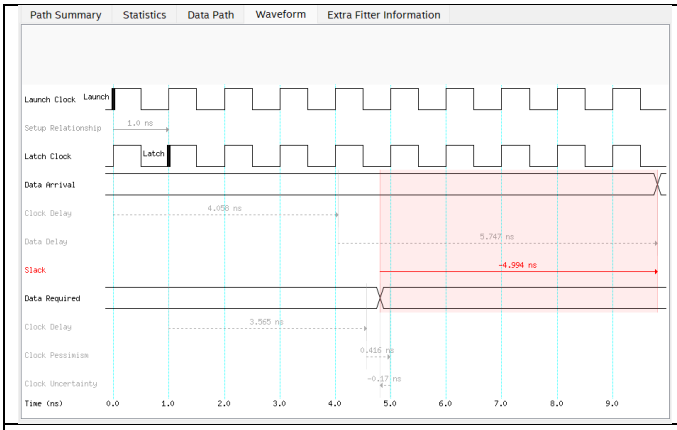

그림18. rca_clk waveform with violation

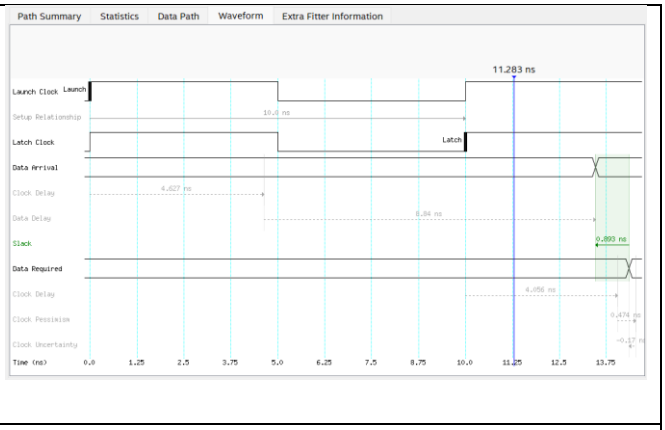

그림19. rca_clk waveform with no violation

Slow 1100mV 85C Model

	Clock	Slack	End Point TNS
1	clk	-4.994	-103.485

그림20. rca_clk Slack and TNS with violation

Slow 1100mV 100C Model

	Clock	Slack	End Point TNS
1	clk	0.893	0.000

그림21. rca_clk Slack and TNS with no violation

Slow 1100mV 100C Model

	Fmax	Restricted Fmax	Clock Name	Note
1	109.81 MHz	109.81 MHz	clk	

그림22. rca_clk Fmax

2) CLA Clock

아래의 그림들은 cla_clk의 timing을 조정하기 전과 후의 모습이다. 그림23과 그림25에서 slack의 값이 음수로 violation이 발생한 것을 확인할 수 있다. 이에 따라 clock을 조절하면, 다음의 그림24와 그림 26과 같이 slack의 값이 양수가 되어 violation이 발생하지 않았다. 그림27에서 cla_clk 모듈의 Fmax는 220.9MHz인 것을 확인할 수 있다.

