Chapter

04

컴퓨터의 두뇌 – CPU

01. CPU의 탄생 비화
02. 범용 논리회로
03. 산술 연산 장치 – Adder의 구현
04. 32비트 ALU의 구현
05. ALU에서의 뺄셈 구현
06. 인스트럭션 맛보기
이것만은 알고갑시다.

☑ CPU와 일반 논리회로의 차이점이 무엇일까요?

시속 300키로를 넘나드는 F1 차량과 엄마가 장 보러 갈 때 타는 경차도 어차피 같은 자동차지만 굳이 F1 차를 '머신1이라 부르는 이유는 사용 용도나 컨셉이 틀리기 때문입니다.

CPU에서도 마찬가지입니다. 논리회로가 주어진 입력(또는 내부 상태값)에 따라 출력이 결정되는 회로라면 CPU 역시 입력에 대해 출력이 결정되는 논리회로에 불과합니다. 단지 CPU는 원하는 용도에 맞는 입력외에 회로의 동작을 결정하는 컨트롤 입력이 별도로 있다는 것이 다를 뿐입니다. 가령 이 컨트롤 입력을 어떤 값으로 주냐에 따라 입력값들을 더하거나 곱하는 등 여러가지 연산을 할 수 있도록 회로를 구성할 수 있을 것입니다. 그저 기계적인 관점에서 보자면 컨트롤 입력과 연산 데이터의 입력 모두를 합쳐 그저 '입력'값으로 보고 이제 맞추어 출력이 나오는 것뿐이지만 우리의 설계 관점에서 보자면 이는 분명히 일반적인 논리회로와는 다른 것입니다. 초창기 CPU는 이러한 컨트롤 입력을 사람이 일일히 손으로 진공관을 뽑았다 꽂았다 하면서 조정하였는데 현재는 메모리에서 자동적으로 이 입력값을 읽어가는 형태로 바뀌었습니다. 그리고 이 입력값을 '인스트럭션'이라고 부릅니다.

Section 01

CPU의 탄생 비화

우리는 앞서 컴퓨터가 왜 등장하였는지, 그리고 이러한 컴퓨터의 발달 과정에 있어 빼놓을 수 없는 디지털 논리회로에 대해서 알아보았습니다. 이미 살펴보았다시피 논리회로는 자판기 같이 주어진 입력(입력 금액, 물건 선택)에 대해 적절한 출력(상품배출, 잔금배출)을 내어주는 절차적 자동화 장치입니다. 옳고 그름의 명확한 2진 논리와 인간의 숫자인 10진수의 2진화를 통해 이러한 논리회로를 설계하여 자판기처럼 단순한 작업을 더 이상 사람이 하지 않아도 되는 것이죠. 하지만 여전히 문제가 있습니다. 자판기 같은 예는 매우 간단한 시스템이지만 프린터 같이 복잡한 시스템을 만들고자 할 때는 논리회로를 어떻게 설계해야 할까요? 또 자판기 회사를 차려서 여러 종류의 자판기를 만들고자 할 때, 과연 그 때마다 이러한 논리회로를 다시 설계해야 할까요?

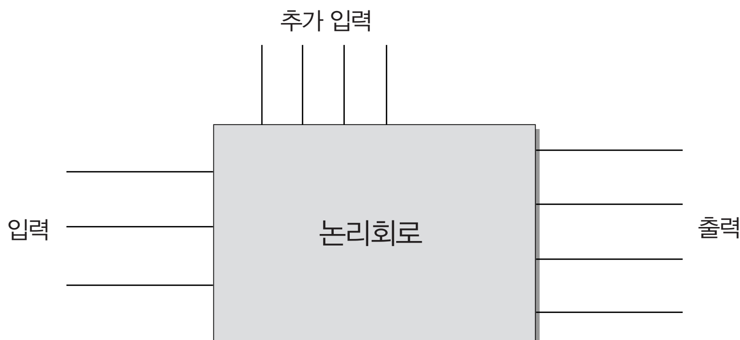
흔히 인간은 생각하는 동물이라 합니다. 그렇기 때문에 일반적인 동물과는 달리 반복되는 단순작업을 더욱 간편하게 해결할 수 있는 방법을 궁리하기 시작했습니다. 그리고 그 결과로 탄생한 것이 바로 CPU(Central Processing Unit – 중앙처리장치)입니다. 우리가 CPU라는 이름에 대해 느끼는 것은 뭔가 막연하게 어렵고, 내가 만들 수 없는 대단한 영역인 것 같은 이미지이지만 사실 그 속을 들여다보면 더도 아니고 덜도 아닌 지금껏 다루어왔던 논리회로의 조합에 불과합니다. 그럼 CPU는 우리가 앞서 살펴 본 자판기의 논리회로와 어떤 차이점이 있을까요?

