2018년 2학기 컴퓨터공학실험Ⅱ
CSE3016-05반 6주차 예비 보고서

학번: 20171665
이름: 이 선 호

2018. 10. 19

목 차

Ⅰ 전 가산기와 반 가산기

1. 전 가산기

이진수로 된 두 수를 더하는 경우 임의의 자리에 들어갈 값이 무엇인지 알기 위해서는 이전 자리의 두 수를 더하여 합 1비트와 캐리(Carry) 1비트를 구해야 한다. 이처럼 이전 자리에서의 캐리를 자리올림수로 여겨 계산 과정에서 반영하는 가산기가 전 가산기(Full Adder)이다. 따라서 전 가산기는 더해야 하는 두 이진수의 비트와 이전 캐리 값을 받아오기 때문에 입력이 3개 존재한다.

A_i	B_i	이전 캐리 C_i	캐리(C_{i+1})	출력(S_i)
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

위의 표를 살펴보면 A와 B의 값이 서로 다르고 이전 캐리 값이 1일 때 캐리가 1로 출력된다는 것을 알 수 있다. 또한 A와 B가 모두 1일 때는 이전 캐리 값에 상관없이 캐리가 1로 출력된다는 것을 알 수 있다. 그리고 A, B, 그리고 이전 캐리 값들 중에서 1값이 홀수 개가 나오면 출력이 1로 나온다는 사실을 알 수 있다. 이를 종합하여 캐리와 출력을 구하는 논리식을 작성하면 아래와 같다.

$$C_{i+1} = A_iB_i + C_i(A_i \oplus B_i)$$
$$S_i = A_i \oplus B_i \oplus C_i$$

예를 들어, A의 값이 1이고, B의 값이 0, 이전 캐리의 값이 1이라고 가정하자. A와 B의 값이 다르므로 XOR 연산을 통해 값이 1이 나오고 이를 이전 캐리와 AND 연산하

면 1이 나온다. A와 B의 AND 연산 값이 0이어도 이는 현재 계산한 값과 OR 연산이 되므로 최종 출력되는 캐리는 1이다. 이 캐리의 값이 다음 자릿수에서의 연산에도 반영이 된다. 또한 A와 B와 이전 캐리의 값들을 살펴보면 1이 짝수개가 존재하므로 비트 합을 계산한 값은 0이 된다.

2. 반 가산기

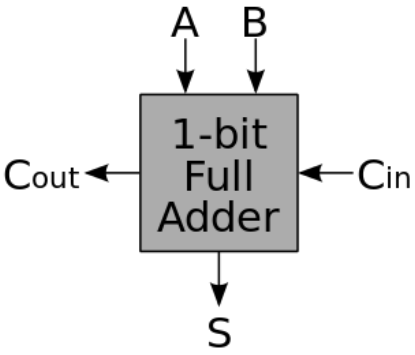
전 가산기와는 다르게 반 가산기에서는 이진수로 된 두 수를 더하는 경우 이전 캐리의 값을 연산 과정에서 반영하지 않는다. 따라서 3개의 입력을 요구하는 전 가산기와는 다르게 반 가산기는 입력을 2개만 필요로 한다. 단, 전 가산기와 같이 입력 값들로 계산한 캐리 값과 합 출력 값을 계산한다.

A_i	B_i	캐리(C_{i+1})	출력(S_i)
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

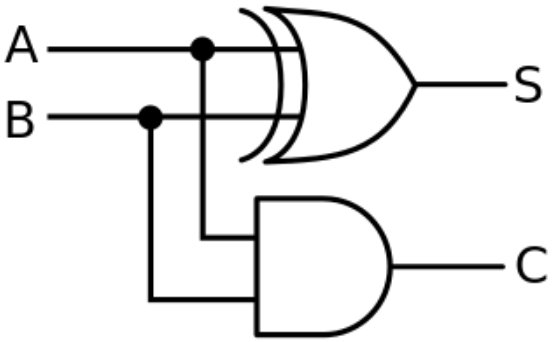
위의 표를 살펴보면 A와 B의 값이 모두 1일 때 현재 캐리 값이 1로 출력된다는 것을 알 수 있다. 또한 A와 B의 값이 서로 다르면 출력 값이 1로 출력된다는 사실을 알 수 있다. 이를 종합하여 캐리와 출력을 구하는 논리식을 작성하면 아래와 같다.

