2018년 2학기 컴퓨터공학실험Ⅱ
CSE3016-05반 7주차 예비 보고서

학번: 20171665
이름: 이 선 호

2018. 11. 02

목 차

Ⅰ Parity Bit

1. Parity Bit 생성기

XOR Logic은 주로 기기 간의 통신 오류를 검출할 때 사용하는 패리티 비트(Parity Bit)에 이용할 수 있다. 어떤 통신 방법이든지 간에 모든 통신에는 오류 또는 누락이 발생할 수 있는데, 예를 들어 중간에 하나의 비트가 누락될 수도 있고, 노이즈 등 다른 요인으로 인해 신호가 잘못 읽혀질 수도 있다. 그래서 데이터 통신에는 오류 검출 단계가 포함되는 경우가 대다수인데, 이 때 대표적으로 사용하는 방법은 바로 XOR Logic을 이용한 패리티 비트를 추가하는 방법이다.

패리티 비트를 생성하는 방법에는 **홀수 패리티 비트(Odd Parity Bit)**와 **짝수 패리티 비트(Even Parity Bit)**가 있다. 다루고자 하는 데이터가 비트열로 되어있다고 가정하면 여기서 홀수 패리티 비트 방법은 1비트의 개수가 홀수일 때 패리티 비트를 0, 짝수일 때 패리티 비트를 1로 설정하여 전체 데이터에 있는 1비트의 개수가 홀수가 되도록 하는 것이다. 반면에 짝수 패리티 비트 방법은 1비트의 개수가 홀수일 때 패리티 비트를 1, 짝수일 때 패리티 비트를 0으로 설정하여 전체 데이터에 있는 1비트의 개수가 짝수가 되도록 하는 것이다.

실제로 논리 게이트를 사용하여 데이터의 패리티 비트를 생성하는 방법은 다음과 같다. 짝수 패리티 비트 방법으로 패리티를 생성한다고 가정하면 데이터의 각 자리 비트를 모두 XOR 연산한 값이 패리티 비트가 된다. 그 이유는 1비트의 개수가 홀수일 때 각 자리 비트를 모두 XOR 연산을 하면 결과 값이 1로 나오는데, 이는 결국 패리티 비트를 포함하여 전체 1비트의 개수가 짝수가 되기 때문이다. 1비트의 개수가 짝수일 때의 XOR 연산 값은 0이 되므로 짝수 패리티 방법의 결과와 일치한다. 따라서 각 자리의 비트를 모두 XOR 게이트 연산을 시행하는 회로가 패리티 비트 생성기이다. 이를 정리하면 아래와 같다.

[짝수 패리티 비트 생성 방법] $A \oplus B \oplus C \oplus \cdots = P_{\text{짝수}}$

[홀수 패리티 비트 생성 방법] $(A \oplus B \oplus C \oplus \cdots)' = P_{\text{홀수}}$

2. Parity Bit 검사기

패리티 비트 생성기를 통해서 만든 패리티 비트를 통해 수신한 데이터에 오류가 생긴

한 개의 비트가 존재하는지의 여부를 알 수 있다. 앞서 언급한 홀수 패리티 비트와 짝수 패리티 비트 생성 방법에 따라 검출하는 방법도 다르다. 홀수 패리티 비트를 생성했으면 데이터의 비트와 패리티 비트에서 전체 1비트 개수가 홀수 개여야 정상이다. 따라서 데이터의 각 비트와 패리티 비트를 XOR 연산한 값이 1이면 데이터에 오류가 발생하지 않았다는 것을 알 수 있다. 반면에 XOR 연산한 값이 0이면 데이터에 오류가 발생했다는 것이다. 짝수 패리티 비트를 생성했으면 데이터의 비트와 패리티 비트에서 전체 1비트 개수가 짝수 개여야 정상이다. 따라서 데이터의 각 비트와 패리티 비트를 XOR 연산한 값이 1이면 데이터에 오류가 발생한 것이다. 반면에 XOR 연산한 값이 0이면 데이터에 오류가 발생하지 않았음을 알려준다. 따라서 각 자리의 비트와 패리티 비트를 모두 XOR 연산한 값의 결과를 적절히 1과 AND 연산한 값에 따라 정상 또는 비정상 여부를 처리하는 논리 회로가 패리티 비트 검출기다.

