컴퓨터 공학 기초 실험2 보고서

실험제목: Carry Look-ahead Adder

실험일자: 2023년 09월 25일 (월)

제출일자: 2023년 10월 4일 (수)

학 과: 컴퓨터정보공학부

담당교수: 이준환 교수님

실습분반: 월요일 0, 1, 2

학 번: 2020202031

성 명: 김재현

1. 제목 및 목적

A. 제목

Carry Look-ahead Adder

B. 목적

ripple carry adder가 계산이 완료될 때까지 시간이 많이 걸리는 단점을 보완하기 위한 가산기 carry look-ahead adder를 full adder와 4-bit carry look-ahead block을 사용하여 구현해본다. 4-bit carry look-ahead adder를 먼저 구현한 후 이를 이어 붙여 32-bit carry look-ahead adder를 구현한다.

2. 원리(배경지식)

1) Full Adder

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

Carry out은 generation signal, propagation signal을 통해 구할 수 있으므로 Full Adder에서는 s값만 출력하면 된다.

2) Ripple Carry Adder

N-bit ripple carry adder는 n-bit를 가지는 두 개의 수를 더하기 위한 간단한 형태의 가산기이다. 더하기 하려는 수의 bit 개수만큼 full adder를 연결하여 구현한다. 하지만 ripple carry adder의 경우 이전 full adder에서 carry out이 출력되어야 다음 full adder에서 연산을 진행할 수 있기 때문에, worst-case delay는 각 full adder의 delay를 모두 더한 값이므로 속도가 느리다.

3) Carry Look-ahead Block

$$G_i = A_i B_i$$
$$P_i = A_i + B_i$$

라고 정의 하고 full adder의 carry out에 적용하면 다음과 같습니다.

$$C_{i+1} = A_i B_i + (A_i + B_i)C_i = G_i + P_i C_i$$

이를 적용하여 4-bits CLA를 위한 carry를 미리 계산하면 다음과 같습니다.

$$C_1 = A_0 B_0 + (A_0 + B_0)C_0 = G_0 + P_0 C_0$$
$$C_2 = A_1 B_1 + (A_1 + B_1)C_1 = G_1 + P_1 C_1$$
$$C_3 = A_2 B_2 + (A_2 + B_2)C_2 = G_2 + P_2 C_2$$
$$C_{out} = A_3 B_3 + (A_3 + B_3)C_3 = G_3 + P_3 C_3$$

C_{out}*을* C_1, C_2, C_3*을 사용하여 정리하면*

$$C_{out} = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_0$$

4) 32-bits CLA with clock

32-bit cla에 앞 뒤로 flip-flop을 추가하여 clock과 연결하는 module이다. 해당 모듈에서 clock을 인가하여 valid한 결과가 나오는 데 필요한 clock period를 확인하는 것을 목표로 한다.

3. 설계 세부사항

1) full adder

이번 실습에서 사용할 full adder는 CLA에 사용될 가산기다. CLA에서는 propagation, generation signal을 통해 Carry out을 계산하기 때문에 full adder에서 carry를 출력할 필요가 없다. 따라서 이번 실습에서의 full adder는 a, b, ci를 입력 받고. 덧셈 결과인 s 만을 출력하도록 module을 설계한다.

2) 4-bit ripple carry adder

full adder를 4개 이어 붙여 만들어지는 가산기로서, full adder instance를 4개 선언하여 각 full adder의 co가 다음 full adder로 전달되도록 설계했다. a, b가 입력 값이고, 각 full adder에서 출력되는 carry와 덧셈 결과는 4비트 s, 3비트 c에 전달했고 마지막 full adder에서 출력되는 carry를 co에 저장했다.

3) 32-bit ripple carry adder

32-bit rca는 4-bit rca 8개를 이어 붙여 구성한다. 따라서 rca4 instance를 8개 선언하여 각 4-bit rca에서 출력되는 값들을 다음 4-bit rca의 입력 값으로 전달하도록 구현한다.

4) 4-bit carry look-ahead block

AND gate와 OR gate를 각각 4번씩 사용하여 generation, propagation signal을 4개씩 만들고, 배경지식에 작성한 carry 식을 활용하여 c1, c2, c3, co를 구한다.

5) 4-bit carry look-ahead adder

clb4에서 출력하는 carry들을 활용하여 full adder instance를 통해 각 비트 덧셈을 진행한다. cla4에서 출력하는 co는 clb4에서 출력하는 co와 동일하게 설계한다.