$$C_{i+1} = A_i B_i$$
$$S_i = A_i \oplus B_i$$

예를 들어, A의 값이 1이고, B의 값이 0이라고 가정하자. A와 B의 값이 다르므로 XOR 연산을 통해 값이 1이 나온다. 따라서 최종 합 비트 값은 1로 출력된다. 또한 A와 B의 AND 연산 값이 0이므로 출력되는 캐리의 값은 0이 된다.



[전 가산기]



[반 가산기]

Ⅱ 전 감산기와 반 감산기

1. 전 감산기

일반적으로 이진수의 뺄셈은 빼고자 하는 수를 2의 보수 표현을 사용하여 덧셈으로 변환하여 수행할 수 있다. 예를 들어, 뺄셈 $A-B$는 $A+B'+1$와 같이 B에 대한 2의 보수를 취하여 A에 더하여 계산이 가능하다. 따라서 실제 회로에서는 효율을 높이기 위해 주로 전 감산기를 별도로 설계하지 않고 전 가산기를 전 감산기로 사용한다. 이를 가감산기라고 한다. 그러나 별도로 뺄셈만 가능한 전 감산기도 구현은 가능하다. 전 감산기는 반 감산기와는 다르게 추가로 아랫자리에서 요구하는 발림수(빌려준 수)에 의한 뺄셈까지 수행한다.

A_i	B_i	빌려준 수(B_{i-1})	발림수(B_i)	출력(S_i)
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

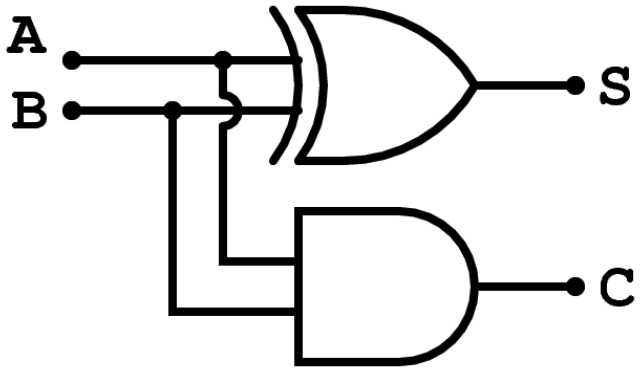
예를 들어, A가 0, B가 1, 그리고 이전 자리에 빌려준 수가 1이라고 가정하자. 그러면 A에서 B와 빌려준 수를 빼면 음수가 출력된다. 그러면 앞의 자리에서 발림수를 가져와야 한다. 따라서 발림수는 1로 출력이 되고, 발림수를 고려하여 뺄셈을 계산하면 출력 값이 0으로 나온다는 것을 알 수 있다.

2. 반 감산기

현재 자리에서 이전 자리에게 수를 빌려주는 감수를 고려하는 전 감산기와는 다르게 반 감산기에서는 감수를 고려하지 않고 이진수로 된 두 수의 차와 발림수를 출력한다. 입력된 두 수에서 같은 자리의 비트 뺄셈은 차로 출력이 되고, 만일 이 차가 음수를 갖게 되는 경우 앞의 자리에서 수를 빌려와 계산하게 되므로 발림수가 1로 출력된다.

