2018년 2학기 컴퓨터공학실험Ⅱ
CSE3016-05반 10주차 예비 보고서

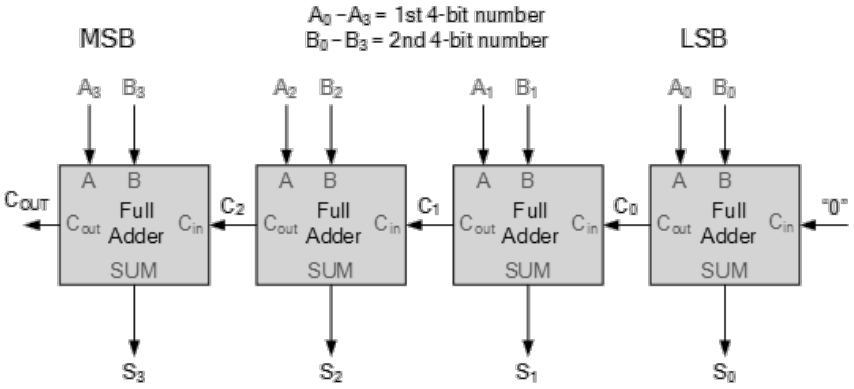
학번: 20171665
이름: 이 선 호

2018. 11. 23

목 차

Ⅰ 4-Bit Adder와 Substractor

1. 4-Bit Adder 이진 병렬 연산



4-Bit 병렬 가산기(이진 병렬 가산 회로)는 두 개의 4비트 피연산자를 가장 오른쪽 자리 비트부터 차례로 더해가면서 발생하는 캐리를 반영하여 다음 자리 비트 연산에 반영하는 논리 회로이다. 즉, 지난 6주차 예비 보고서에서 조사한 4개의 전 가산기(Full Adder)를 병렬로 연결한 것이다. 각 전 가산기에서는 같은 자리의 두 개의 비트를 가지고 이진 덧셈 연산을 수행해야 하는데, 이 때 이전 자리의 두 수를 더하여 합 1비트와 캐리(Carry) 1비트를 구해야 한다. 따라서 전 가산기는 더해야 하는 두 이진수의 비트와 이전 캐리 값을 받아오기 때문에 위의 그림과 같이 A_i, B_i, C_i라는 3개의 입력이 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.

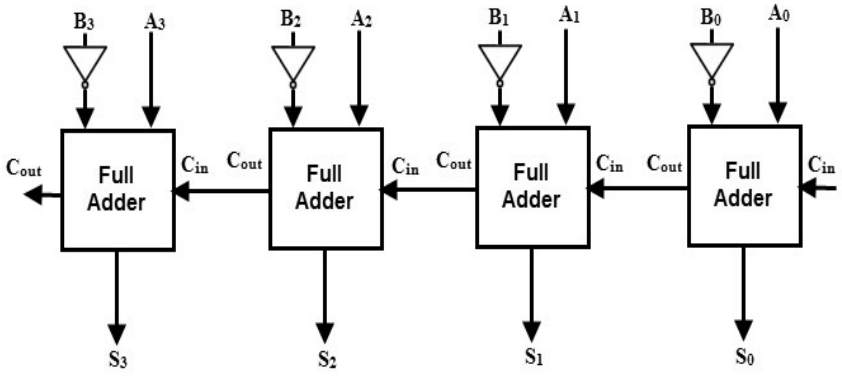
$$C_{i+1} = A_iB_i + C_i(A_i \oplus B_i)$$
$$S_i = A_i \oplus B_i \oplus C_i$$

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 연산하면 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.