바로 CPU의 유연성과 범용성이라는 특징입니다. 다시 한번 자판기의 예를 보도록 하죠. 자판기라는 시스템(논리회로)에서 입력은 목이 마른 당신이 투입한 동전과, 음료수의 종류를 선택한 버튼입니다. 이러한 입력은 0과 1을 나타내는 전기적 신호로 바뀌어 논리회로의 입력으로 들어가고, 회로의 출력단에서는 선택한 음료수가 배출되도록 트레이와 같은 기계적 장치에 적절한 접점신호를 주고 또 잔돈의 형태에 맞추어 각 단위(100원, 10원 등)의 동전함에도 신호를 내보냅니다. 이러한 논리회로는 마치 우리가 사다리타기 게임을 하듯이 위에서 어떤 선을 선택하면 그 선을 따라 주어진 규칙대로 흘러가 최종 목적지에 도착하게 되는 것과 같습니다. 만약 앞의 회로에 얼음조각을 섞어주는 얼음선택 버튼이나 커피의 설탕 없음 버튼처럼 추가로 어떤 입력을 가하고 그 입력에 따라 원하는 새로운 결과를 만들어 내려면 논리회로 내부를 다시 설계해야 합니다. 그렇기 때문에 회로를 필요한 기능이 있을 때마다 재설계하지 않도록 앞서 말한 유연성과 범용성을 가진 회로를 처음부터 설계할 수 있다면 좋겠죠? 그럼 새로운 연산을 위해 회로를 재설계하지 않아도 될테니까요.

[그림 4-1] 입력이 조합되어 곧바로 출력이 결정되는 논리회로

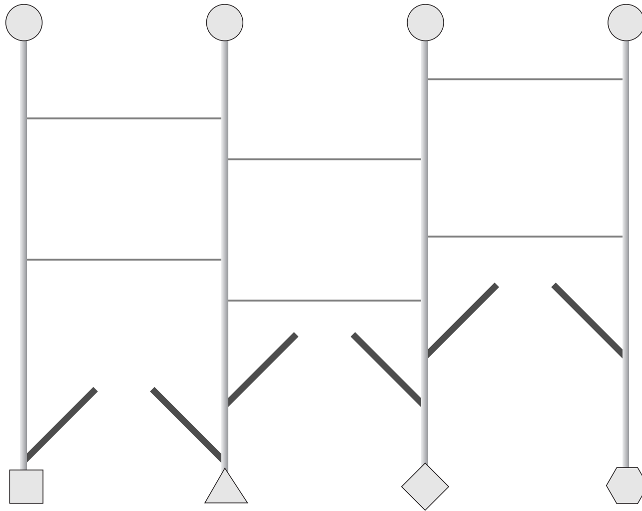
그럼 새로운 회로 설계를 위해 조금 관점을 바꾸어서 생각해보죠. 앞의 자판기의 입출력과 동일한 또다른 자판기 시스템을 만들 때 앞에서와 같이 입력이 주어지면 선을 따라 죽 내려가서 목적지(출력)에 도달하는 수동적인 시스템이 아니라, 좀 더 유연한 형태로 바꾸어봅시다.

이전 시스템은 우리가 필요로 하는 입력만 있었고 논리회로 안에서 그 입력들의 조합으로 순수하게 출력이 결정되었습니다. 이제 여기에 여분의 입력선을 회로에 추가해보겠습니다.



[그림 4-2] 추가 입력을 통해 유연한 조합을 가지게 되는 논리회로

이러한 추가 입력선으로 무엇을 할 수 있을까요? 바로 위에서 언급한 CPU의 특징인 유연성과 (어느 정도의) 범용성을 갖출 수 있게 된 것입니다. 아직 무슨 얘기인지 잘 모르겠죠? 자, 그럼 사다리 게임을 한번 생각해봅시다. 만약 여러분께 아래의 사다리 게임 그림에서 굵게 표시된 세 개의 선을 이었다 떼었다 할 수 있는 막강한 권한이 주어졌다고 가정합시다. 결과가 눈에 보이나요? 100%는 아니지만 사다리의 어떠한 출발점을 고르더라도 여러분은 원하는 도착지점을 저 세 개의 조합을 통해 고를 수 있습니다.



[그림 4-3] 사다리 예제

위의 논리회로에서도 마찬가지입니다. 추가의 입력만 충분히 확보되고 또 그 입력이 적절하게 기존 입력과 조합이 되도록 회로를 꾸민다면 추가 입력을 조정함으로써 기존 입력에 대한 출력 결과를 얼마든지 조절할 수 있는 것입니다.

그런데 눈치 빠른 독자들은 이쯤에서 몇 가지 의문을 품게 될 것입니다.

예를 들면 '그럼 저 추가 입력은 과연 어떻게 결정할 것인가?', '어차피 자판기의 로직이라는 것이 한번 결정되면 변할 일이 없을 텐데 왜 굳이 추가 입력이라는 형태로 만드는 것일까?', '처음부터 논리회로를 그렇게 설계하면 되지 않나?', '또 입력을 결정하였다면 그 입력을 어떤 형태로 줘야 하는가?' 등 꼬리에 꼬리를 무는 질문일 것입니다. 즉, 앞의 두 개의 질문이 추가입력이 무엇이 되어야 하는지의 WHAT이라면, 뒤에 두 가지 질문은 어떻게 입력을 가하는가의 HOW인 것입니다.

그럼 추가 입력을 결정하는 부분에 대해서 생각해봅시다. 우리가 여기서 추가 입력이라는 것을 도입한 것은 '범용성' 때문입니다. 위에서 특정 목적을 위해서 설계한 논리회로는 다른 용도로 사용할 수 없습니다. 매번 용도가 변경될 때마다 논리회로를 다시 설계해야 하죠. 하지만 입출력 관계를 한가지 동작 형태로 고정시켜버리지 않고 덧셈, 뺄셈 등 아주 일반적인 형태의 연산으로 나누어 설계하고 이런 연산 자체를 선택할 수 있도록 추가 입력선을 구성하면 앞으로 그러한 연산을 위해 회로를 재설계해야 할 필요가 없습니다. 즉, 우리가 필요로 하고 앞으로 필요로 할 것 같은 여러 연산(Operation)을 정의하고 입력값은 그 연산의 대상이 되도록, 또한 추가 입력은 그 연산의 종류를 지목하는 값이 되도록 설계하면 어떠한 형태의 로직이 되더라도 이러한 연산의 조

합으로 원하는 목적을 이룰 수 있습니다. 여기까지 읽고 아직 이게 무슨 소린가 하고 난감해 하는 분들, 걱정하지 마세요. 이렇게 몇 줄의 글로 모두 이해할 수 있다면 누가 힘들여 공부하겠습니까? 이제 방금 얘기한 컨셉으로 실제 회로를 한번 설계해보며 더 확실히 이해해봅시다.