A_i	B_i	발림수(B_i)	출력(S_i)
0	0	0	0
0	1	1	1
1	0	0	1
1	1	0	0

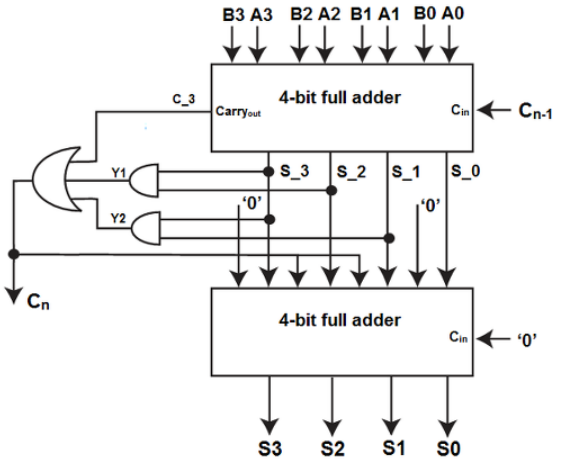
예를 들어, A가 0, B가 1이라고 하자. 이 두 수의 차를 계산하면 음수가 나온다. 따라서 앞의 자리에서 수를 빌려와서 계산을 해야 하므로 발림수가 1로 출력이 된다. 그리고 차를 계산한 결과 값은 1이 나온다.



반 감산기는 카르노 맵으로 최소항을 구하여 논리 회로를 그렸을 때 위와 같은 회로도를 갖으며, 전 감산기는 반 감산기의 회로를 2개 필요로 한다.

Ⅲ BCD 가산기

1. BCD 가산기



앞에서는 이진수로 된 입력 값을 서로 더하거나 빼는 연산을 하는 논리 회로를 확인하였다. 그러나 우리가 실제로 많이 사용하는 십진수로 된 연산 회로는 보지 못하였다. BCD는 십진수로 된 두 수를 더하는 연산을 작업을 수행하는 역할을 한다.

BCD 코드는 이진화 십진법을 사용한 코드를 의미하며 네 자리를 묶어 십진수 한 자리로 사용하는 기수법이다. 각 자리마다 0과 1을 사용하고 이를 네 개 사용하여 십진수를 만들므로 BCD 코드는 2의 네제곱인 16개의 수를 표현할 수 있고, 0부터 15까지 표현이 가능하다.

0과 9까지의 수로 표현된 BCD 코드의 연산에서는 이진수의 경우와 다르지 않다. 그러나 실제로 십진수에서 한 자리에서는 0부터 9까지의 숫자만 사용하므로 BCD 코드에서 쓰이지 않는 10부터 15까지의 6개의 수가 발생한다. 그래서 BCD 가산기에서는 10이상인 수는 앞의 자리에 캐리를 보내게 되어 있다. 만일 덧셈 결과가 1001(=9) 이하에 해당하는 경우는 이진수에서의 연산과 같다. 그러나 만일 덧셈 결과가 1010(=10)이상에 해당하는 경우는 이진 결과에 0110(=6)을 더해주고 발생한 올림수는 앞의 자릿수에 해당하는 BCD의 최하위 비트에 캐리로 올려준다. 즉, 덧셈 결과가 1010(=10)이상에 해당하면 0110(=6)을, 그렇지 않은 경우에는 0000(=0)을 더해주는 회로를 구성한 것이 바로 BCD 가산기이다.

