



# Project

---

## Adder Delay

과목명	하드웨어소프트웨어 통합설계
담당교수	이준환 교수님
학과	컴퓨터정보공학부
학년	3학년
학번	2019202009
이름	서여지
제출일	2021.12.14(화)

## □ Introduction

SystemC를 이용하여 adder를 설계하는 것을 목표로 한다. 상위수준에서 모델링 되는 adder는 회로의 delay를 고려하여 작성한다. 제시된 adder는 RCA, CLA, CSA의 3종류이며, CBA를 추가하여 총 4가지 adder를 구현하였다. 구현한 adder는 별도의 tb모듈을 이용하여 입력을 제어하고, 출력을 확인한다.

## □ Project specification

RCA, CLA, CSA, CBA 의 구조를 확인하여 delay 를 계산하였다. 계산한 delay 를 이용하여 adder 를 모델링하고, SystemC 를 이용하여 구현한다. 구현한 adder 를 확인하기 위한 tb 모듈을 작성하여 동작을 확인한다. adder 와 testbench 의 입출력 port 는 제시된 것에 따른다.

SystemC 를 이용한 구현은 상위수준에서 이루어지므로 gate 의 동작을 기술할 필요는 없으나, 구조에 맞는 delay 를 분석하여 delay 만큼 wait 하도록 구현한다. 사용하는 clock 은 FA 의 delay 에 해당하며, 1ns 로 정의한다. 동시에 FA 의 delay 는  $4 T_g$ 에 해당한다.

adder 의 A,B,Cin 입력과 result, Cout 출력은 각각 valid, ready 의 handshake 신호를 이용한다. 각각의 신호를 이용하여 tb 모듈과 유효한 신호를 주고받는 것을 확인하며, valid 와 ready 가 모두 1 일 경우에 모듈간 신호가 정상적으로 전달된다.

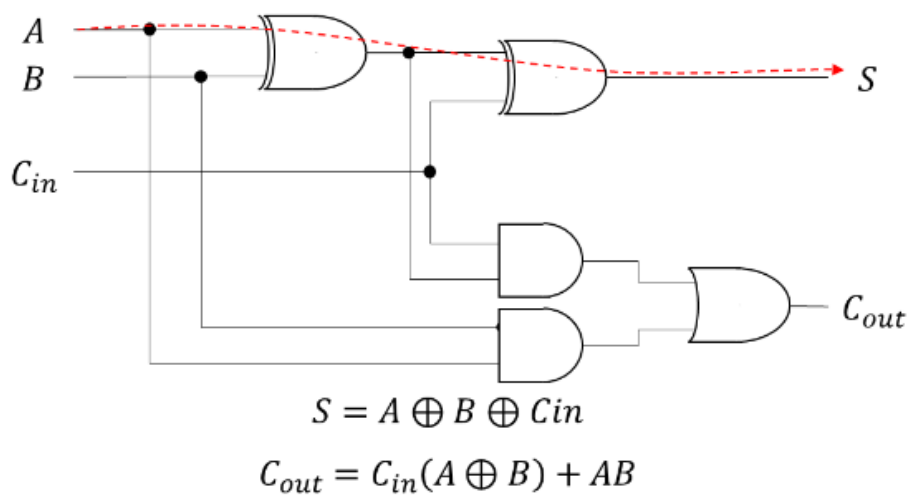
tb 모듈은 형식에 맞게 작성된 Simulation.txt 에서 A,B,Cin 의 값을 읽고, adder 에 전달한다. adder 에서 출력되는 결과는 형식에 맞추어 console 에 출력한다.