이번에는 어떻게라는 부분에 대해서 고민해보죠. 논리회로는 디지털 회로입니다. 즉 0과 1의 조합으로만 이루어져 있고 당연히 입력도 0 아니면 1입니다. 이런 입력은 전압의 높낮이 값으로 결정됩니다. 회로마다 틀리겠지만 일단 3V 이상의 전압이 가해지면 1, 그 아래면 0인 회로라고 가정합시다.

입력값들은 자판기에 투입한 동전이나, 선택 버튼 등으로 결정되어 적절한 전압으로 바뀌어 가해집니다. 즉, 미리 결정해 두는 값이 아니라 그때그때 외부 입력에 따라 결정되는 값입니다. 하지만 추가 입력값은 일종의 논리를 결정하는 값으로, 자판기를 설계할 당시에 결정됩니다. 이렇게 추가 입력이 결정되어 특정 입력을 1로 주고 싶으면 3V 이상의 전압을 가해주면 되므로 슈퍼에서 1.5V 건전지를 2개 이상 사서 직렬로 연결(3V)해 입력단에 걸어주면 됩니다. 0으로 하고 싶으면 회로에서 접지단을 찾아서 연결해주면 되죠.

이는 마치 여러분들이 C 프로그램으로 자판기를 설계하는 것과 비슷한 논리입니다. C로 여러분이 원하는 형태의 프로그램을 만들고 이를 컴파일하여 나오는 바이너리(Binary)가 결국 추가 입력이 되는 것입니다. 또한 프로그램이 실행되면서 사용자에게 입력받는 값이 진짜 '입력'인 것입니다. 추가 입력은 자판기가 동작하기 전에 이미 결정이 나 있어야 하는 것이고, 입력값은 자판기가 동작하면서 동적으로 외부에서 받는 입력이라는 것이죠. 즉, 추가 입력은 프로그램을 짜는 과정이고, 입력은 프로그램 수행 과정이라고 할 수 있습니다. 그리고 CPU는 결국 위 자판기의 논리회로인 것이지요. 사실 지금 한 얘기가 이 책 전체에 걸쳐서 할 가장 큰 주제이고 결론입니다.

쉬운 얘기를 너무 복잡하게 한 것 같나요? 하지만 세상의 모든 복잡한 장치, 심지어 수천만 개의 게이트로 이루어진 최신 펜티엄 CPU도 실상을 들여다보면 아주 간단한 아이디어가 모이고 모여 구성되었다는 것을 곧 알 수 있을 것입니다. 그리고 아직은 어떻게 이런 C 프로그램이 CPU와 연계해서 실행되는지 막막하더라도, 이 책을 덮을 때쯤엔 그 막막함이 사라지게 될 것입니다.

자, 이제 본격적으로 CPU에 대해 얘기해볼까요?

[예제 4-1] 자판기 C 프로그램

```
01 int MONEY,ACC_MONEY,PRODUCT,PRICE,CHARGE;
02
03  START_AGAIN:
04
05  ACC_MONEY =  0;
```

```
06
07   INSERT_MONEY:
08
09   printf("동전을 투입하세요");
10   scanf("%d",&MONEY);
11   ACC_MONEY += MONEY;
12
13   SELECT_PRODUCT:
14
15   printf("물건을 선택하세요");
16   scanf("%d",&PRODUCT);
17   switch ( PRODUCT )
18   {
19   case 1: PRICE = 500; break;
20   case 2: PRICE = 600; break;
21   case 3: PRICE = 700; break;
22   default  : printf("다시 선택하세요"); goto SELECT_PRODUCT;
23   }
24   if ( ACC_MONEY >= PRICE )
25   {
26      CHARGE = ACC_MONEY - PRICE;
27      printf("맛있게 드세요. 잔돈은 %d입니다.",CHARGE);
28      goto START_AGAIN;
29   } else
30   {
31      printf("금액이 모자랍니다.");
32      goto INSERT_MONEY;
33   }
```

Vitamin Quiz

건전지 예제

앞서 언급한 건전지를 통해 추가입력을 조절하는 방식이 이해를 돕기 위한 설명일 뿐일까요? 아니면 실제로 쓰이기도 할까요?

