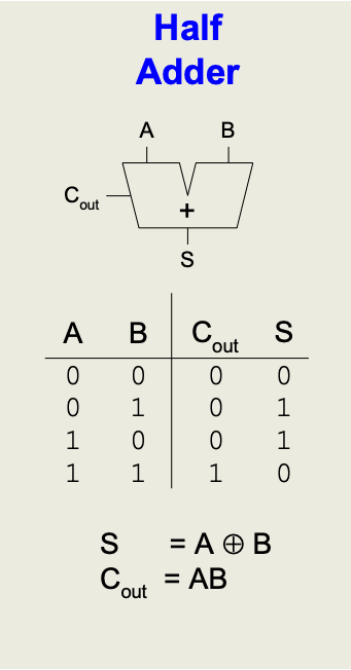
# 개요

Digital building blocks인 Adder 대해 알아보고 bit 변화에 따른 Adder 베릴로그 로직과 이에 대한 구현을 실습한다. 먼저 1-bit Full-Adder를 구현하고 이를 이용하여 N-bit Full-Adder를 Ripple Carry 방식으로 구현한 다음 시뮬레이션 결과를 분석한다.

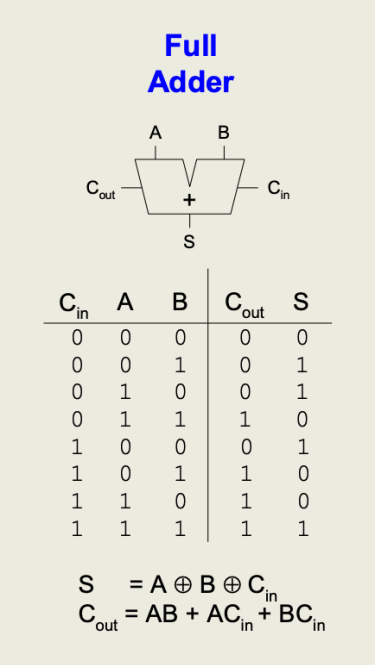
# 이론적 배경

## Half Adder




한 자릿수 이진수를 연산하고, 자리 올림수는 carry로 출력한다. XOR, AND 게이트로 회로 구성이 가능하다.

# Full Adder




| $C_{in}$ | A | B | $C_{out}$ | S |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

$S = A \oplus B \oplus C_{in}$
$C_{out} = AB + AC_{in} + BC_{in}$

하위의 자리 올림수 `Cin` 입력을 포함하여 한 자릿수 이진수를 연산한다. `XOR`, `AND`, `OR` 게이트로 회로 구성이 가능하다.

# Multi-Bit Adders(CPAs)

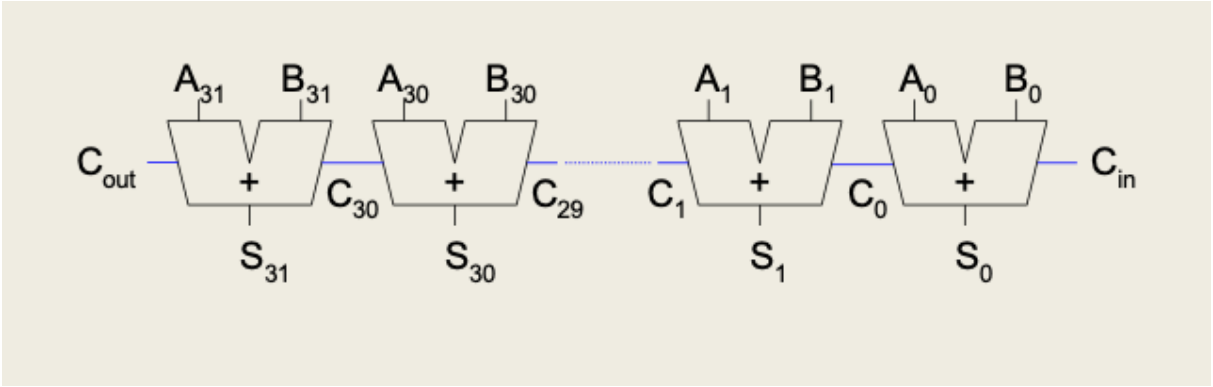
CPA는 Carry Propagate Adder의 약자로서 `Cout`을 그다음 1-bit 또는 하위 비트 연산 서브 모듈의 `Cin`으로 주입하면서 최종 `sum`과 `Cout`를 구하는 연산기를 말한다. CPAs의 종류로는 Ripple-carry, Carry-lookahead, Prefix가 존재한다. Carry-lookahead, Prefix는 빠르지만 추가의 hardware를 요구한다. Ripple-carry는 1-bit 연산기를 일렬로 연결하여 이를 서브 모듈처럼 취급하는 것을 말한다. 각 연산기의 carry를 전체 chain를 통해 연쇄적으로 연결한다. 연산의 속도가 느리지만 매우 간단하며 별도의 hardware를 요구하지 않는다.