## □ Design details

### 1) RCA - Ripple Carry Adder

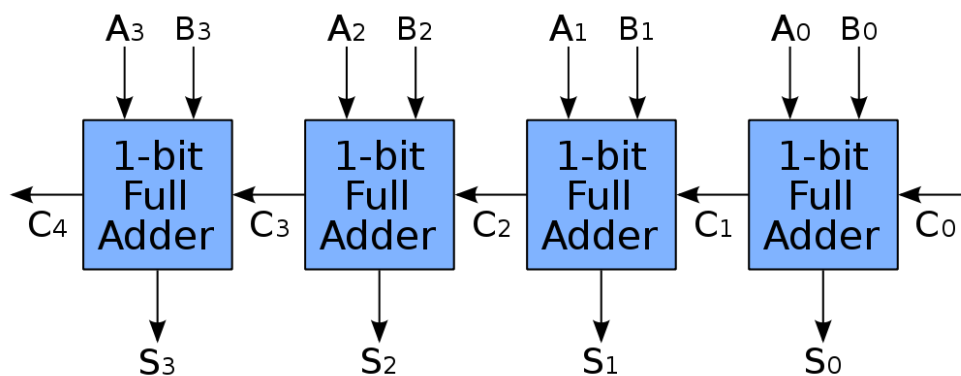
RCA는 FA의 Cout을 다음 bit을 계산하는 FA의 Cin으로 연결하는 것을 통해 Carry가 ripple되는 형태를 갖는 가산기이다. N의 크기만큼 일정한 delay가 추가되기 때문에 적은 bit의 계산에서는 별도의 처리를 하는 다른 가산기보다 빠를 수 있지만, 많은 bit의 덧셈은 FA의 delay를 가려서 계산 시간을 줄이는 다른 가산기보다 느리다.

RCA의 delay를 구하는 과정은 다음과 같다.



[그림 1. FA diagram]<sup>1</sup>

FA의 delay는 Sum을 구하는 것에  $4 T_g$ 이고, Cout을 구하는 것에  $4 T_g$ 이다.



[그림 2. 4bit RCA]<sup>2</sup>

4bit RCA는 위의 그림2와 같이 4개의 FA를 연결한 구조를 갖는다. 이러한 구조에서 FA의 A xor B 연산은 Cin이 입력되기전에도 병렬적으로 처리될 수 있다. 따라서 두 번째 이상의 FA에서 Cin이 입력된 이후  $2T_g$ 만에 Sum과 Cout이 출력된다. 따라서 4bit RCA의 delay는 첫 번째 FA의 delay  $4T_g$ 와 3개의  $2T_g$ 를 더한  $10T_g$ 이다.

32bit RCA는 4bit RCA의 Cout을 다음 4bit RCA의 Cin으로 연결하는 구조이고, 총 8개의 4bit RCA를 사용한다. 이 경우 4bit RCA와 동일하게 첫 번째 FA에서만  $4T_g$ 의 delay를 갖고, 나머지 31개의 FA는 Cin이 입력된 이후  $2T_g$ 만에 결과를 출력한다.  $4T_g + 31 \cdot 2T_g = 66T_g$

FA	Sum	$4 T_g(2T_g)$	$4 T_g = 1\text{Cycle}$
	Cout	$4 T_g(2T_g)$	
RCA4	Sum	$10 T_g$	$10 T_g \leq 3\text{Cycle}$
	Cout	$10 T_g$	
RCA32	Sum	$66 T_g$	$66 T_g \leq 17\text{Cycle}$
	Cout	$66 T_g$	

## 2) CLA - Carry Lookahead Adder

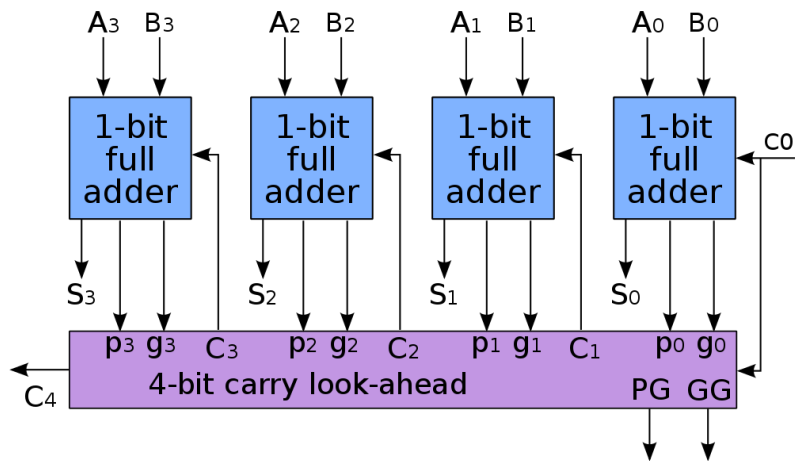
CLA는 carry를 따로 계산하는 CLB를 이용하여 계산시간을 줄인 가산기이다. CLB에서는 carry를 계산하기 위해 A와 B를 이용하여 G(generate)와 P(propagate)를 구하여 이용한다.

g는 A와 B에 의해 carry가 발생하는 경우이고, A와 B가 모두 1인 경우를 의미한다. ( $G = A \& B$ )

p는 이전 bit의 carry(Cin)가 다음 bit로 전달(Cout)되는 경우이고, A와 B중 하나 이상이 1인 경우를 의미한다. ( $P = A + B$ )