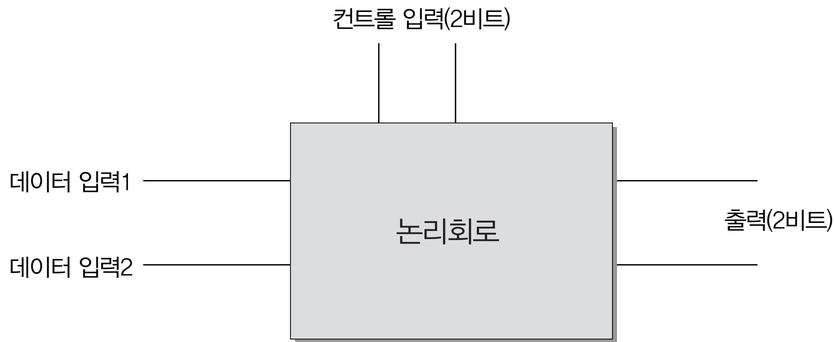
Section 02 범용 논리회로

이제 우리는 두 가지 입력과 출력을 가진 범용적인 논리회로를 만들려고 합니다. 이렇게 만든 회로는 자판기 회사를 차려 여러 종류의 자판기를 한가지 회로 기반으로 만들게 할 수도 있을 것이고, 타임머신을 타고 50년 전으로 돌아가 애니악에 프로그래밍을 하는 장교에게 팔 수도 있을 것입니다.

또다시 개념적인 얘기부터 시작해야 할 것 같군요. 앞서의 자판기 회로에서 다시 시작하도록 하죠. 이 자판기 회로를 설계할 때 처음 입력으로 잡은 부분, 즉 동전의 종류와, 선택된 버튼 등은 해당하는 출력을 만들기 위한 근거 자료, 흔히 데이터라고 표현합니다. 반면에 유연성을 위해 나중에 추가된 입력은 이 논리회로 자체를 변경하여 새로운 논리(로직)를 만들기 위해 회로를 컨트롤하는 부분입니다. 즉, 처음 입력 부분을 데이터 입력, 추가 입력 부분을 컨트롤 입력이라고 부르겠습니다.

컨트롤 입력은 데이터 입력을 잘 조합하여 원하는 형태로 가공하게 하는 입력이죠. 즉, 같은 형태의 데이터 입력이 들어가더라도 컨트롤 입력을 바꾸는 것만으로 출력이 얼마든지 바뀔 수 있는 것입니다.

이제 데이터 입력을 조금 더 확장하여 두 군데에서 데이터 입력을 받을 수 있다고 합시다. 또한 좀 더 단순하면서도 시스템을 구체화시키기 위해 각 데이터 입력은 1비트, 즉 하나의 입력라인으로 이루어져 있고, 출력은 2비트, 즉 두 개의 라인으로 구성되어 있다고 합시다. 우리는 이 시스템을 가지고 4가지 연산–AND, OR, XOR, PASS–을 하려 합니다(PASS는 입력을 그대로 출력으로 내보내는 것을 의미합니다). 즉, 데이터 입력1과 데이터 입력2를 컨트롤 입력의 값에 따라 AND 하거나 OR하기도 하고, 또는 그냥 그대로 출력단으로 내보내기도 하는 회로를 만들려고 하는 것입니다. 따라서 총 연산의 수가 4가지이고 이러한 4가지 연산의 종류를 선택하려면, 2비트의 컨트롤 입력이 필요합니다(동전이 2개 있을 때 총 나타낼 수 있는 경우의 수를 생각해보세요).



[그림 4-4] 컨트롤 입력을 통해 입력 데이터간의 연산이 변경 가능한 범용 논리회로

다음은 앞에서 본 것과 같이 이러한 입출력 관계를 정리한 진리표입니다.

데이터입력1	데이터입력2	컨트롤입력	의미	출력
0	0	00	AND	0X
0	0	01	OR	0X
0	0	10	XOR	0X
0	0	11	PASS	00
0	1	00	AND	0X
0	1	01	OR	1X
0	1	10	XOR	1X
0	1	11	PASS	01
1	0	00	AND	0X
1	0	01	OR	1X
1	0	10	XOR	1X
1	0	11	PASS	10
1	1	00	AND	1X
1	1	01	OR	1X
1	1	10	XOR	0X
1	1	11	PASS	11

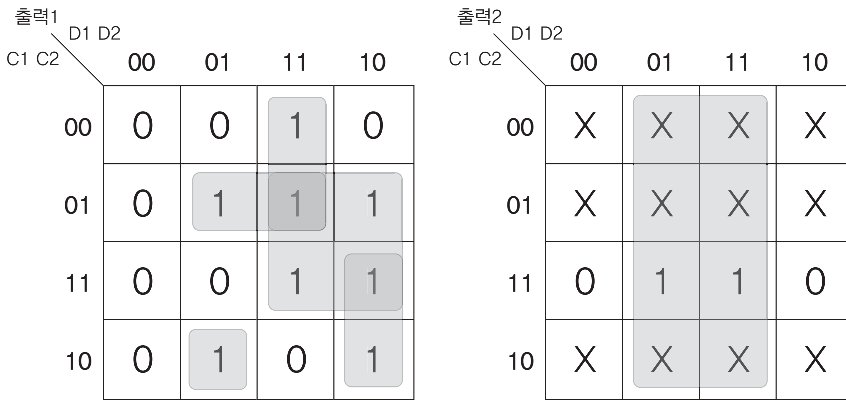
[표 4-1] 범용 논리회로를 위한 진리표

위 표에서 X 표시는 소위 '묻지마 조건'의 표시입니다. AND, OR, XOR 같은 비트 연산을 논리 연산(Logic Operation)이라고 하는데, 이런 논리 연산에서는 두 개의 입력이 들어가면 출력은 하나뿐이죠. 그래서 첫 번째 출력라인으로만 결과를 내보내고 두 번째 라인은 어떤 결과가 나와도 신경 안 쓰겠다는 뜻입니다. 이런 '묻지마 조건'이라는 말이 농담처럼 들리겠지만, 실제로 바다 건너 사람들은 이를 두고 'Don't Care Condition'이라고 합니다.