# 1-bit Full-Adder

`always@(A or B or Cin)` 구문과 `^, &, |` 연산을 이용하여 S와 Cout를 계산하였다. 결과를 확인해 보기 위해 tb_fa.v를 코딩하였다. 베릴로그 과제 2와 유사하게 `#n always` 구문을 이용하여 시간에 따른 input 값을 조절하고 이에 대한 결과를 분석하였다. waveform을 이용하여 시뮬레이션 해본 결과 진리표와 동일한 결과를 확인할 수 있었다.

# 32-bit Ripple Carry Adder

1. 접근 방법




서브 모듈 1-bit Full-Adder 32개를 연결하여 각 서브 모듈의 Cout을 그다음 서브 모듈의 Cin으로 주입한다. input으로는 각 32bit로 구성된 A, B와 1bit인 Cin을 주입한다. output은 32bit로 구성된 S와 최종 연산의 carry인 Cout으로 연결한다. 여기서 중요한 점은 각 서브 모듈의 carry를 wire 키워드를 사용하여 그다음 서브 모듈의 Cin으로 연결해 주는 것이다. 이를 통해 모듈에 주입되는 Cin의 propagate 현상을 유도할 수 있다.

2. rca.v 구현

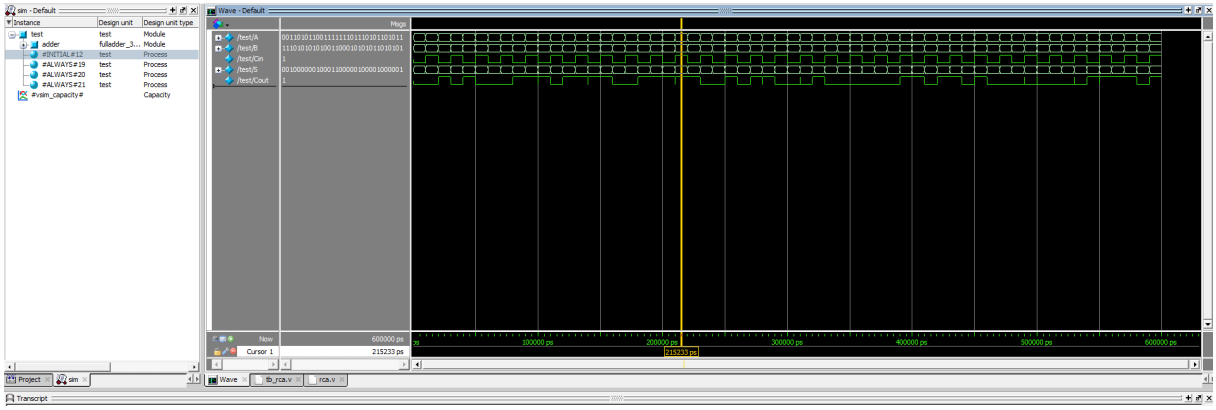
32bit 구성된 각 input과 32bit로 구성된 output를 정의하고 1bit Cin과 Cout을 정의한다. 1-bit 연산 서브 모듈은 위에서 미리 코딩한 1-bit Full-Adder를 이용하였다. 즉 1bit A, B, Cin