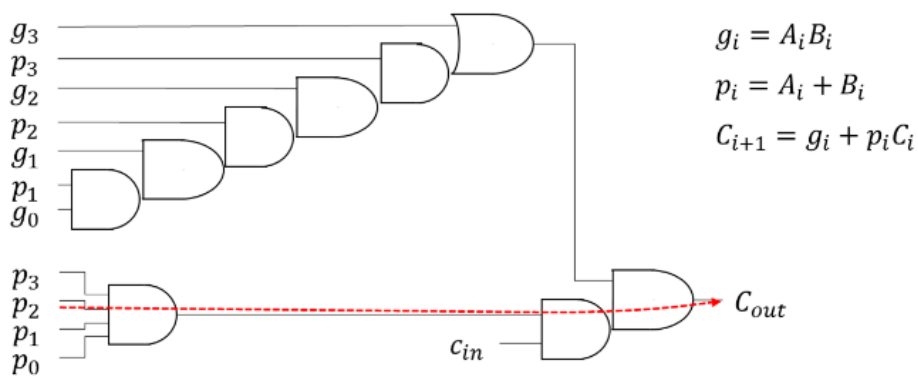
각 bit (i)에서 다음 bit(i+1)로 넘어가는 carry를  $\text{Cout}_{i+1} = G_i + P_i \text{Cin}$ 으로 구할 수 있다.

또한 다음 bit에서는 이전의 carry를 구한 식을 이용하여  $\text{Cout}_{i+2} = G_{i+1} + P_{i+1} (G_i + P_i \text{Cin})$ 으로 구할 수 있다. 이 과정을 통해 FA의 Cout을 기다릴 필요 없이 입력으로 전해진 Cin만을 이용하여 block 내의 carry를 모두 구할 수 있다.



[그림3. 4bit CLA]<sup>3</sup>

4bit CLA는 위와 같이 4개의 FA와 4bit CLB를 갖는다. CLB의 구조는 다음 그림4와 같다.



[그림4. CLB block diagram]<sup>4</sup>

4bit CLA에서의 critical path는 input을 이용하여 CLB에서 C3을 구한 뒤, 그 결과를 이용한 FA가 동작하는 것이다. CLB의 계산을 위한 g와 p는 각각 and와 or gate를 하나 사용한다. (1T<sub>g</sub>) C3을 구하기 위해 pg block에서 and와 or연산을 반복한다. (4 T<sub>g</sub>) pg block과 p0p1p2Cin의 결과를 or하여 c3을 얻는다. (1 T<sub>g</sub>) C3의 결과를 이용하여 FA의 계산결과를 얻는다. (2 T<sub>g</sub>) 위의 과정에서 총 8 T<sub>g</sub>의 delay가 발생함을 알 수 있다.

32bit CLA는 32bit RCA와 유사하게 4bit CLA를 8개 연결한다. 4bit CLA의 Cout을 다음 4bit CLA의 Cin으로 연결한다. 32bit CLA에서의 critical path는 input data를 이용하여 첫번째 4bit CLA의 CLB에서 Cout을 계산하고, 그 결과가 나머지 7개의 4bit CLA의 CLB를 거쳐 31번째 bit의 FA계산에 이용되는 것이다.

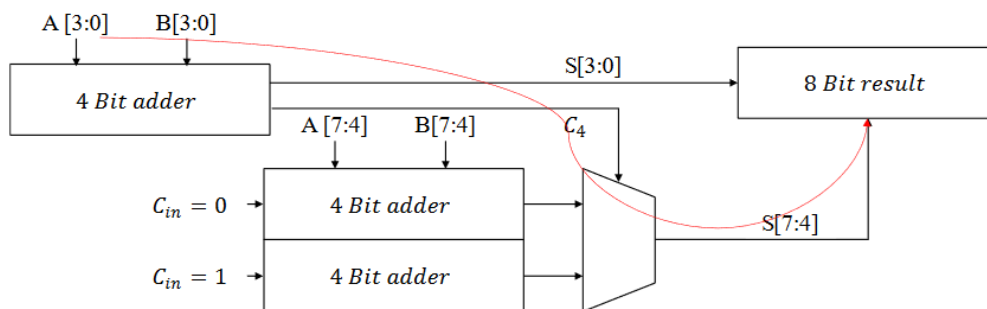
첫번째 4bit CLA에서의 Cout은 8T<sub>g</sub>만에 계산할 수 있음을 위에서 살펴보았다. 나머지 7개의 CLA에서는 pg block의 계산이 이미 완료된 상태이므로, 그림4의 하단에 위치한 Cin이 입력되어 2개의