이제 남은 것은 위 그림의 가운데 논리회로 박스를 어떻게 잘 만들어서 위 표와 같은 결과가 나오게 하느냐입니다. 회로 설계 입장에서 보자면 4비트의 입력(두 개의 데이터 입력이 각각 하나씩, 컨트롤 입력이 두 개)을 받아 2비트의 출력을 내도록 하면 되겠죠.

우선 위 진리표를 보고 이러한 입출력 관계에 대해서는 앞서 공부한 카르노 맵을 그려 입출력 관계를 단순화하고 이에 따라 AND, OR 등의 게이트를 적절히 배치하면 됩니다.

다음은 위 진리표의 카르노 맵입니다.



[그림 4-5] 각 출력의 카르노 맵

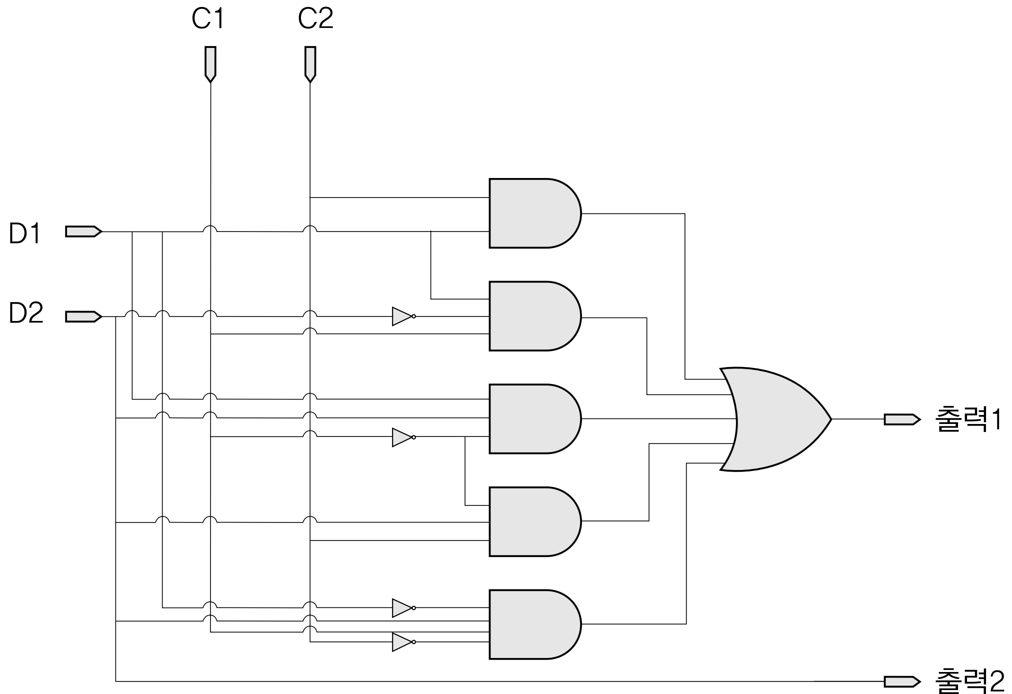
출력1 = $D_1C_2 + D_1\overline{D}_2C_1 + D_1D_2\overline{C}_1 + D_2\overline{C}_1C_2 + \overline{D}_1D_2C_1\overline{C}_2$

출력2 = D_2

우선 위와 같이 카르노 맵에서 공통 분모가 되는 부분을 찾아 그룹핑하여 수식을 간소화합니다. 그리고 이렇게 얻은 결과 수식을 가지고 그대로 게이트로 표현하면 되는 것입니다. 가령 출력 1은 총 5덩어리의 AND를 가지고 있고 이 AND들이 합인 OR로 결합되어 있으므로 5개의 AND 게이트와 한 개의 5입력짜리 OR 게이트를 필요로 합니다.

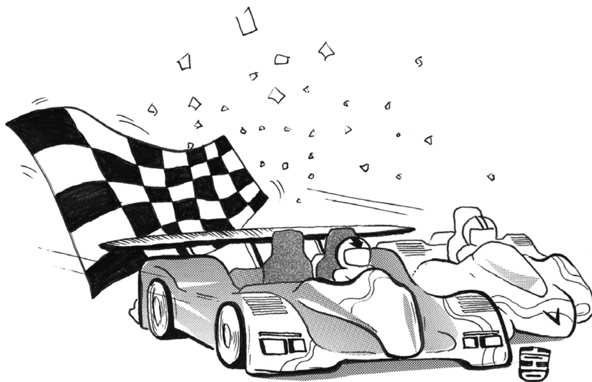
또한 출력 2는 예상할 수 있다시피 데이터 입력을 그대로 내보내는 경우 외에는 모두 Don't care이므로 데이터 입력 D_2가 그대로 나타납니다.

다음 그림은 위 진리표대로 설계한 논리회로의 그림입니다.



