10주차 결과보고서

전공: 컴퓨터공학과 학년: 3학년 학번: 20212022 이름: 이예준

**1.**

스크린샷, 소프트웨어, 멀티미디어 소프트웨어, 컴퓨터이(가) 표시된 사진

자동 생성된 설명

\*4-bit Binary Parallel Adder Simulation 결과

텍스트, 스크린샷, 디스플레이, 소프트웨어이(가) 표시된 사진

자동 생성된 설명텍스트, 소프트웨어, 컴퓨터 아이콘, 스크린샷이(가) 표시된 사진

자동 생성된 설명

\*4-bit Binary Parallel Adder Verilog 코드(Design, Simulation Code)

4-bit Binary Parallel Adder는 1-bit Full Adder 4개 또는 1-bit Full Adder 3개와

1-bit Half Adder 1개를 병렬로 연결하여 구현한 논리연산회로이다.

4-bit의 데이터 2개를 받아서 합연산을 한다.

위의 Verilog 코드를 보게 되면 s 배열이 보이는데 이 s는 Sum 공식이며, Sum은 1-bit의

데이터 두개와 이전 자릿수의 Carry를 입력값으로 받아 합 연산을 하는 공식이다.

만약 입력 값의 1의 개수가 짝수이면 0이 Output으로 나오게 된다.

왜냐하면 입력 값들이 모두 0이면 합한 결과가 0이 되고,

입력값 중에 1의 개수가 2개이면 더해서 2가 되어 다음 자릿수로 넘어가서

현재 자릿수가 0이 되기 때문에 두 경우 모두 0이 Output으로 나오게 된다.

반대로 입력 값의 1의 개수가 홀수이면 1이 Output으로 나오게 된다.

왜냐하면 입력값 중에 하나가 1이면 합한 결과가 1이 되고,

입력 값의 1의 개수가 3개이면 더해서 3이 되어 2는 다음 자릿수로 넘어가고

현재 자릿수는 1이 남기 때문에 두 경우 모두 1이 Output으로 나오게 된다.

또 Verilog 코드를 보게 되면 c0~c4가 보이는데 이 c는 Carry 공식이며, Carry는 1-bit의

데이터 두개와 이전 자릿수의 Carry를 입력값으로 받아 Carry 연산을 하는 공식이다.

만약 입력 값의 1의 개수가 0~1개이면 합한 결과가 모두 현재 자릿수에서

표현할 수 있기 때문에 Output으로 0이 나오게 된다.

입력값 중에 1의 개수가 2~3개이면 합한 결과가 모두 현재 자릿수에서

표현할 수 있는 범위를 넘었기 때문에 결과값에서 2를 빼서 남은 건 현재 자릿수에

남겨두고 2는 다음 자릿수에서 표현하기 때문에 Output으로 1이 나오게 된다.

위의 Sum과 Carry에 대한 조건을 공식을 나타내면

이 된다.

예를 들어 입력값으로 x를 1001(9)으로, y를 0011(3)으로 주고 b0를 0으로 줬을 때,

x[0]~x[3]은 각각 1,0,0,1이 주어지고, y[0]~y[3]은 각각 1,1,0,0이 주어진다.

첫번째 자릿수:

s[0]는 x[0]^y[0]^**c0** = 1^1^0 = 0 이 되고,

**c1**은 (x[0]&y[0])|(x[0]^y[0])&**c0** = (1&1)|(1^1)&0 = 1이 된다.

두번째 자릿수:

s[1]는 x[1]^y[1]^**c1** = 0^1^1 = 0 이 되고,

**c2**는 (x[1]&y[1])|(x[1]^y[1])&**c1** = (1&1)|(1^1)&1 = 1이 된다.

세번째 자릿수:

s[2]는 x[2]^y[2]^**c2** = 0^0^1 = 1 이 되고,

**c3**은 (x[2]&y[2])|(x[2]^y[2])&**c2** = (0&0)|(0^0)&1 = 0이 된다.

네번째 자릿수:

s[3]는 x[3]^y[3]^**c3** = 1^0^0 = 1 이 되고,

c4는 (x[3]&y[3])|(x[3]^y[3])&**c3** = (1&0)|(1^0)&0 = 0이 된다.

따라서 s[0]~s[3]까지 보게 되면 합연산의 결과는 1100(12)이 되어 정확한 값이 나왔고,

마지막 c4가 0이 되면서 Overflow가 발생하지 않았다는 것도 알 수 있었다.

**2.**

스크린샷, 소프트웨어, 멀티미디어 소프트웨어, 컴퓨터이(가) 표시된 사진

자동 생성된 설명

\*4-bit Binary Parallel Subtractor Simulation 결과

텍스트, 스크린샷, 디스플레이, 소프트웨어이(가) 표시된 사진

자동 생성된 설명텍스트, 스크린샷, 디스플레이, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

\*4-bit Binary Parallel Subtractor Verilog 코드(Design, Simulation Code)

4-bit Binary Parallel Subtractor 는 1-bit Full Subtractor 4개 또는 1-bit Full Subtractor 3개와

1-bit Half Subtractor 1개를 병렬로 연결하여 구현한 논리연산회로이다.