gate를 거쳐 Cout으로 출력되는 path를 지난다. 마지막 4bit CLA에서는 Cin을 이용하여 31번째 bit의 sum을 계산하는 FA를 거치는 path가 critical path가 된다. 처음과 마지막의 CLA를 제외하고, 6개의 CLA에서는 2개의 gate만을 통과한다. ( $6 \times 2T_g$ ) 마지막 CLA에서는 Cin이 입력되어 C3을 구하는 2개의 gate를 통과하고( $2T_g$ ), C3을 이용한 FA에서 다시  $2T_g$ 의 시간이 걸린다. 위의 과정을 통해 32bit CLA에서는 총  $24 T_g$ 의 delay가 걸리는 것을 확인할 수 있다.

CLA4	Sum	$8T_g$	$8T_g = 2\text{Cycle}$
	Cout	$8 T_g$	
CLA32	Sum	$24 T_g$	$24 T_g = 6\text{Cycle}$
	Cout	$22 T_g$	

### 3) CSA - Carry Select Adder

CSA는 각 자리에서 carry가 1인 경우와 0인 경우를 모두 계산한 뒤 mux를 이용하여 결과를 선택하는 구조의 가산기이다. 두 경우를 계산한 뒤 이전 bit의 carry out에 따라 선택한다. 각 FA의 delay가 숨겨지고, MUX delay가 남는다. 계산 시간이 줄어드는 장점과, 필요한 FA와 MUX가 많은 단점이 있다.



[그림5. 8bit CSA diagram] <sup>5</sup>

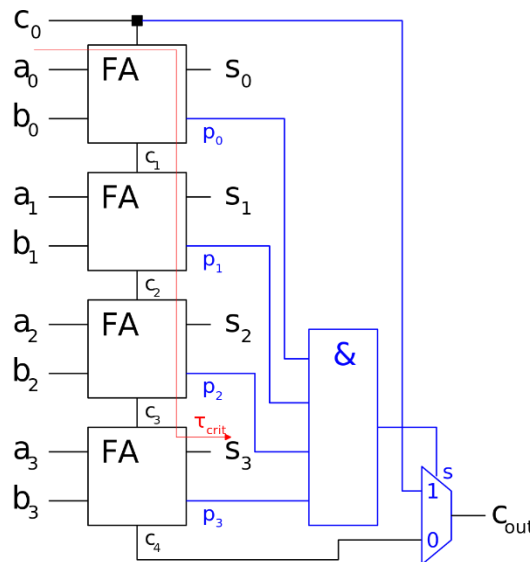
4bit CSA에서의 critical path는 input을 이용하여 첫 번째 FA가 동작한 뒤, carry out의 값에 따라 3개의 MUX에서 계산 결과를 선택하는 것이다. 7개의 FA는 동시에 동작하고( $4 T_g$ ), 3개의 MUX는 순차적으로 동작한다. ( $3 \times 2 T_g$ ) 따라서 4bit CSA는  $10 T_g$ 의 delay를 갖는다.

32bit CSA는 15개의 4bit CSA를 이용하여 구성한다. 다른 가산기와 동일하게 4bit CSA의 Cout을 다음 4bit CSA의 Cin으로 연결한다. 32bit CSA에서 모든 4bit CSA 는 동시에 동작하고 ( $10 T_g$ ), 7개의 MUX가 순차적으로 동작한다. ( $7 \times 2 T_g$ ) 따라서 32bit CSA는  $24T_g$ 의 delay를 갖는다.

CSA4	Sum	$10 T_g$	$10T_g \leq 3\text{Cycle}$
	Cout	$10 T_g$	
CSA32	Sum	$24 T_g$	$24 T_g = 6\text{Cycle}$
	Cout	$24 T_g$	