4-Bit Adder에서 각 output의 boolean 식을 정리하면 아래와 같다.

$$C_0 = A_0B_0 + 0 \bullet (A_0 \oplus B_0),\ S_0 = A_0 \oplus B_0 \oplus C_0$$
$$C_1 = A_1B_1 + C_0(A_1 \oplus B_1),\ S_1 = A_1 \oplus B_1 \oplus C_1$$
$$C_2 = A_2B_2 + C_1(A_2 \oplus B_2),\ S_2 = A_2 \oplus B_2 \oplus C_2$$
$$C_{out} = A_3B_3 + C_2(A_3 \oplus B_3),\ S_3 = A_3 \oplus B_3 \oplus C_3$$

2. 4-Bit Subtractor 이진 병렬 연산



4-Bit 병렬 감산기(이진 병렬 감산 회로)는 두 개의 4비트 피연산자를 가장 오른쪽 자리 비트부터 차례로 빼가면서 발생하는 빌림수를 반영하여 다음 자리 비트 연산에 반영하는 논리 회로이다. 즉, 지난 6주차 예비 보고서에서 조사한 4개의 전 감산기(Full Subtractor)를 병렬로 연결한 것이다. 각 전 감산기에서는 같은 자리의 두 개의 비트를 가지고 이진 뺄셈 연산을 수행해야 하는데, 이 때 이

전 자리의 두 수를 뺀 값의 1비트와 앞의 비트 자리로부터 빌려오는 1비트를 구해야 한다. 따라서 전 감산기는 더해야 하는 두 이진수의 비트와 이전 캐리 값을 받아오기 때문에 빌림수를 C_i라고 하면 A_i, B_i, C_i라는 3개의 입력이 존재한다.

A_i	B_i	빌려준 수(C_{i-1})	발림수(C_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

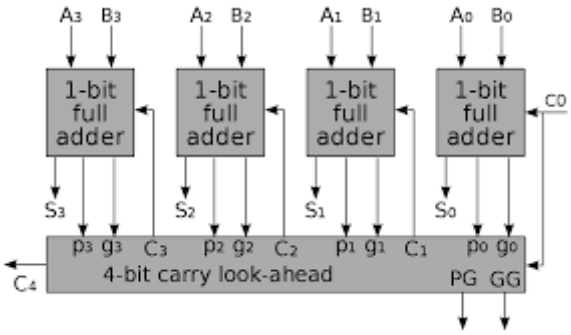
예를 들어, A가 0, B가 1, 그리고 이전 자리에 빌려준 수가 1이라고 가정하자. 그러면 A에서 B와 빌려준 수를 빼면 음수가 출력된다. 그러면 앞의 자리에서 발림수를 가져와야 한다. 따라서 발림수는 1로 출력이 되고, 발림수를 고려하여 뺄셈을 계산하면 출력 값이 0으로 나온다는 것을 알 수 있다. 위의 진리표를 기반으로 boolean expression을 작성하면 다음과 같다.

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

여기서 알 수 있는 사실은 위의 4-bit 이진 병렬 가산기에서 입력 B를 보수 처리하여 입력하기만 해도 4-bit 이진 병렬 감산기와 같아진다는 것이다. 병렬 가산기의 캐리 구하는 식에서 B만 보수 처리로 바꿔주면 위의 식과 같아지기 때문이다.

Ⅱ Look Ahead Carry

1. Look Ahead Carry



4-Bit 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, Carry Look-Ahead에 의해서 나온 캐리를 Look Ahead Carry라고 한다.

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

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.

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

$$C_1 = g_0 + p_0 C_0$$

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

$$C_{i+1} = g_i + p_i g_{i-1} + p_i p_{i-1} g_{i-2} + \cdots + p_i p_{i-1} \cdots p_0 C_0 \quad \textit{(Look Ahead Carry)}$$

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 4-bit Adder에서 delay에 있어서 각 자리의 전 가산기마다 일일이 캐리를 계산하는 것보다 훨씬 절약이 된다. 다만, 자릿수가 더 커질수록 bottleneck이 발생하기 때문에 대개 4비트 또는 8비트 단위로 끊어서 이를 결합하여 큰 비트 자리의 가산기를 만들기도 한다.

Ⅲ XOR을 활용한 2의 보수 가감산

1. XOR를 활용한 2의 보수 가감산

위에서 살펴본 것처럼 4-bit Adder에 빼고자 하는 비트 피연산자를 보수 처리하여 입력하면 4-bit Subtractor가 되듯이, 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.