4-bit의 데이터 2개를 받아서 차연산을 한다.

위의 Verilog 코드를 보게 되면 s 배열이 보이는데 이 s는 Difference 공식이며,

Difference 는 1-bit의 데이터 두개와 이전 자릿수의 Borrow를 입력값으로 받아 차 연산을

하는 공식이다. 또한 b0~b4가 보이는데 이 c는 Borrow 공식이며, Borrow는 1-bit의

데이터 두개와 이전 자릿수의 Borrow를 입력값으로 받아 Borrow 연산을 하는 공식이다.

1. 데이터 값이 모두 0일 때

Borrow가 0이면 0에서 0을 빼서 차는 0이 나오고

Borrow도 없기 때문에 Difference와 Borrow의 Output은 모두 0이 나오게 된다.

Borrow가 1이면 0에서 0을 빼서 차는 0이 되지만 Borrow가 1, 즉 현재 자릿수의 1을

이전 자릿수에서 빌려갔기 때문에 없는 1을 이전 자릿수에게 빌려줄 수 없기에

다음 자릿수에서 Borrow하여 2를 가져오기 때문에Difference와 Borrow의 Output은

모두 1로 나오게 된다.

2. 앞에 있는 데이터 값이 0이고, 뒤에 있는 데이터 값이 1일 때

Borrow가 0이면 0에서 1을 빼게 되어 -1이 될 수 없기에 다음 자릿수에서 2를 빌려와

Difference와 Borrow의 Output 모두 1이 나오게 된다. Borrow가 1이면 차 연산을 하고

다음 자릿수에서 2를 빌려와서 계산해도 이전 자릿수에서 1을 빌려갔기 때문에

Difference의 Output은 0이 나오고 Borrow의 Output 은 1이 나오게 된다.

3. 앞에 있는 데이터 값이 1이고, 뒤에 있는 데이터 값이 0일 때

Borrow가 0이면 1에서 0을 빼서 1이 나오고 이전 자릿수에서 빌려가지 않았기 때문에

Difference의 Output은 1이 나오고 Borrow의 Output 은 0이 나오게 된다.

Borrow가 1이면 차 연산을 해서 1이 나와도 이전 자릿수에서 1을 빌려갔기 때문에 Difference와 Borrow의 Output 모두 0이 나오게 된다.

4. 데이터 값이 모두 1일 때

Borrow가 0이면 1에서 1을 빼서 0이 나오고 이전 자릿수에서 빌려가지 않았기 때문에

Difference와 Borrow의 Output 모두 0이 나오게 된다.

Borrow가 1이면 차 연산을 해서 0이 나오고 이전 자릿수에서 1을 빌려갔기 때문에

0에서 1을 빌려줄 수 없기 때문에 다음 자릿수에서 2를 빌려와 계산해서

Difference와 Borrow의 Output 모두 1이 나오게 된다.

위의 Difference과 Borrow에 대한 조건을 공식을 나타내면

이 된다.

예를 들어 입력값으로 x를 0101(5)으로, y를 0111(7) 으로 주고 b0를 0으로 줬을 때,

x[0]~x[3]은 각각 1,0,1,0이 주어지고, y[0]~y[3]은 각각 1,1,1,0이 주어진다.

첫번째 자릿수:

s[0]는 x[0]^y[0]^**b0** = 1^1^0 = 0 이 되고,

**b1**은 (~x[0]&y[0])|~(x[0]^y[0])&**b0** = (0&1)|~(1^1)&0 = 0이 된다.

두번째 자릿수:

s[1]는 x[1]^y[1]^**b1** = 0^1^0 = 1 이 되고,

**b2**는 (~x[1]&y[1])|~(x[1]^y[1])&**b1** = (1&1)|~(0^1)&1 = 1이 된다.

세번째 자릿수:

s[2]는 x[2]^y[2]^**b2** = 1^1^1 = 1 이 되고,

**b3**은 (~x[2]&y[2])|~(x[2]^y[2])&**b2** = (0&1)|~(1^1)&1 = 1이 된다.

네번째 자릿수:

s[3]는 x[3]^y[3]^**b3** = 0^0^1= 1 이 되고,

b4는 (~x[3]&y[3])|~(x[3]^y[3])&**b3** = (1&0)|~(0^0)&1 = 1이 된다.

따라서 s[0]~s[3]까지 보게 되면 합연산의 결과는 1110(14)가 되고,

마지막 b4가 1이 되면서 다음 자릿수에서 빌려와 Borrow가 발생했다는 것도 알 수 있었다.

만약 데이터 5-bit이고 x가 10101(21)이었다면 1110(14)가 정확한 값이 되었을 것이다.

**3.**

텍스트, 스크린샷, 디스플레이, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

\*BCD Adder Simulation 결과

텍스트, 스크린샷, 폰트, 문서이(가) 표시된 사진

자동 생성된 설명텍스트, 소프트웨어, 스크린샷, 운영 체제이(가) 표시된 사진

자동 생성된 설명

\* BCD Adder Verilog 코드(Design, Simulation Code)

BCD Adder는 2개의 4-bit의 BCD 코드를 더하는 연산을 하여 결과값을 출력한다.