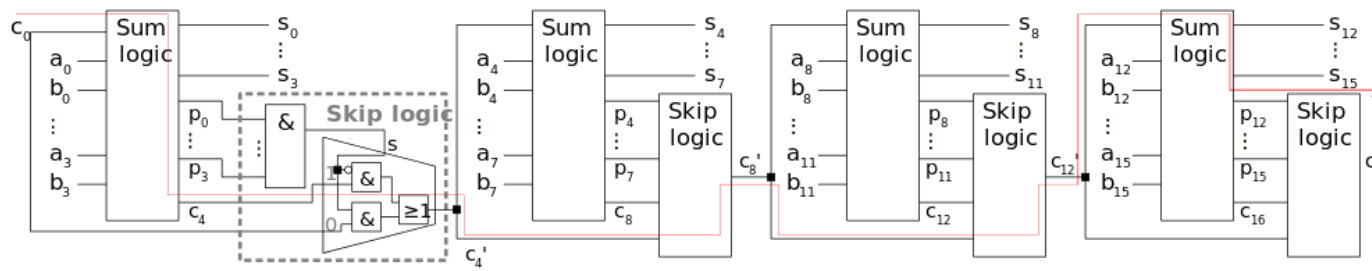
#### 4) CBA - Carry Bypass Adder (Carry Skip Adder)

CBA는  $C_{in}$ 을 그대로  $C_{out}$ 으로 출력하는 경우를 확인하여  $C_{out}$ 을 빠르게 구할 수 있는 가산기이다. CLA에서 이용하였던 P(propagate)와 G(generate)와 더불어 D(delete)를 이용한다. 그러나 이 경우 사용하는 P는 G와의 구분을 위해  $A \text{ xor } B$ 의 값을 이용하여 A와 B중 하나의 값이 1일 때를 의미한다. D는 carry가 다음 bit로 전달되지 않도록 한다. A와 B가 모두 0인 경우가 해당한다. block 단위의 덧셈기에서 모든 bit의 p가 1이라면,  $C_{in}$ 을 그대로  $C_{out}$ 으로 출력할 수 있음을 이용한다.



[그림6. 4bit CBA]<sup>6</sup>

4bit CBA에서 critical path는 4개의 FA를 지나 MUX ( $2T_g$ )를 거치는 것이다. 4개의 FA는 순차적으로 계산되므로( $4 \times 4T_g$ ) 4bit CBA의 delay는  $18 T_g$ 가 된다. 또한  $C_{in}$ 이 입력되어 skip로직을 통해  $C_{out}$ 으로 출력되는 시간은 p를 계산하기 위한  $2T_g$ 와  $p_0p_1p_2p_3$ 를 얻는 and에서  $1T_g$ , MUX를 거치는  $2T_g$ 를 더한  $5 T_g$ 이며, p를 구하는 xor과 and의 delay는  $C_{in}$ 이 입력되기 전에 미리 계산될 수 있다.



[그림6. 16bit CBA]<sup>7</sup>

32bit CBA 또한 Cout을 다음 adder의 Cin으로 연결하는 구조이다. 32bit CBA에서 critical path는 첫 번째 adder와 마지막 adder에서 4개의 FA를 통과하고, 나머지 6개의 adder에서는 FA를 거치지 않고 skip되는 path이다. carry를 계산하지 않고 skip할 수 있는 이유는 다음과 같다.

어떤 bit i에 대해서  $p[i]$ 가 1인 경우 i에서 Cin은 Cout으로 전달된다.

$g[i]$ 가 1인 경우 새로운 carry가 발생하여 Cout은 1이다.

$d[i]$ 가 1인 경우 Cin은 사라지고, Cout은 0이다.

위의 세 가지 경우를 통해 FA를 통해 carry를 계산하는 과정을 생략하고 Cout을 구할 수 있다.

따라서 32bit CBA에서의 delay는 첫 번째 CBA를 통과하는  $18 T_g$ , 이후 6개의 CBA를 skip하는  $6 * 2 T_g$ , 마지막 CBA에서 4개의 FA를 통과하며 SUM을 계산하는  $4 * 4 T_g$ 에 의해  $46 T_g$ 이다.

CBA4	Sum	$16 T_g$	$18 T_g \leq 5 \text{ Cycle}$
	Cout	$18 T_g$	
CBA32	Sum	$46 T_g$	$46 T_g \leq 12 \text{ Cycle}$
	Cout	$32 T_g$	