6) 32-bit carry look-ahead adder

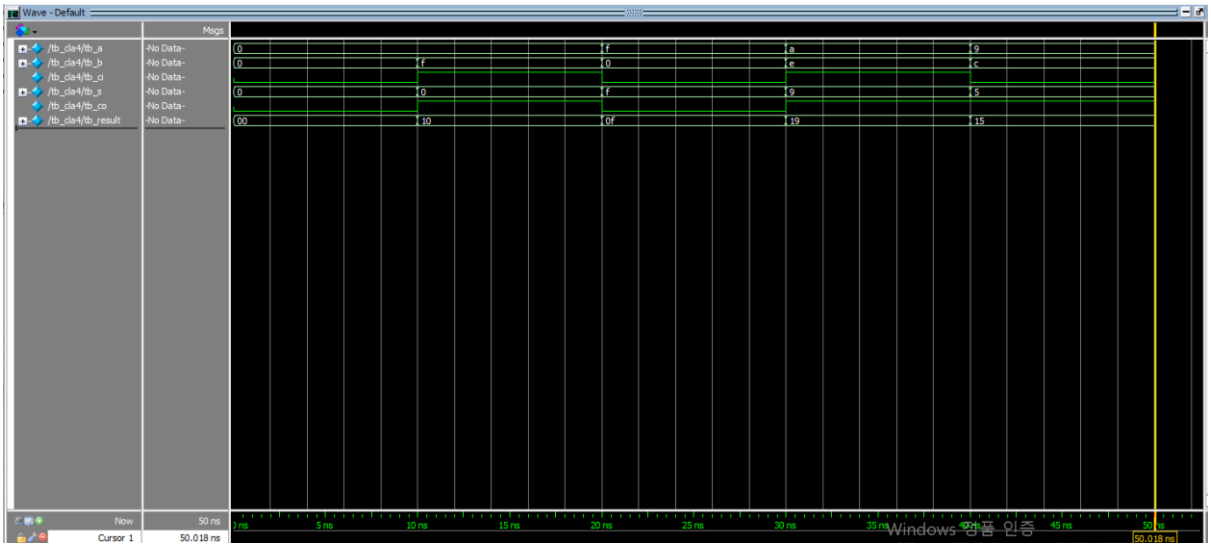
cla4를 8개 이어 붙여 구성한다. 이 때 cla4의 값을 다음 cla4에 전달할 수 있도록 wire 7개를 각 cla4 사이에 연결하도록 구현한다.

7) 32-bits CLA with clock, 32-bits RCA with clock

cla_clk, rca_clk module의 입력과 출력은 D-flipflop과 연결되어 있다. clk가 rising될 때마다 reg_a, reg_b, reg_ci, reg_s, reg_co에 a, b, ci, wire_s, wire_co 값을 저장한다.

4. 설계 검증 및 실험 결과

A. 시뮬레이션 결과



tb_cla4 testbench의 결과화면이다.

case1) 계산결과가 최솟값이 나오는 경우를 테스트했고 s가 0이고 co가 0으로, 옳은 값이 출력됐습니다.

case2) a + b가 최대값일 경우 ci가 1일 때 오버플로우가 발생하는 경우를 테스트했고 co가 1이고 s가 0으로, 옳은 값이 옳은 값이 출력됐습니다.

case3) 계산결과가 최댓값이 나오는 경우를 테스트했고 co가 0이고 s가 f로, 옳은 값이 출력됐습니다.

case4) a, b에 랜덤한 값을 넣었고, a + e + 1 = 19이므로 co가 1이고 s가 9이라는 옳은 값이 출력됐습니다.

case5) case4와 같이 a, b에 랜덤한 값을 넣었고, 9 + c + 0 = 15이므로 co가 1이고 s가 5라는 옳은 값이 출력됐습니다.

tb_cla32 testbench의 결과화면이다.

case1) 계산결과가 최솟값이 나오는 경우를 테스트했고, co와 s 둘 다 0으로 옳은 값이 출력됐습니다.