단, 여기서 **데이터의 오류가 발생하지 않았다고 해서 그 데이터가 완전히 정확하다고는 판단할 수 없는데**, 그 이유는 데이터에서 짝수 개의 비트가 오류가 생기면 데이터의 비트의 개수가 정상적일 때와 다를 바가 없기 때문에 오류가 발생했는지의 여부를 알 수 없기 때문이다. 다시 말해서, 여기서 말하는 오류는 최소한 한 개의 비트에는 오류가 발생했는지의 여부를 나타내는 상태이다. 이를 정리하면 아래와 같다.

[짝수 패리티 비트 검출 방법]

$A \oplus B \oplus C \oplus \cdots \oplus P_{짝수} = 0$ (오류가 발생하지 않았다.)

$A \oplus B \oplus C \oplus \cdots \oplus P_{짝수} = 1$ (오류가 발생했다.)

[홀수 패리티 비트 검출 방법]

$A \oplus B \oplus C \oplus \cdots \oplus P_{홀수} = 1$ (오류가 발생하지 않았다.)

$A \oplus B \oplus C \oplus \cdots \oplus P_{홀수} = 0$ (오류가 발생했다.)

3. Hamming Code *(Parity Bit 검사기 외의 다른 오류 검출기)*

해밍 코드(Hamming Code)는 패리티 비트처럼 Single Error Correction(단일 오류 검출, 즉 1개의 비트 오류 검출)의 대표적인 방법이다. 해밍 코드를 생성하는 방법은 짝수 패리티 비트 생성과 유사하다. 해밍 코드는 4비트의 데이터 비트 검출에 주로 사용하는데, 이 때 3개의 검출 비트를 가진다. 해밍 코드에서 각 자리 비트를 차례대로 $a_1, a_2, a_3, \cdots, a_7$라고 하면 a_1, a_2, a_4가 검출 비트에 해당하고, 나머지는 데이터 비트에 해당

한다. 각 검출 비트는 데이터 각 자리 비트의 일부와 함께 1비트의 개수가 짝수개가 되도록 만들어 줘야 한다. 예를 들어, a_1는 a_3, a_5, a_7의 1비트의 개수가 짝수 개가 되게끔 값이 설정된다. 각 검출 비트를 구하는 방식을 식으로 정리하면 다음과 같다.

$$a_1 = a_3 \oplus a_5 \oplus a_7$$
$$a_2 = a_3 \oplus a_6 \oplus a_7$$
$$a_4 = a_5 \oplus a_6 \oplus a_7$$

수신한 데이터의 비트의 총 개수가 4개일 때 만들어지는 정상적인 해밍 코드를 표로 정리하면 다음과 같다.

데이터	a_1	a_2	a_3	a_4	a_5	a_6	a_7
0000	0	0	0	0	0	0	0
0001	1	1	0	1	0	0	1
0010	0	1	0	1	0	1	0
0011	1	0	0	0	0	1	1
0100	1	0	0	1	1	0	0
0101	0	1	0	0	1	0	1
0110	1	1	0	0	1	1	0
0111	0	0	0	1	1	1	1
1000	1	1	1	0	0	0	0
1001	0	0	1	1	0	0	1
1010	1	0	1	1	0	1	0
1011	0	1	1	0	0	1	1
1100	0	1	1	1	1	0	0
1101	1	0	1	0	1	0	1
1110	0	0	1	0	1	1	0
1111	1	1	1	1	1	1	1

해밍 코드를 통해서 수신한 데이터가 오류가 없는 데이터인지 판단 가능하다. 예를 들어, 0010011의 해밍 코드를 데이터로 수신했다고 가정하자. 그러면 첫 번째, 두 번째, 그리고 네 번째 비트가 모두 정상적인지 우선 확인한다. 첫 번째 비트는 세 번째, 다섯 번째, 일곱 번째 비트의 1비트 개수가 짝수이므로 0이 된다. 그래서 정상이다. 두 번째 비트는 세 번째, 여섯 번째, 일곱 번째 비트의 1비트 개수가 홀수이므로 1이 된다. 그런데 수신한 데이터의 두 번째 비트는 0이므로 잘못되었다. 네 번째 비트는 다섯 번째, 여섯 번째, 일곱 번째 비트의 1비트 개수가 짝수이므로 0이 된다. 그래서 정상이다. 검출 비트에서 두 번째 비트의 경우에서만 오류가 발생했으므로 두 번째 비트

에서 오류가 발생했다는 것을 알 수 있다.