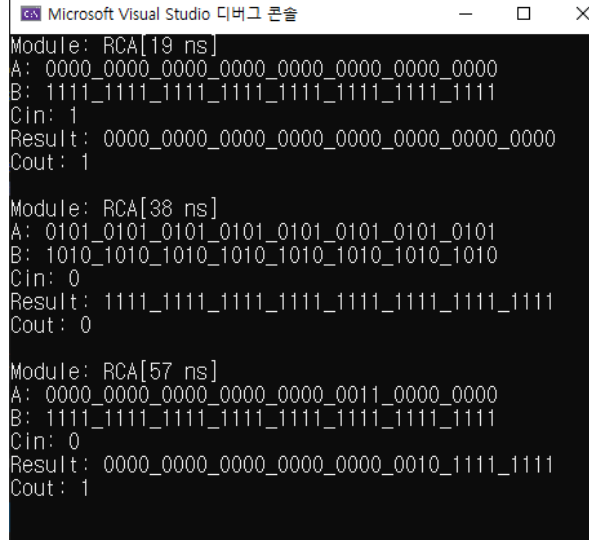
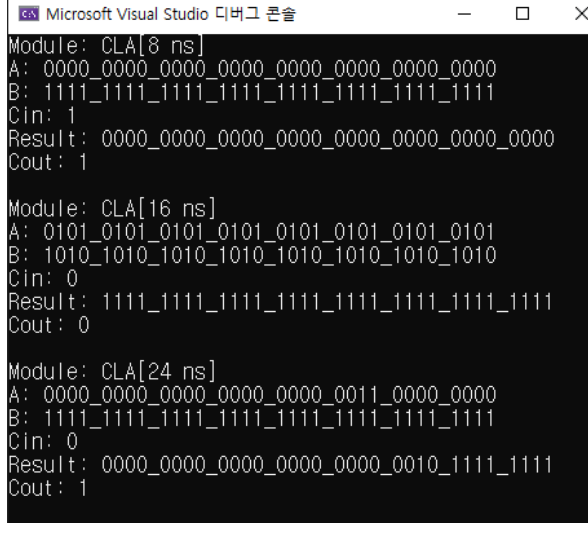
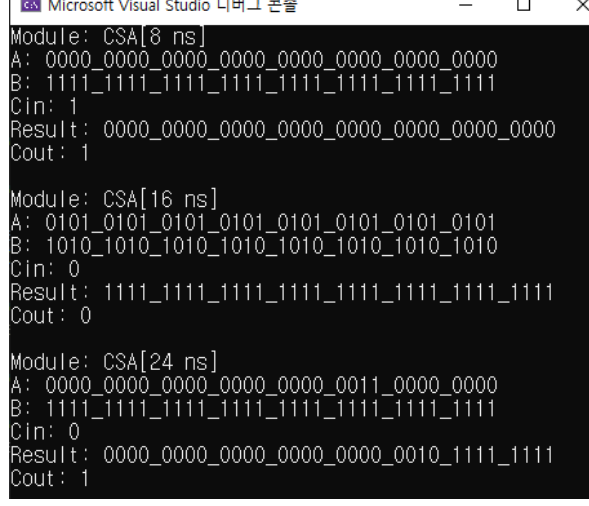
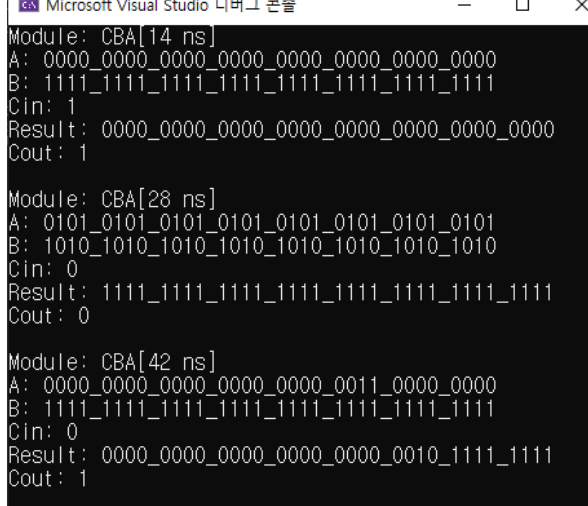
## □ Design verification strategy and results 설계 전략, 결과

RCA, CLA, CSA, CBA 각각의 4개의 프로젝트를 생성하였다. 각 프로젝트는 adder 모듈을 제외하면 모두 동일하다. main.cpp, SYSTEM.h, tb.h, tb.cpp를 공통으로 갖는다. 또한 모든 adder는 delay의 차이만을 갖고, 동일한 동작을 한다.

main.cpp는 sc\_main에서 SYSTEM의 인스턴스를 생성하여 모듈을 실행시키는 동작을 한다. SYSTEM.h에는 SYSTEM모듈이 정의되어있다. SYSTEM모듈은 adder와 tb의 포인터와 함께 clock, input, output signal을 멤버로 갖는다. SYSTEM모듈의 생성자에서는 adder와 tb의 인스턴스를 생성하여 모든 signal을 모듈의 포트에 연결한다.