BCD 코드는 한자릿수의 십진수를 4-bit의 이진수로 표현한 코드이기 때문에

한 자릿수의 십진수는 0~9까지 밖에 표현할 수 없기 때문에BCD Adder의 하나의

입력 값은 1010(10)이상이 되지 않는다.

위의 Verilog 코드를 보게 되면 먼저 4-bit Binary Parallel Adder와 마찬가지로 각 bit의

Sum과 Carry를 계산하여 z배열과 c1~c3,k에 각각 저장한다. 이때 k 는 합연산을 했을 때

마지막에 계산되는 Carry를 의미한다(연산 결과가 16이상인지를 알 수 있다).

w1은 4번째 bit의 Sum결과와 3번째 bit의 Sum 결과를 AND연산하여 연산 결과가

12, 13, 14, 15 중 하나인지 확인하고, w2는 4번째 bit의 Sum결과와 2번째 bit의 Sum

결과를 AND연산하여 10, 11중 하나인지 확인한다. 그 다음 k와 w1과 w2를 OR연산하여

최종적으로 BCD코드를 합 연산했을 때 10을 넘어가서 Carry가 발생하는지 안 하는지를

확인한다. 이때 Carry가 발생했다면 결과값에 보정이 필요하기 때문에 0110(6)과 결과값을

다시 한번 합 연산을 해준다.

예를 들어 입력값으로 a를 0110(6)으로, b를 1000(8) 으로 주고 c0를 0으로 줬을 때,

a[0]~a[3]은 각각 0,1,1,0이 주어지고, b[0]~b[3]은 각각 0,0,0,1이 주어진다.

위에서 4-bit Binary Parallel Adder예시를 설명한 것과 같은 방법으로 z배열 값과 c1~c3,k를

구하게 되면 z[0]~z[3]은 각각 0,1,1,1이 된다. 값이 1110(14)가 되어 10을 넘어갔기 때문에

결과를 보정할 필요가 있다. 먼저 10이 넘어갔다는 것을 판단하기 위한 Carry를 구한다.

w1 = z[3] & z[2] = 1 & 1 = 1

w2 = z[3] & z[1] = 1 & 1 = 1

k = 0

Cout = 0 | 1 | 1 = 1

그 다음 s[0]~s[3]까지 각각 0,Cout,Cout,0을 저장하면 0,1,1,0이 된다.

이 s배열과 z배열을 다시 한번 합 연산하면 결과는 0100(4)이 되면서

최종적으로 Carry가 발생했고, BCD코드는 0100(4)라는 결과를 얻게 된다.

**4.**

이번 실험에서 4-bit Binary Parallel Adder와 4-bit Binary Parallel Subtractor의 작동방식에

대해 알아봤고 각각 4개의 Full Adder와 Full Subtractor로 Verilog코드를 구현하여

Simulation결과와 FPGA 보드 구현 결과가 정상적으로 나오는 것을 확인할 수 있었다.

또한 BCD Adder의 경우도 첫번째 합 연산과 두번째 합 연산을 4개의 Full Adder로

Verilog코드를 구현했고, Cout은 2개의 AND gate와 1개의 OR gate로 Verilog코드를

구현했다. 이 역시 Simulation결과와 FPGA 보드 구현 결과가 정상적으로 나오는 것을

확인할 수 있었다. 이번 실험의 논의사항으로 다른 회로의 구현방식에 대해 생각해 보았다.

이번 실험에서는 각 회로를 Ripple Carry 방식으로 구현했는데, Carry Look Ahead 방식으로도

구현해서 결과가 같은 지 또는 회로의 속도차이가 어느 정도 나는지 확인할 수 있으면

좋을 것 같다.

**5.**

이번 실험에서 사용한 방식은 Ripple Carry 방식으로 이전 자릿수에서 Carry가 계산돼야만

현재 자릿수의 Sum과 Carry를 계산할 수 있어 속도가 느리다는 단점이 있다.

이를 보완하기 위해 나온 방식이 Carry Look Ahead 방식인데 Carry를 예측한다는 의미의

방식으로 처음의 들어오는 Carry를 통해 모든 자릿수의 Sum과 Carry를 예측할 수 있다.

하지만 이 방식도 단점이 있어 bit 수가 늘어나면 늘어날수록 상위 bit의 Logic은

기하급수적으로 복잡 해진다. 따라서 이 단점을 다시 한번 극복하기 위한 방식이

Prefix 방식이며 연산을 단계를 나누어 각 단계에서 병렬 연산한다.

Prefix 방식에는 Kogge-Stone 방식과 Brent–Kung 방식이 있다.

스크린샷, 다채로움, 원이(가) 표시된 사진

자동 생성된 설명스크린샷, 블랙, 흑백, 그래픽이(가) 표시된 사진

자동 생성된 설명

\*Kogge-Stone 방식 \*Brent–Kung 방식