패리티 비트가 단순히 오류가 발생했는지 여부만을 판단할 수 있었던 반면에, 위의 예시처럼 해밍 코드에서는 어느 비트 자리에서 오류가 발생했는지 유추할 수 있다는 장점이다.

Ⅱ N Bit 비교기

1. N Bit 비교기

비교기는 두 개의 input 값을 비교하여 그 두 input 간의 대소 관계를 판단하는 장치이다. 5주차 실습에서 배웠던 1비트 비교기에서 두 input을 각각 A, B라고 하면 크게 3가지의 output wire를 가지는데, A가 B보다 클 때, A가 B보다 작을 때, 그리고 A와 B가 같을 때 1을 출력하는 wire로 구성되어 있다.

N Bit 비교기는 N개의 비트를 서로 비교하여 그 결과 값을 출력하는 장치를 말하며, 1비트 비교기를 각 비트 자리마다 사용함으로써 더 큰 비트의 비교기를 만들 수 있다. 그러나 N Bit 비교기에서 유의해야 할 점은 모든 비트를 비교할 필요가 없을 수도 있다는 것이다. 만일 input A, B가 각각 n비트로 구성되었으면 적어도 이 회로의 입력 수는 $2n$개가 되며, 이 회로를 설계하기 위해서는 2^{2n}가지의 조합을 갖는 진리표를 작성하고 이로부터 간소화된 논리식을 구해야 한다. 이러한 경우에는 비용이 많이 들기 때문에 간소화할 필요가 있다.

해결 방법의 실마리는 실제 비교 과정에서 얻을 수 있다. 비교하고자 하는 두 수의 자릿수를 맞춘 다음 맨 오른쪽 자리부터 한 자리씩 차례로 대응되는 숫자끼리 비교해 나가면서 각 자리의 대소 비교 결과를 다음 비트 자리 비교 연산에 반영하는 것이다.

A와 B가 같은 경우에는 두 수의 임의의 자리의 비트 A_i, B_i의 값이 모두 1이거나 모두 0인 경우이다. 그 외의 경우에는 두 수가 서로 다르다는 것을 알 수 있다. 또한 A_i가 B_i보다 클 때는 전자가 1이고 후자가 0인 경우이다. 그리고 A_i가 B_i보다 작을 때는 전자가 0이고 후자가 1인 경우이다. 그러나 각 비트마다 비교 연산을 할 때 이전 비트 자리에서 비교한 연산 결과를 반영해야 한다. 예를 들어, 지난 비트 자리에서 A가 큰 상태가 유지되는데 현재 자리에서는 두 비트가 같으면 그래도 A가 큰 상태를 유지하는 것이므로 큰 상태의 output이 1로 유지가 되어야 한다. 반대로 A가 작은 상태에서도 마찬가지이다. 이를 종합하여 정리하면, output들을 각각 x_i, y_i, z_i라고 할 때 다음과 같은 식으로 표현될 수 있다.

$$x_i = x_{i-1} \bullet (A_iB_i + A_i'B_i') \text{ (}A_i\text{와 }B_i\text{가 같은지)}$$
$$y_i = A_iB_i' + x_i \bullet y_{i-1} \text{ (}A_i\text{가 }B_i\text{보다 큰지)}$$
$$z_i = A_i'B_i + x_i \bullet z_{i-1} \text{ (}A_i\text{가 }B_i\text{보다 작은지)}$$

각 자리의 비트를 모두 비교하고 나서 두 수가 서로 같음을 나타내는 output을 X라고 하면 이는 다음과 같은 식으로 표현될 수 있다.

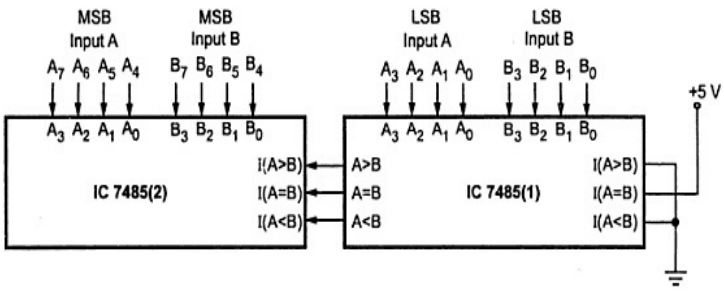
$$X = x_ix_{i-1}\cdots x_1x_0 \text{ (A와 B가 같은지)}$$