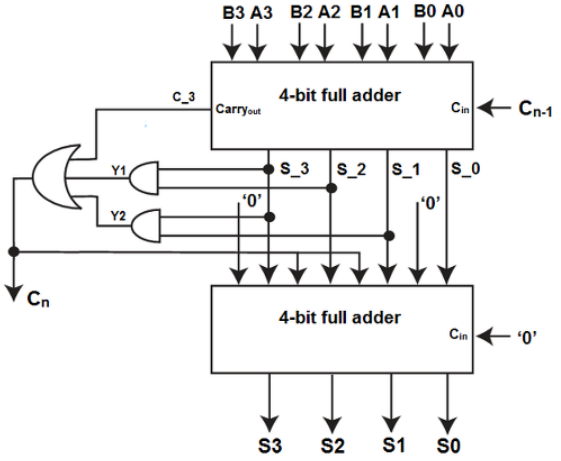
$$1 \oplus x = x' \quad \text{그리고} \quad 0 \oplus x = x$$

Ⅳ BCD 연산

1. BCD 연산

BCD 연산은 십진수로 된 두 수를 더하는 연산을 작업을 수행하는 역할을 한다. BCD 코드는 십진수의 값을 갖는 이진코드를 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 코드의 값에서 쓰이지 않는 10부터 15까지의 6개의 수가 발생한다. 그래서 BCD 가산기에

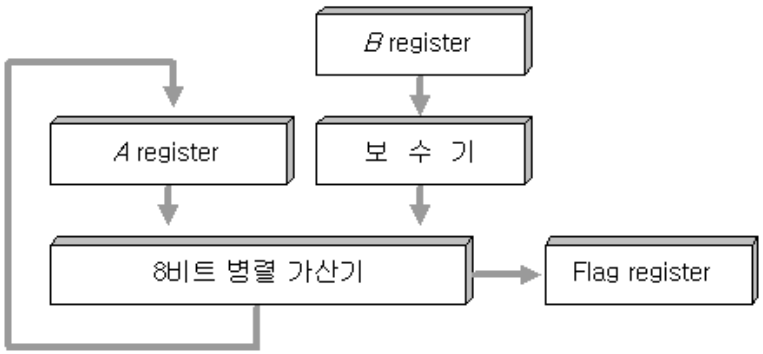


서는 10이상인 수는 앞의 자리에 캐리를 보내게 되어 있다. 만일 덧셈 결과가 1001(=9) 이하에 해당하는 경우는 이진수에서의 연산과 같다. 그러나 만일 덧셈 결과가 1010(=10)이상에 해당하는 경우는 이진 결과에 0110(=6)을 더해주고 발생한 올림수는 앞의 자릿수에 해당하는 BCD의 최하위 비트에 캐리로 올려준다. 즉, 덧셈 결과가 1010(=10)이상에 해당하면 0110(=6)을, 그렇지 않은 경우에는 0000(=0)을 더해주는 회로를 구성한 것이 바로 BCD 가산기이다.

반대로 BCD 감산기에서는 만일 현재 자리의 BCD 코드에서 연산을 수행했을 때 음수가 나오면 앞의 자릿수에 해당하는 BCD에서 빌림수인 0110(=6)을 빌려올 수 있다.

V ALU의 기능

1. ALU의 기능



ALU(산술 논리 연산 장치)는 CPU에서 레지스터와 논리 부분의 작업을 수행하며, 덧셈, 뺄셈, 곱셈, 나눗셈과 같은 산술 연산이나 두 수의 비교와 같은 논리 연산을 담당하는 장치이다. 이러한 산술 연산에는 고정 소수점 연산과 부동 소수점 연산 등이 있다. 구체적으로 고정 소수점 수의 덧셈과 뺄셈 등 산술 연산, AND, OR, XOR 등 비트 논리 연산, rotating 또는 shift 연산과 같은 비트 자리 이동 연산, 이진 데이터를 보수로 취하는 연산, 연산 결과의 상태를 나타내는 플래그 저장 등의 기능을 수행하며, 각 기능은 산술 연산 장치, 논리 연산 장치, 보수기, shift 레지스터, 상태 레지스터 등이 역할을 담당한다.