각 adder의 헤더파일에는 adder모듈이 정의되었으며, input, output 포트와 adder\_main함수를 갖는다. adder모듈의 인스턴스가 생성되었을 때, 생성자는 adder\_main함수를 실행시킨다. adder.cpp파일에는 adder\_main함수의 동작이 기술되어있다. adder\_main함수는 실행되면 input의 ready 신호를 1로 설정하여 data를 입력받을 수 있음을 tb에 알린다. 신호를 받았는지 여부를 나타내는 set\_signal배열의 값을 0으로 초기화한다. A,B,Cin신호를 모두 받기 전까지 wait()을 반복한다. 아직 수신하지 않은 입력신호의 valid가 1이 되면 wait을 중단하고 해당 신호를 읽은 뒤, ready신호를 0으로 만들고, 수신여부를 의미하는 set\_signal배열을 1로 만든다. A,B,Cin이 모두 입력되면 값을 더해 result와 Cout을 구한다. 각 adder의 delay만큼 wait한다. 이후에 출력인 result와 Cout 포트에 값을 쓰고, valid를 1로 설정한다. 처음 input을 입력받는 과정과 유사하게, tb에서 result와 Cout을 모두 읽을 때까지 wait을 반복한다. 아직 송신하지 않은 출력신호의 ready가 1이 되면 wait을 중단하고 valid를 0으로 만든 뒤, 송신여부를 의미하는 set\_signal의 값을 1로 만든다.

tb의 헤더파일에는 tb모듈과 함께 inputFile명과 testCase의 수가 정의되어 있다. tb모듈에는 input, output포트와 source, sink함수가 존재한다. source함수는 inputFile의 내용을 읽어 입력을 생성하고, adder에 전달한다. source함수는 동작이 시작되면 inputFile의 내용을 testCase의 수만큼 반복하여 읽어 A, B, Cin를 모두 저장한다. 이후 adder모듈에서 output을 보낸 방식과 같은 방식으로 adder모듈에 A,B,Cin을 송신한다. A,B,Cin의 값을 쓰고, valid를 1로 설정한다. adder모듈이 A,B,Cin을 모두 받을 때 까지 wait을 반복하고, 아직 송신하지 않은 출력신호의 ready가 1이 되면 wait을 중단하고 valid를 0으로 만든 뒤, 송신여부를 의미하는 set\_signal의 값을 1로 만든다. 반대로 sink함수는 adder의 계산결과를 수신하여 console에 출력한다. result, Cout의 ready 신호를 1로 설정하여 data를 입력받을 수 있음을 adder에 알리고, valid가 1이 될 때까지 wait한다. valid가 1이 되면 해당 신호를 읽은 뒤, ready신호를 0으로 만들고, 수신여부 정보를 갱신한다. result와 Cout을 모두 받으면 이전에 저장했던 A,B,Cin과, sc\_time\_stamp의 값과 함께 출력한다.