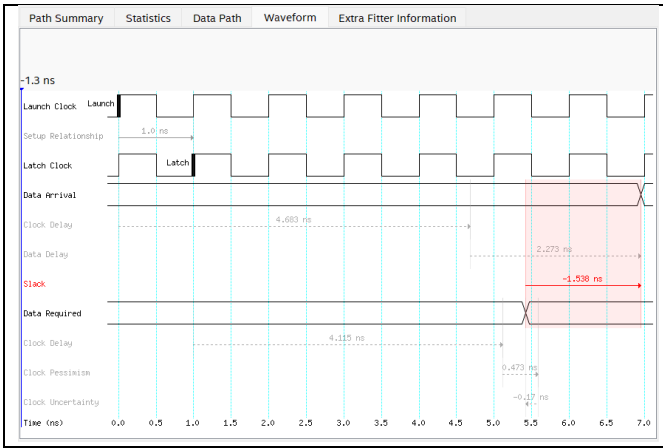

그림23. cla_clk waveform with violation

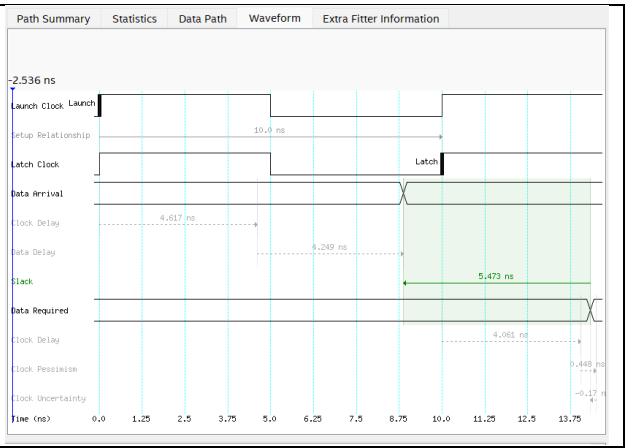

그림24. cla_clk waveform with no violation

Slow 1100mV 100C Model

	Clock	Slack	End Point TNS
1	clk	-4.640	-78.579

그림25. cla_clk Slack and TNS with violation

Slow 1100mV 100C Model

	Clock	Slack	End Point TNS
1	clk	5.473	0.000

그림26. cla_clk Slack and TNS with no violation

Slow 1100mV 100C Model

	Fmax	Restricted Fmax	Clock Name
1	220.9 MHz	220.9 MHz	clk

그림27. cla_clk Fmax

5. 고찰 및 결론

A. 고찰

1. CLA clock의 RTL simulation이 작동하지 않는 문제가 발생하였다. 오류 문구로 #error loading design이 나타났는데, 이는 testbench를 compile하는 과정에서 발견하지 못한 오류가 waveform을 실행시키자 문제가 된 것이었다. testbench에서의 오류는 input과 output의 값을 반대로 입력하여 발생한 문제였다. 이후에 제대로 입력하니 오류가 해결되었다.
2. RTL simulation을 실행할 때 Nativelink Error가 발생하였다. 이는 이전에 실행시킨 Model Sim window를 닫지 않고 다시 RTL simulation을 실행하면 발생하는 오류였다. 이전의 창을 닫고 다시 실행하였더니 성공적으로 실행되었다.
3. RCA Clock의 testbench waveform에서 tb_s와 tb_co의 값이 HiZ 값으로 나타났다. 이는 RCA Clock에서 reg의 값을 output port에 할당해주지 않아 생긴 문제였다. S와 co에 각각 reg_s와 reg_co를 할당해주니 해결할 수 있었다.
4. 프로젝트 cla_clk에 rca_clk 파일을 추가한 뒤, RTL simulation을 사용하여 tb_rca_clk를 실행시키면 cla_clk에 오류가 발생하였다는 에러가 나타났다. 이는 top level-entity name이 cla_clk인 상태로 tb_rca_clk를 실행시켜 발생한 오류였다. 이후에 top level-entity name을 rca_clk로 변경하였더니 해결되었다.
5. waveform에서 세번째 clock rising edge에서 결과값이 나타나지 않았다. 이는 이전의 값과 현재의 값이 같아 명시적으로 값을 표시하지 않은 것이고 waveform을 통해 이전의 값이 계속 유지되고 있다는 것을 확인할 수 있었다.
6. CLA32의 testbench에서 carry out의 값이 0으로 나오며 덧셈이 제대로 수행되지 않는 문제가 발생하였다. 이는 CLB4에서 c3와 co의 연산과정에서 cin값이 제대로 할당되지 않아 발생한 문제였다. 이후에 계산을 올바르게 수정하였더니 원하는 결과값을 얻을 수 있었다.

B. 결론

1. 32-bit CLA의 logic utilization은 38, 32-bit RCA은 31으로 rca32가 cla보다 크기가 작다는 것을 알 수 있다. 또한, 그림19에서 rca32의 data delay는 8.84ns이고 그림24 cla32의 data delay는 4.249ns로 cla가 rca보다 속도가 빠른 것을 확인할 수 있다.

6. 참고문헌

[1] 이준환 / Adder Design – CLA Adder / 광운대학교 / 2023년

[2] CLA / https://verilog-hdl-design.tistory.com/entry/CarryLookaheadAdder

[3] Verilog 기본 문법 / https://blog.naver.com/culonion/80022938473

[4] 파라미터 / https://zrr.kr/6HeP

[5] 결합연산자 / https://wh00300.tistory.com/166