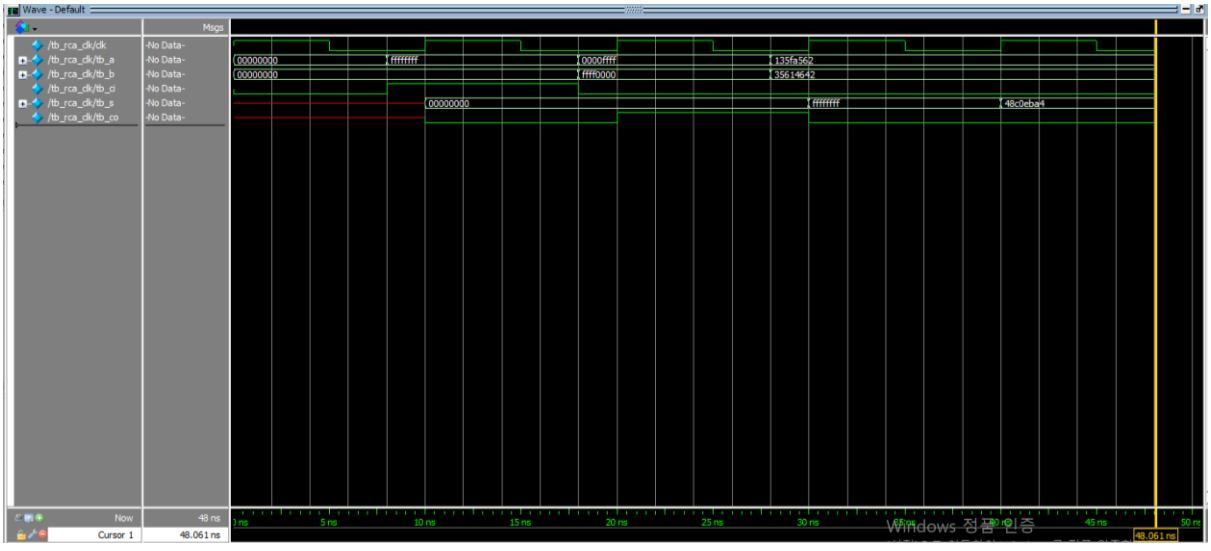
case2) a + b가 ffffffff가 나오도록 하고 ci를 1로 하여 모든 비트에서 carry out이 발생하도록 설계했습니다. co가 1, s가 00000000으로 옳은 값이 출력됐습니다.

csae3) s가 최대값이 출력되도록 설계했습니다. co가 0, s가 ffffffff로 옳은 값이 출력됐습니다.

case4, 5) a, b에 랜덤한 수를 넣었고 그 결과 옳은 값이 출력됐습니다.

tb_cla_clk testbench의 결과화면이다.

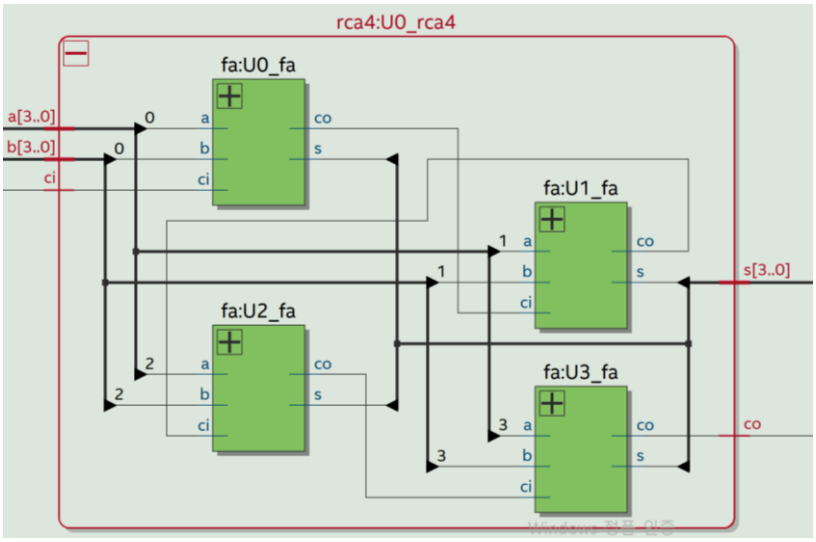
처음에 clk를 1로 설정해놓고 시작한다. 미리 설정해놓은 parameter clock주기 값 10을 주기로 값이 들어가도록 설계한다. 하지만 처음 step값은 -2~4정도를 해줌으로써 time delay를 방지할 수 있다. 그러고 마지막에 무한히 진행하는 것을 방지하기 위해 stop을 넣어준다. 그리고 잘 작동하는지 test를 위해 input값들을 넣어주었다. clock주기에 따라 잘 작동하는지 확인했다.