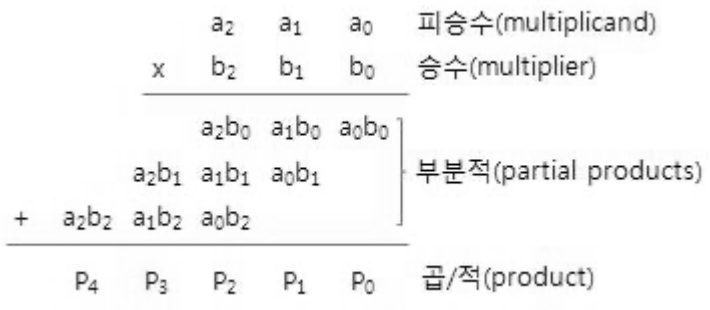
이러한 다양한 연산을 담당하기 위해 ALU는 덧셈을 위한 가산기를 중심으로 연산에

서 사용되는 데이터 또는 연산 결과를 임시적으로 저장하기 위한 레지스터, 보수를 만드는 보수기, overflow를 검출하는 overflow 검출기 등으로 구성되어 있다. 기본적으로 내부는 전 가산기로 구성되어 있으며, 산술 연산 장치를 구현하기 위해 전 가산기 회로를 이용하여 병렬 가산기로 구성된다.

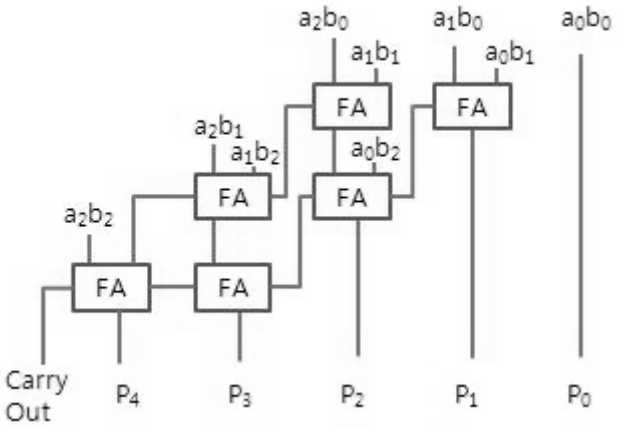
Ⅵ 기타

1. 이진 병렬 승산기

덧셈과 뺄셈을 수행하는 가감산기만 존재하는 것이 아니라 곱셈(또는 나눗셈)을 수행하는 승(제)산기도 존재한다. 곱셈은 덧셈을 반복하여 수행할 수도 있고, shift 등의 논리 연산을 통해서도 수행 가능하다. 이진 병렬 승산기에서는 크게 전자의 방식을 선택하여 곱해야 하는 수(피승수)를 곱하는 수(승수)만큼 누적하여 더하는 경우와, 후자의 방식을 택하여 shift 논리 연산회로를 사용하여 수행하는 경우가 있다. 그러나 주로 피연산자의 비트 자릿수가 작으면 사람이 직접 계산할 때와 비슷하게 직접 피승수와 승수를 각 자리별로 곱하여 더하는 과정을 따른다.



위의 그림은 우리가 일반적으로 곱셈을 수행할 때 그 과정을 도식화하여 나타낸 것이다. 실제로 각 자리의 비트를 차례로 AND 연산하여 연산 결과 자리에 맞는 수들끼리 더하는 과정을 수행하면 각 자리 비트별로 최종 연산 결과가 나오게 된다. 전 가산기를 사용하여 논리 회로를 구성하면 아래와 같다.



각 자리 비트별로 AND 연산한 결과를 전 가산기의 입력으로 넣어서 같은 자리에 해당하는 값들은 따로 더하여 최종 각 자리 비트의 output으로 연결시켰다. 또한 이전 자리 비트에서 발생한 캐리를 반영하기 위해서 다음 자리의 캐리 입력으로 넣어준다.