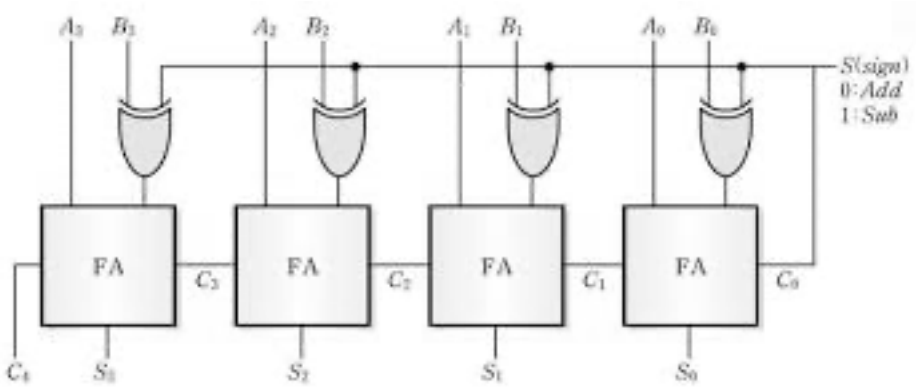
Ⅳ 병렬 가감산기

1. 병렬 가감산기

Ⅱ-1에서 언급한 바와 같이 일반적으로 이진수의 뺄셈은 빼고자 하는 수를 2의 보수 표현으로 바꾸어서 덧셈으로 연산하기 때문에 가산기에 일부 논리 연산을 더하여 뺄셈 역할을 수행하는 회로를 제작하는 것이 가능하다. 이를 병렬 가감산기라고 한다. 앞에서 말한 바와 같이 뺄셈 $A-B$는 $A+B'+1$와 같이 B에 대한 2의 보수를 취하여 A에 더하여 계산이 가능하다. 따라서 덧셈일 때는 $A+B$를 계산하고 뺄셈일 때는 $A+B'+1$을 계산하도록 하는 기능을 넣어야 한다.

이를 해결할 수 있는 방법은 덧셈을 수행할지 또는 뺄셈을 수행할지에 따라 입력 값 신호를 다르게 하는 입력을 하나 더 추가하는 것이다. 덧셈일 때는 0을 신호로 넣으면 이를 더하는 수의 각 비트마다 XOR 연산을 수행하여 가산기로 입력이 들어가게끔 하고, 반대로 뺄셈일 때는 1을 신호로 넣어 이를 빼고자 하는 수의 각 비트마다 XOR 연산을 수행하여 가산기로 입력이 들어가고, 추가로 넣었던 신호도 같이 가산기의 입력으로 넣어서 마치 캐리처럼 계산 결과에 반영하게 하는 것이다. 이런 방법이 가능한 이유는 어떤 변수를 1과 XOR 연산하면 그 변수의 보수가 결과 값으로 출력되고, 0을 XOR 연산하면 그 변수가 그대로 출력되기 때문이다. 이를 정리한 식은 아래와 같다.

$$1 \otimes x = x' \quad \text{그리고} \quad 0 \otimes x = x$$

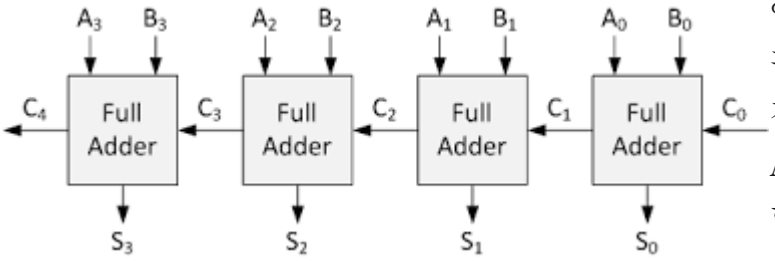


V Bits Adder

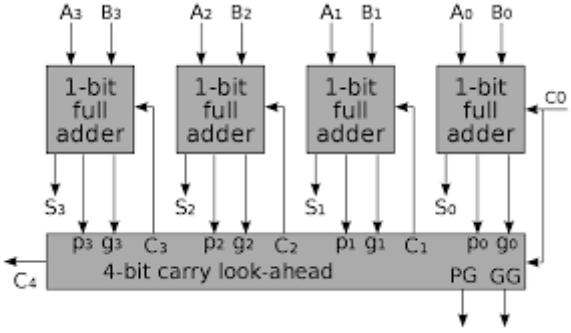
1. Ripple Carry Adder

Carry Look-Ahead Adder를 설명하기 전에 Ripple Carry Adder를 먼저 설명하고자 한다. Ripple Carry Adder는 앞에서 설명한 1비트 전 가산기를 계산하고자 하는 이진

수의 n비트 수만큼 직렬 연결하여 만든 회로이다. 전 가산기에서 이진수를 각 자리마다 덧셈 연산을 할 때는 자리올림수인 캐리가 발생할 수 있기 때문에 이를 다음 자리의 계산에 입력으로 반영해줘야 한다. 따라서 앞의 1비트 전 가산기에서 계산한 캐리를 다음 자리 비트 계산을 수행하는 전 가산기에 입력으로 전달하도록 회로를 구현해야 한다. 이처럼 1비트 가산기를 필요한 비트 수만큼 직렬적으로 연결하여 덧셈 계산을 수행하는 Adder를 Ripple Carry Adder라고 한다.