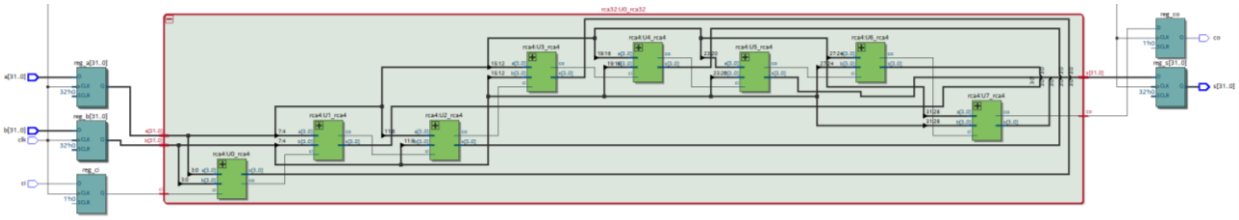
tb_rca_clk의 testbench 결과 화면이다.

tb_cla_clk와 testbench의 case를 동일하게 설정했는데 waveform 확인결과 tb_rca_clk와 tb_cla_clk가 동일한 것을 확인할 수 있었다.

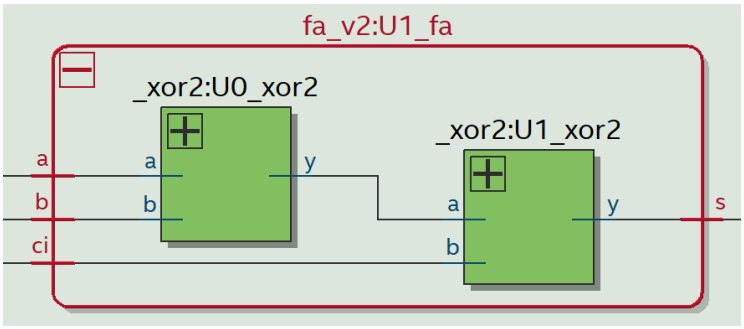
B. 합성(synthesis) 결과



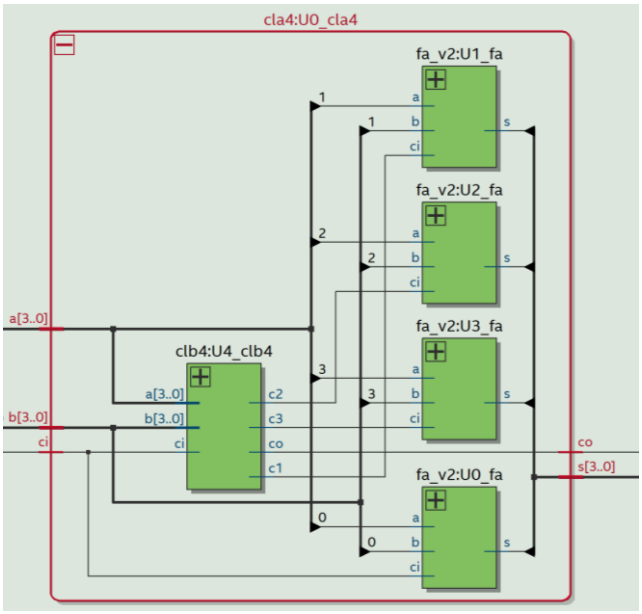
full adder 4개를 이어 붙인 구조인 4bit rca4 입니다.



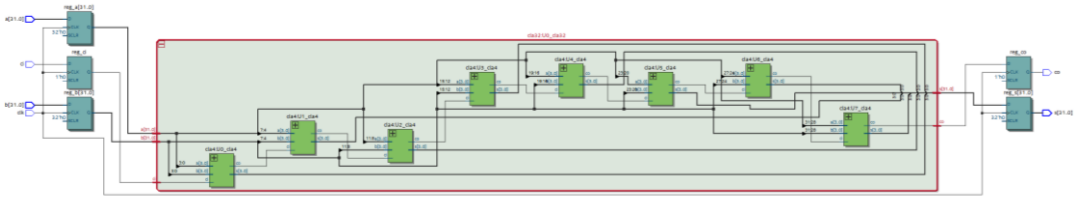
rca4를 8개 이어 붙여 32bit Ripple carry adder를 구현하고 D-flipflop과 이어 붙여 rca_clk를 구현했습니다.