[그림 4-6] AND, OR, XOR, PASS가 가능한 논리연산회로

갑자기 기호랑 논리도가 나타나니 아마 또 머리가 지끈한 독자들이 있을 겁니다. 하지만 앞 장을 차근차근 읽어왔다면 방금 얘기한 내용은 큰 무리 없이 받아들일 수 있을 것이라 생각합니다. 혹시, 그래도 아직 이해가 어렵다고 생각하는 독자들도 걱정마세요. 지금까지 배운 논리회로가 어떤 원리로 동작한다는 정도만 이해한다면 내부의 실제 게이트 배치라든지 카르노 맵 등에 대해서는 사실상 개념 정도만 알아도 상관 없습니다. 앞에서 다룬 논리회로는 이 책을 읽는 여러분이 지금 여기서 공부하고 있는 CPU에 대해 잘 이해할 수 있도록 개념을 잡기 위한 과정이었습니다. 이 책에서 다루고자 하는 것은 프로그래머로서 알아두면 좋은 원리들이기 때문에 이런 부분들이 완벽하게 이해되지 않더라도 걱정하지 마십시오. 예를 들어 자동차에 조금 관심 있는 독자라면 알겠지만 수백억의 연봉을 받는 세계적인 드라이버인 마이클 슈마허가 운전을 하기 위해 자동차 엔진을 설계할 줄 알아야 하는 건 아니니까요. 하지만 마이클 슈마허가 단순히 운전 방법만 익히고 감각에만 의존해 운전을 했다면 지금처럼 유명한 선수가 될 수는 없었을 겁니다. 엔진을 설계하진 않더라도 그 구조와 원리에 대해 파악하고 있어야만 그 특성을 최대한 활용할 수 있을 테니까요. 마찬가지로 여러분도 실제 논리회로를 설계하진 않더라도 어떤 형태로 논리회로가 이루어져 있으며 또 동작을 어떻게 하는지 정도에 대한 개념은 파악해야 합니다. 왜냐하면 CPU라는 엔진을 운전하는 사람이 바로 여러분, 즉 프로그래머들이고 그 CPU를 이루는 기본 단위가 바로 논리회로이기 때문입니다.



자, 여기까지 별 문제 없이 잘 따라온 여러분 축하합니다.

벌써 여러분은 CPU의 가장 중요한 핵심 요소 중 하나인 산술논리연산 장치인 ALU(Arithmetic Logic Unit)를 설계하였습니다. ALU는 말 그대로 더하기, 빼기 등의 산술 연산과 방금 만든 것과 같은 AND, OR 등의 논리 연산을 하는 조합 논리회로(Combinational Logic Circuit)를 의미합니다.

엄밀히 따지자면 위에서 만든 회로는 AND, OR 등의 논리 연산만 들어가 있으므로 산술연산은 빠진 '논리연산장치'라고 불러야 하지만, 위 회로에 산술 기능까지 들어간 회로를 꾸미는 일 역시 그리 큰 일이 아닙니다. 그 외에도 몇 가지 생각해야 할 부분이 있지만 위 개념만 이해하고 있는다면 상식적인 수준에서 이루어집니다.

Section 03 산술 연산 장치 – Adder의 구현

이제 위 논리 연산 장치에 산술 기능을 더하여 본격적인 ALU를 만들기 위해 먼저 덧셈에 대해서 알아보겠습니다. 일단 가장 익숙한 10진수의 덧셈을 생각해봅시다.

3 + 4 = 7
5 + 3 = 8
1 + 4 = 5
.
.
.

너무나 쉬운 덧셈입니다. 그럼 다음은 어떤가요?

5 + 7 = 12
4 + 9 = 13
9 + 9 = 18
.
.
.

첫 번째와 두 번째 그룹의 덧셈은 어떤 차이가 있을까요? 두 종류 모두 단자리 덧셈입니다만, 결과는 어떤가요? 첫 번째는 덧셈 결과 역시 단자리 수이지만 두 번째는 자리 올림이 발생하여 두 자리 수가 되었습니다. 이를 2진수의 세계에 적용해도 마찬가지입니다. $1_2 + 1_2 = 10_2$이 되는 식이죠. 이렇게 발생한 자리 올림 값을 캐리(Carry)라고 합니다.

위에서 설계한 논리 연산 유닛에서는 입력이 1비트씩 각각 2개, 출력은 1비트 하나였습니다(앞서의 논리 연산기에서 출력이 두 개 있는 것은 PASS를 위한 경우이고 논리연산 자체는 첫 번째 출력만 사용하고 있습니다). 하지만 덧셈과 같은 산술 연산에서는 자리 올림, 즉 캐리가 발생하므로 캐리를 위한 출력 라인이 하나 더 있어야 합니다. 다행히도 우린 PASS 기능을 위해 이미 두 번째 출력 라인을 확보해 두었습니다.

또하나 변경해야 할 점은 컨트롤 라인입니다. 이미 기존에 4종류의 연산을 정의하였기 때문에 ADD라는 연산을 추가하기 위해선 2비트의 컨트롤 라인을 확장해야 합니다. 3비트로 확장하게 되면 총 9가지의 연산이 가능한데, 우선 여기선 PASS 기능을 삭제하고 대신 ADD 연산을 넣어서 컨트롤 라인을 2비트로 유지합시다.

다음은 앞의 진리표에서 PASS 대신 ADD를 넣어 수정한 진리표입니다. 음영 부분이 수정된 부분입니다.