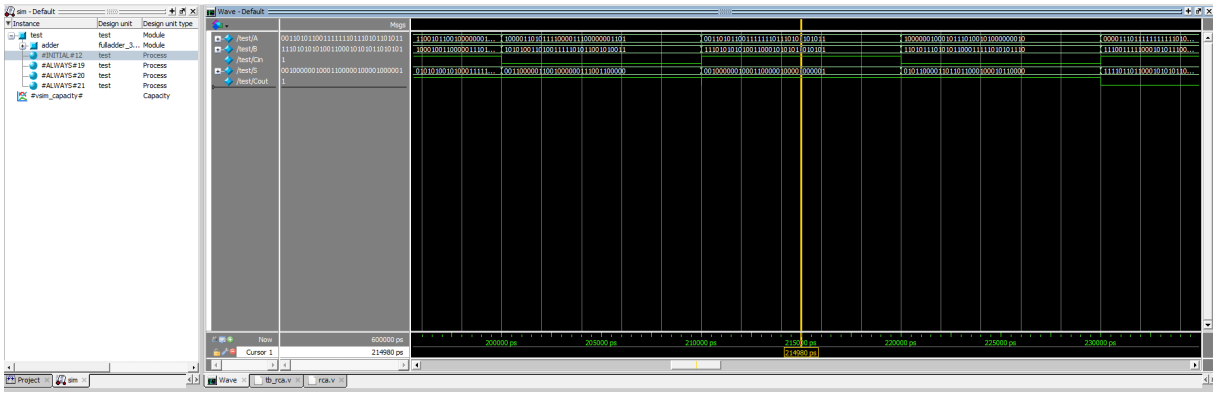
input과 S, Cout output으로 구성된다. 모듈에서 1-bit 연산 서브 모듈의 인스턴스를 32개 생성하고 첫 번째 서브 모듈의 인풋으로 Cin를 주입한다. 각 서브 모듈의 carry를 wire를 통해 그다음 서브 모듈의 Cin으로 넣어주면서 연산을 진행하고 서브 모듈의 S를 전체 모듈의 S[i]와 연결한다. 마지막 서브 모듈의 Cout는 전체 모듈의 Cout과 연결되며 결과적으로 32bit 연산 결과와 1bit carry을 얻을 수 있게 된다.

3. tb_rca.v 구현

초기 데이터 값을 전부 0으로 초기화한다. 이후 always #n 구문과 random() 키워드를 사용하여 난수를 일정 간격으로 생성한 후 이를 모듈에 주입한다. Cin의 경우 ~Cin 키워드를 이용하여 같은 시간 간격으로 변경해 준다.

4. waveform 분석





10ns 간격으로 input 값이 난수로 생성되는 것을 확인할 수 있었다. 각 난수의 조합에 따른 bit 연산을 직접 해본 결과 시뮬레이션 값과 정확히 일치했다. 경우의 수에 따라 Cout이 0 또는 1이 되는 것을 관찰할 수 있었고 각 서브 모듈의 carry가 알맞게 그다음 서브 모듈의 Cin으로 propagate 되는 것을 직접 확인할 수 있었다.

# 결론 및 고찰

N-bit 더하기 연산도 결국 우리가 초등학교 때부터 배워온 10진수 연산과 동일하다. 각 자릿수의 더하기 연산 후 그 carry를 그다음 자릿수의 Cin으로 넘겨주는 것은 진수와 상관없이 항상 동일하다. 이번 중간 과제의 핵심은 1-bit Full-Adder를 먼저 구현하고 이를 이용하여 N-bit Adder를 설계하는 것이다. 1-bit Full-Adder의 Cout은 그다음 자릿수, 즉 그다음 bit 연산의 Cin으로 주입되고 이를 N 번 반복하는 것이다. 이를 연속하여 수행하면 각 서브 모듈에서의 S와 마지막 서브 모듈의 Cout를 취합하여 최종 N-bit 연산의 결과를 확인할 수 있는 것이다.

이번 중간 과제 실습을 통해 직접 N-bit Full-Adder를 설계해 보았다. 아이디어를 고안해 내고 이를 베릴로그 문법을 통해 직접 코딩하는 것이 처음에는 만만치 않았다. 하지만 수업 시간의 교수님 설명과 함께 수업자료를 고민하면서 공부하는 과정을 통해 이번 과제를 무사히 마무리할 수 있었다. 이번 과제를 통해 Full-Adder의 작동원리, bit 연산의 핵심 과정, 이를 직접 코딩하는 문법 스킬 등에 대해 체득할 수 있었다.