Carry Look Ahead에 사용되는 full adder v2 module입니다. carry out을 출력할 필요가 없기 때문에 sum 결과만을 출력하기 위해 xor gate 두개를 사용하여 회로를 구성했습니다.



4-bit CLA module 입니다. clb4에서 출력되는 carry out들로 각 bit의 덧셈을 진행해 출력하고, co도 출력합니다.



cla_clk module 입니다. D-flipflop과 cla4 8개를 사용하여 32-bit cla_clk를 구현했습니다.

<Flow Summary>

Flow Summary

<<Filter>>

Flow Status	Successful - Wed Oct 04 14:05:35 2023
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Lite Edition
Revision Name	cla_clk
Top-level Entity Name	rca_clk
Family	Cyclone V
Device	5CSXFC6D6F31C6
Timing Models	Final
Logic utilization (in ALMs)	31 / 41,910 (< 1 %)
Total registers	98
Total pins	99 / 499 (20 %)
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 %)

rca_clk의 flow summary입니다. total pins 는 32bit a, 32bit b, 32bit s, ci, co, clk로 총 32+32+32+1+1+1=99가 나온 것을 확인할 수 있고, total registers는 32bit reg_a, 32bit reg_b, 32bit reg_s, reg ci, reg co로 총 32+32+32+1+1=98이 나온 것을 확인할 수 있습니다. logic utilization 값은 31인 것을 확인할 수 있습니다.

Flow Summary

<<Filter>>

Flow Status	Successful - Wed Oct 04 14:18:14 2023
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Lite Edition
Revision Name	cla_clk
Top-level Entity Name	cla_clk
Family	Cyclone V
Device	5CSXFC6D6F31C6
Timing Models	Final
Logic utilization (in ALMs)	43 / 41,910 (< 1 %)
Total registers	98
Total pins	99 / 499 (20 %)
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 %)

cla_clk의 flow summary입니다. total pins 는 32bit a, 32bit b, 32bit s, ci, co, clk로 총 32+32+32+1+1+1=99가 나온 것을 확인할 수 있고, total registers는 32bit reg_a, 32bit reg_b, 32bit reg_s, reg ci, reg co로 총 32+32+32+1+1=98이 나온 것을 확인할 수 있습니다. logic utilization 값은 43인 것을 확인할 수 있습니다.

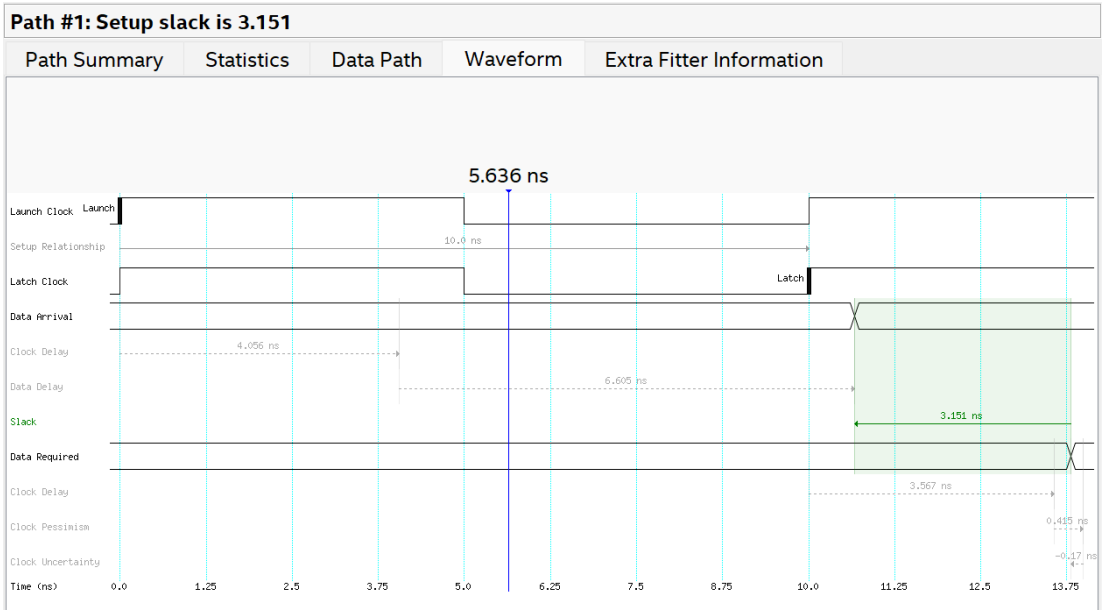
<Timing Analysis>

Slow 1100mV 85C Model

	Clock	Slack	End Point TNS
1	clk	3.151	0.000

Slow 1100mV 85C Model

	Fmax	Restricted Fmax	Clock Name	Note
1	146.01 MHz	146.01 MHz	clk	



rca_clk의 timing analysis 결과입니다.