RCA	CLA
 <pre> Module: RCA[19 ns] A: 0000_0000_0000_0000_0000_0000_0000_0000 B: 1111_1111_1111_1111_1111_1111_1111_1111 Cin: 1 Result: 0000_0000_0000_0000_0000_0000_0000_0000 Cout: 1  Module: RCA[38 ns] A: 0101_0101_0101_0101_0101_0101_0101_0101 B: 1010_1010_1010_1010_1010_1010_1010_1010 Cin: 0 Result: 1111_1111_1111_1111_1111_1111_1111_1111 Cout: 0  Module: RCA[57 ns] A: 0000_0000_0000_0000_0000_0011_0000_0000 B: 1111_1111_1111_1111_1111_1111_1111_1111 Cin: 0 Result: 0000_0000_0000_0000_0000_0010_1111_1111 Cout: 1 </pre>	 <pre> Module: CLA[8 ns] A: 0000_0000_0000_0000_0000_0000_0000_0000 B: 1111_1111_1111_1111_1111_1111_1111_1111 Cin: 1 Result: 0000_0000_0000_0000_0000_0000_0000_0000 Cout: 1  Module: CLA[16 ns] A: 0101_0101_0101_0101_0101_0101_0101_0101 B: 1010_1010_1010_1010_1010_1010_1010_1010 Cin: 0 Result: 1111_1111_1111_1111_1111_1111_1111_1111 Cout: 0  Module: CLA[24 ns] A: 0000_0000_0000_0000_0000_0011_0000_0000 B: 1111_1111_1111_1111_1111_1111_1111_1111 Cin: 0 Result: 0000_0000_0000_0000_0000_0010_1111_1111 Cout: 1 </pre>
CSA	CBA
 <pre> Module: CSA[8 ns] A: 0000_0000_0000_0000_0000_0000_0000_0000 B: 1111_1111_1111_1111_1111_1111_1111_1111 Cin: 1 Result: 0000_0000_0000_0000_0000_0000_0000_0000 Cout: 1  Module: CSA[16 ns] A: 0101_0101_0101_0101_0101_0101_0101_0101 B: 1010_1010_1010_1010_1010_1010_1010_1010 Cin: 0 Result: 1111_1111_1111_1111_1111_1111_1111_1111 Cout: 0  Module: CSA[24 ns] A: 0000_0000_0000_0000_0000_0011_0000_0000 B: 1111_1111_1111_1111_1111_1111_1111_1111 Cin: 0 Result: 0000_0000_0000_0000_0000_0010_1111_1111 Cout: 1 </pre>	 <pre> Module: CBA[14 ns] A: 0000_0000_0000_0000_0000_0000_0000_0000 B: 1111_1111_1111_1111_1111_1111_1111_1111 Cin: 1 Result: 0000_0000_0000_0000_0000_0000_0000_0000 Cout: 1  Module: CBA[28 ns] A: 0101_0101_0101_0101_0101_0101_0101_0101 B: 1010_1010_1010_1010_1010_1010_1010_1010 Cin: 0 Result: 1111_1111_1111_1111_1111_1111_1111_1111 Cout: 0  Module: CBA[42 ns] A: 0000_0000_0000_0000_0000_0011_0000_0000 B: 1111_1111_1111_1111_1111_1111_1111_1111 Cin: 0 Result: 0000_0000_0000_0000_0000_0010_1111_1111 Cout: 1 </pre>

[그림7. 실행결과]

4종류의 adder를 실행한 결과 모두 같은 결과가 출력되었지만, delay가 다른 것을 확인할 수 있다. 앞에서 직접 구한 adder delay보다 2만큼 큰 값이 출력되었는데, 각각 source와 sink에서 입력과 출력을 준비하는 과정으로 예상할 수 있다.

## □ Conclusion 결론

여러 종류의 adder에 대하여 gate level에서 delay를 분석한 뒤, systemC를 이용하여 상위수준에서 모델링 하였다. verilog를 이용하여 gate level로 adder를 설계한 경험과 비교했을 때, 빠르고 간단하게 원하는 동작을 구현할 수 있었지만, 동시에 low level에서의 동작을 고려하여 delay를 직접 계산하는 과정이 필요했다. gate level에서 adder를 구현하는 과정에서의 관점과 전체적인 시스템을 구현하는 과정에서의 관점 차이와 유사성이 특징적이었다. 이러한 관점의 차이는 특히 clock에 관하여 나타났는데, gate level에서 adder를 구현할 때는 logic gate를 연결할 때 마다 clock이 지나는 것을 명확하게 알 수 있지만, 이번 과제에서는 wait()을 이용하여 delay를 명시하였다. 따로 명시하지 않으면 시뮬레이션이 어떤 순간에 모든 동작을 완료하는 전혀 다른 시스템을 의미한다는 점이 흥미로웠다. 과제를 진행하며 지금까지 hw/sw를 바라보던 이원적인 관점과 다른, 새로운 level에서 시스템을 보는 기회를 가질 수 있었다.

---

<sup>1</sup> FA, 이준환교수님, 하드웨어 소프트웨어 통합설계 Term Project 제안서, 광운대학교 컴퓨터정보공학부, 2021

<sup>2</sup> RCA, By en:User:Cburnett - Own work This W3C-unspecified vector image was created with Inkscape ., CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=1477488>

<sup>3</sup> CLA, By en:User:Cburnett - Own work This W3C-unspecified vector image was created with Inkscape ., CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=1477525>

<sup>4</sup> CLB, 이준환교수님, 하드웨어 소프트웨어 통합설계 Term Project 제안서, 광운대학교 컴퓨터정보공학부, 2021

<sup>5</sup> CSA, 이준환교수님, 하드웨어 소프트웨어 통합설계 강의자료, 광운대학교 컴퓨터정보공학부, 2021

<sup>6</sup> CBA, By Tibor89 - Own work, CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=31644986>

<sup>7</sup> CBA, By Tibor89 - Own work, CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=31748842>