2. Carry Look-Ahead Adder



Ripple Carry Adder는 앞자리 비트의 연산을 수행하는 가산기의 계산이 다 끝나고 나서 carry가 결정되어야 다음 자리의 비트 연산을 수행할 수 있어서 연산자의 비트 수가 많아질수록 delay 시간이 오래 걸린다는 단점을 지니고 있다. 이러한 delay 문제를 해결하고자 구현한 Adder가 바로 Carry Look-Ahead Adder이다. Carry Look-Ahead Adder는 말 그대로 캐리를 먼저 빠르게 훑어보겠다는 것이며, 각 비트 연산의 캐리를 빠르게 알기 위해서는 앞에서 정리했던 식을 참고할 필요가 있다.

$$C_{i+1} = A_i B_i + C_i(A_i \oplus B_i)$$

위 식은 이전 비트 자리 연산에서의 캐리를 받아 현재 캐리를 구하는 것이다. 그러나 이를 아래 식처럼 나타내면 굳이 이전 캐리를 참조하지 않고도 빠르게 현재 캐리를 계산 가능하다.

$$g_i = A_i B_i \quad \text{그리고} \quad p_i = A_i \oplus B_i$$

$$C_1 = g_0 + p_0 C_0$$

$$C_2 = g_1 + p_1 C_1 = g_1 + p_1 g_0 + p_1 p_0 C_0$$

$$C_{i+1} = g_i + p_i g_{i-1} + p_i p_{i-1} g_{i-2} + \cdots + p_i p_{i-1} \cdots p_0 C_0$$

처음 입력 받을 때의 캐리 값과 각 자리에서의 두 비트의 AND 연산 값과 XOR 연산 값만을 알면 굳이 이전 자리에서의 캐리를 미리 계산하지 않고도 빠르게 현재 자리에서의 캐리 연산이 가능하다. 그래서 delay에 있어서 Ripple Carry Adder보다 훨씬 절약이 된다. 다만, 자릿수가 더 커질수록 bottleneck이 발생하기 때문에 대개 4비트 또는 8비트 단위로 끊어서 이를 결합하여 큰 비트 자리의 가산기를 만들기도 한다.

Ⅵ 기타 이론

1. Carry Save Adder

자리올림수 저장 가산기(Carry Save Adder)는 이진법에서 3개 또는 그 이상의 비트로 된 이진수 덧셈을 계산하기 위한 가산기의 한 종류이다. 이 가산기는 두 개의 수를 출력하는데, 하나는 부분적인 덧셈 비트의 반복이고, 다른 하나는 지라올림수(Shift carry) 비트의 반복이다. 3개의 입력값을 계산하기에 먼저 두 수의 각각의 비트 계산은 전 가산기를 통해 계산하도록 하고, 캐리가 존재할 경우 그 다음 자리 수에서 계산을 반복할 수 있도록 Ripple Carry Adder를 사용한다. 이 때 알 수 있는 사실이, 이 가산기가 자리올림수 저장 가산기로 불리는 이유는 보통 Ripple Carry Adder를 사용하게 되면 한 전 가산기에서 계산한 캐리를 다음 자리 계산에 반영한다. 그러나 여기서는 각 자리에서의 계산을 이미 다 하고 나서 Ripple Carry Adder를 실행하기 때문에 캐리를 중간 과정에서 저장한다는 표현을 사용하는 것이다. 이 자체로는 크게 성능 향상이 된다고 할 수는 없지만, 입력이 여러 개일 때 Carry Save Adder를 통해 병렬성을 이용하여 계산 성능을 크게 향상시킬 수 있다.