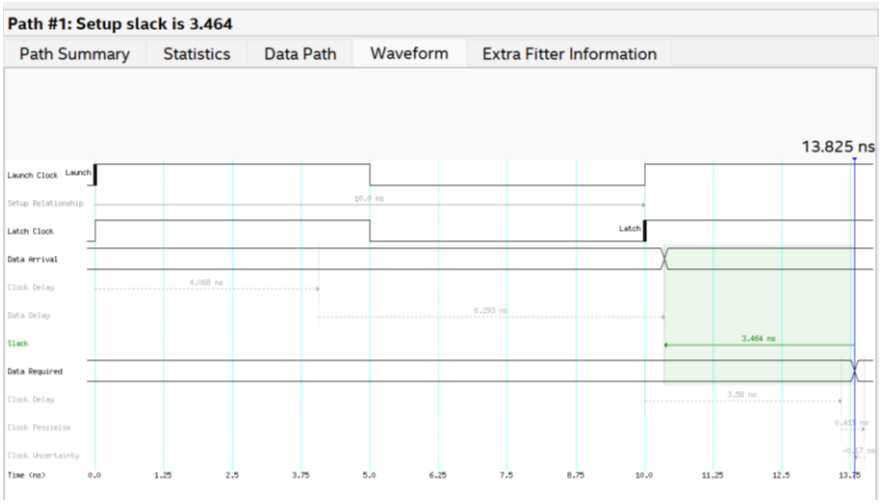
clock의 기존 period 값에 slack 값을 더하고 여유있게 2~3을 추가로 더해준 후 진행했더니 잘 작동하는 것을 확인할 수 있습니다.

Slow 1100mV 85C Model

	Clock	Slack	End Point TNS
1	clk	3.464	0.000

Slow 1100mV 85C Model

	Fmax	Restricted Fmax	Clock Name	Note
1	153.0 MHz	153.0 MHz	clk	



cla_clk 역시 rca_clk와 동일한 방법으로 진행했다.

logic utilization을 비교해보면 rca_clk은 31, cla_clk은 43인 것으로 보아 cla_clk의 area가 더 큰 것을 확인할 수 있습니다. slack 차이를 비교함으로써 속도 비교를 할 수 있는데, clock period 값을 동일하게 한 후 slcak이 rca_clk는3.151, cla_clk는 3.464로, cla_clk가 더 큰 진동수 즉, 더 빠른 속도를 가진다는 것을 알 수 있습니다.

5. 고찰 및 결론

A. 고찰

Timing Analysis를 처음 접하다보니 사용 방법이나 해석 방법 면에 있어서 많이 헤매게 됐던 것 같습니다. 또한 CLA 뿐만 아니라 RCA도 번갈아 컴파일해가면서 비교해야 했기 때문에 매번 top level module과 testbench를 변경해주어야 하는 부분에서도 잦은 어려움이 있었습니다. 이번 과제를 통해 항상 top level module을 확인하는 습관을 들일 수 있게 됐습니다.

B. 결론

rca_clk, cla_clk testbench를 구성할 때 clk 주기를 10으로 했습니다. 그 이유는 clk가 변화하기 전, setup time을 확보할 수 있도록 하기 위해 첫번째 입력 값 변화 시기를 8로 설정했습니다.

rca는 이전 full adder의 계산결과를 다음 full adder가 넘겨받아야 연산이 진행된다는 점에서 full adder의 모든 연결지점에서 하나하나 delay가 중첩되고, cla는 carry out을 clb에서 한번에 계산해주기 때문에, 속도 면에서 rca보다 cla가 더 빠르다는 것을 알 수 있습니다. 따라서 여유시간이라는 의미를 내포하는 slack과 주기의 반비례인 frequency에서

cla가 더 큰 값을 가진다는 사실을 알 수 있다. 하지만 propagation, generation signal을 구성하는데 있어서 area는 cla가 더 많이 차지한다는 것을 알 수 있다.

6. 참고문헌

이준환 교수님/컴퓨터공학기초실험2/광운대학교(컴퓨터정보공학부)/2023