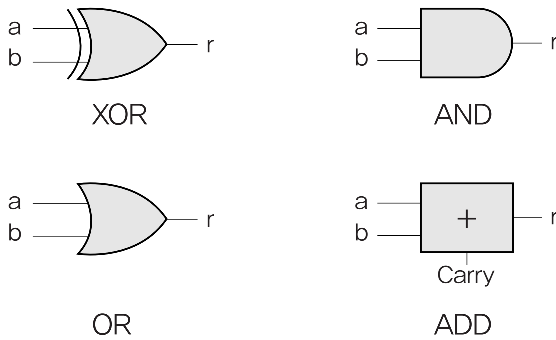
데이터입력1	데이터입력2	컨트롤입력	의미	출력
0	0	00	AND	0X
0	0	01	OR	0X
0	0	10	XOR	0X
0	0	11	ADD	00
0	1	00	AND	0X
0	1	01	OR	1X
0	1	10	XOR	1X
0	1	11	ADD	01
1	0	00	AND	0X
1	0	01	OR	1X
1	0	10	XOR	1X
1	0	11	ADD	01
1	1	00	AND	1X
1	1	01	OR	1X
1	1	10	XOR	0X
1	1	11	ADD	10

[표 4-2] PASS 대신 ADD가 들어간 진리표

자, 이번에는 앞에서와 조금 다른 방식으로 회로를 설계해보겠습니다. 앞에서는 진리표가 주어지면 카르노 맵을 그리고 맵에서 간소화한 식대로 게이트를 배치하여 회로를 구성하였습니다만, 이제는 블록화라는 개념을 도입하여 좀 더 단순한 형태로 이를 구성해보죠. 실제 회로를 설계함에 있어서도 모든 입출력 관계를 고려하여 일일이 게이트를 그려나가지는 않습니다. 자주 쓰이는 용도의 회로를 구성하여 모듈화하고 이런 모듈, 즉 블록을 연결해 점점 큰 회로를 구성해나가는 방식이죠.

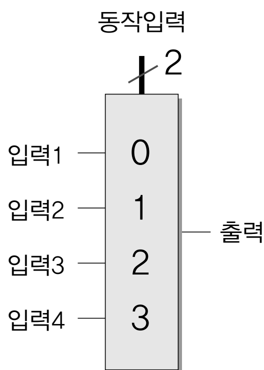
우선 입출력과 모듈을 다시 정의해봅시다. 데이터 입력은 두 개의 1비트 데이터로서 a, b로 나타내기로 하죠. 그리고 각 연산의 결과값은 r로 나타내겠습니다. 다음으로 우리가 필요로 하는 연산의 종류는 AND, OR, XOR 그리고 ADD 4가지입니다. 그 중 AND, OR, XOR 등은 기본 게이트이므로 하나의 모듈로 나타낼 수 있습니다. 하지만 ADD의 경우는 그 자체가 입력과 출력을 가지면서 AND, OR 등의 기본 게이트로 구성된 또 다른 하나의 논리회로입니다. 이러한 ADD 회로는 두 입력 a, b와 출력 r 및 캐리(자리올림) 값에 대한 진리표를 작성하여 앞서와 마찬가지로 논리회로를 설계할 수 있습니다. 여기서는 이 과정은 생략하고 모듈화된 ADD회로를 사용하도록 하겠습니다.

다음은 우리가 지원하고자 하는 각 연산별 모듈을 나타낸 그림입니다.



[그림 4-7] 각 연산 모듈

각각의 연산 모듈은 두 개의 데이터 입력 a, b를 받아들여 출력 r을 생성합니다(단, ADD 모듈은 캐리(Carry)출력이 하나 더 있습니다). 이제 남은 것은 어떻게 이 각각의 연산기를 선택하고 선택된 결과값을 출력으로 받아들이냐 하는 부분입니다. 이에 대한 해답은 바로 MUX(Multiplexor)입니다. MUX는 잘 알다시피 여러 개의 입력 중 하나를 선택하여 그 값을 출력으로 내보내는 논리회로입니다. 이 때 입력들을 선택하기 위해서 동작(operation) 입력이 따로 있었습니다. 우리는 총 4가지 연산 중 하나를 선택할 것이므로 2비트의 동작입력이 필요합니다. 다음은 MUX 모듈을 나타낸 그림입니다.



[그림 4-8] 4 입력 MUX

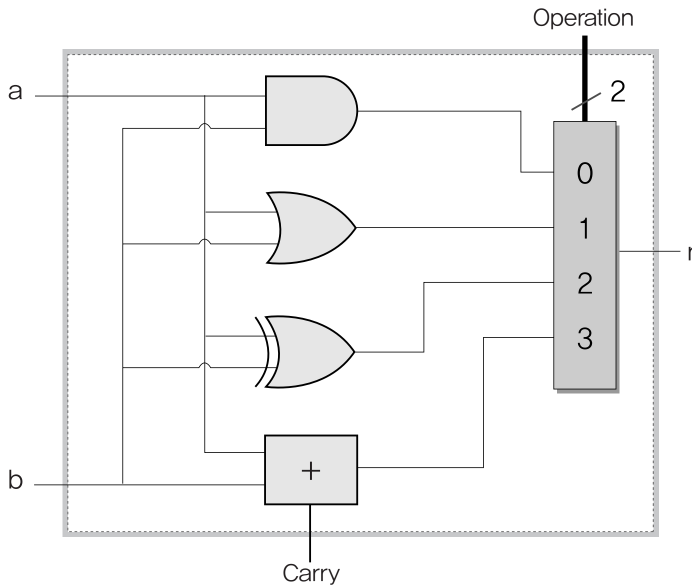
이제 우리가 원하는 결과에 거의 가까워졌습니다. 남은 것은 이 모듈을 잘 결합하여 하나의 논리회로로 구성하는 것뿐입니다. 각각의 연산 모듈에 공통적으로 a, b의 입력을 가하고 각 모듈의 출력을 MUX에 연결한 후 이를 동작입력을 통해 선택한 MUX의 출력을 전체 회로의 최종 출력값으로 사용하면 되는 것입니다.