A와 B가 모두 같은 경우는 x_i부터 x_0이 모두 1인 경우를 의미하며, 이는 대응되는 모든 비트가 같음을 뜻한다. 위와 같은 방식으로 나머지 output의 논리식을 구하면 다음과 같이 표현된다.

$$Y = A_iB_i' + x_iA_{i-1}B_{i-1}' + \cdots + x_ix_{i-1}\cdots x_1A_0B_0' \text{ (A가 B보다 큰지)}$$
$$Z = A_i'B_i + x_iA_{i-1}'B_{i-1} + \cdots + x_ix_{i-1}\cdots x_1A_0'B_0 \text{ (A가 B보다 작은지)}$$

Ⅲ IC 7485

1. IC 7485



IC 7485는 위의 N Bit 비교기의 진화된 형태이며, 대체로 하나의 IC 7485에서 두 input을 4비트씩 비교하고 그 결과를 다음 IC 7485에 넘겨서 delay 시간을 줄이고자 한다.

마치 앞선 실습에서 배웠던 ripple carry adder의 delay를 줄이기 위해서 4비트 크기의 carry look-ahead adder를 도입하여 지연 시간을 줄이려고 하는 것과 유사하다. 회로의 논리식은 N Bit 비교기에서 유도된 식을 그대로 따른다.

입 력							출력		
A_3, B_3	A_2, B_2	A_1, B_1	A_0, B_0	$I_{A>B}$	$I_{A<B}$	$I_{A=B}$	$O_{A>B}$	$O_{A<B}$	$O_{A=B}$
$A_3>B_3$	X	X	X	X	X	X	1	0	0
$A_3<B_3$	X	X	X	X	X	X	0	1	0
$A_3=B_3$	$A_2>B_2$	X	X	X	X	X	1	0	0
$A_3=B_3$	$A_2<B_2$	X	X	X	X	X	0	1	0
$A_3=B_3$	$A_2=B_2$	$A_1>B_1$	X	X	X	X	1	0	0
$A_3=B_3$	$A_2=B_2$	$A_1<B_1$	X	X	X	X	0	1	0
$A_3=B_3$	$A_2=B_2$	$A_1=B_1$	$A_0>B_0$	X	X	X	1	0	0
$A_3=B_3$	$A_2=B_2$	$A_1=B_1$	$A_0<B_0$	X	X	X	0	1	0
$A_3=B_3$	$A_2=B_2$	$A_1=B_1$	$A_0=B_0$	1	0	0	1	0	0
$A_3=B_3$	$A_2=B_2$	$A_1=B_1$	$A_0=B_0$	0	1	0	0	1	0
$A_3=B_3$	$A_2=B_2$	$A_1=B_1$	$A_0=B_0$	0	0	1	0	0	1
$A_3=B_3$	$A_2=B_2$	$A_1=B_1$	$A_0=B_0$	0	1	1	0	0	1
$A_3=B_3$	$A_2=B_2$	$A_1=B_1$	$A_0=B_0$	1	0	1	0	0	1
$A_3=B_3$	$A_2=B_2$	$A_1=B_1$	$A_0=B_0$	1	1	1	0	0	1
$A_3=B_3$	$A_2=B_2$	$A_1=B_1$	$A_0=B_0$	1	1	0	0	0	0
$A_3=B_3$	$A_2=B_2$	$A_1=B_1$	$A_0=B_0$	0	0	0	1	1	0

위의 표는 IC 7485에서 두 개의 input을 비교할 때 각 경우에 따른 input과 output을 진리표를 그린 것이다.

Ⅳ 기타

1. N비트 비교기 또 다른 종류

앞서 말한 N비트 비교기는 두 input의 크기를 비교하기 위한 목적으로 디자인할 때를 기준으로 설명했다. 그러나 N비트 비교기는 크기 비교뿐만이 아니라 단순히 각 자리 수의 비트를 비교하여 두 개의 비트가 서로 같은지 여부만을 판단하는 비교기도 존재한다. 이런 경우에는 앞서 언급한 크거나 작다는 개념이 들어가지 않고 하나의 비교기에 오직 이전 비트 자리에서 같은지 다른지의 여부만을 판단하는 한 개의 input과 현재 자리에서의 비교 연산의 output이 추가되는 것뿐이다.