그런데 우리가 지금까지 언급하지 않은 항목이 하나 있습니다. 바로 컨트롤 입력입니다. 그렇다

면 이 컨트롤 입력은 이 회로에서 어떻게 사용될까요? 눈치 빠른 독자들은 이미 파악하였을 겁니다. 바로 동작입력이 컨트롤 입력인 것입니다. 위 진리표처럼 컨트롤 입력 00은 AND, 01은 OR, 10는 XOR, 11은 ADD가 되게 하려면, 이 컨트롤 입력을 MUX의 동작입력으로 삼고 각 연산 모듈(AND, OR, XOR, ADD)을 MUX에 방금 얘기한 순서대로 연결하면 되는 것입니다. 즉, AND는 입력1에, OR은 입력2 같은 방식인 것이죠.

그럼 캐리(Carry) 출력은 어떻게 할까요? 이 캐리 출력에 대해선 바로 뒤에서 다시 논의하도록 하고, 우선 MUX에 연결하지 않고 별도로 빼 놓겠습니다.

다음은 전체 모듈이 결선된 최종 회로도입니다.



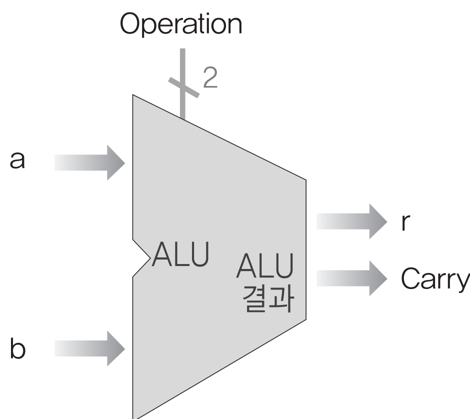
[그림 4-9] ADD가 추가된 ALU

자, 드디어 덧셈까지 완성된 ALU(산술논리연산장치)를 설계하였습니다. 위 ALU는 앞서 직접 카르노 맵을 그려서 일일이 게이트를 나열하여 설계한 경우보다 훨씬 심플하게 보이면서도 ADD 기능까지 추가되어 있습니다. 여러분이 C로 프로그램을 만들 때랑 똑같은 이치입니다. C 프로그램을 main안에 처음부터 끝까지 코드를 나열하여 작성해도 되지만 대부분의 경우는 기능별로 함수를 만들고 이 함수를 호출하는 방식으로 작성합니다. 마찬가지로 하드웨어에서도 블록 단위로 작성하여 해당 블록을 함수처럼 입출력 관계만 파악하여 가져다 쓰면 보다 큰 규모의 장치도 단계적으로 수립해나갈 수 있습니다.

Section 04
32비트 ALU의 구현

지금까지 여러분은 1비트 ALU를 작성하였습니다. 이제 여기서 좀 더 확장하여 펜티엄에서 사용하는 32비트용 ALU를 작성하려면 어떻게 해야 할까요?

이 질문에 대답을 하기 위해선 우선 몇 가지 정리해야 할 사항들이 있습니다. 앞서 작성한 1비트용 ALU를 이제 내부를 신경 쓰지 않기 위해 전체 덩어리를 하나로 모듈화합시다.



[그림 4-10] 1비트 ALU 모듈

위 ALU는 a, b 두 입력 데이터를 받아 동작입력(Operation)값으로 선택된 연산을 행하여 그 결과를 r로 출력합니다. 단 ADD 연산인 경우에는 캐리 출력으로 자리 올림 결과가 출력됩니다.

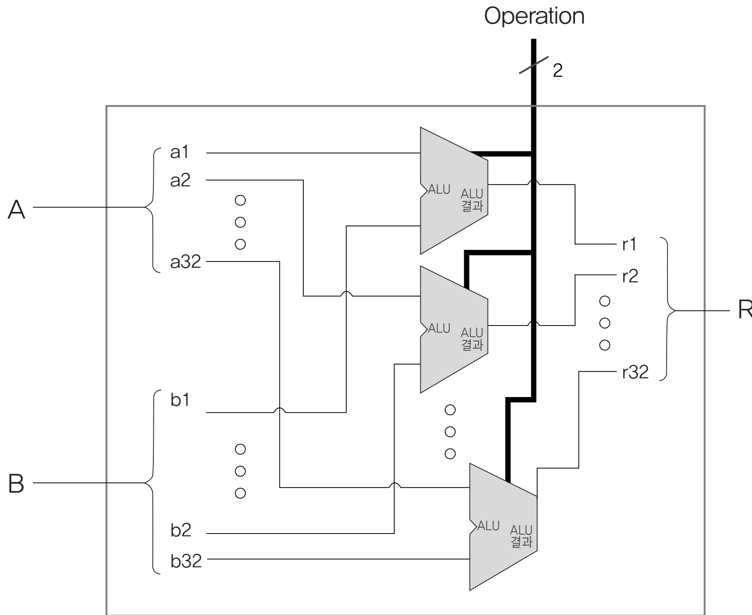
우리가 하고자 하는 연산은 총 4가지이므로 동작입력(Operation)은 2비트 라인으로 설정되어 있습니다. Operation 라인에 빗금치고 2라고 적힌 것이 바로 2비트 라인이 들어간다는 의미입니다.

Vitamin Quiz

Add 이외 연산의 캐리 출력

위 ALU에서 연산의 결과들은 r로 출력되고 ADD일 경우에만 캐리 출력으로 자리 올림 결과가 나온다고 하였습니다. 그렇다면 ADD가 아닌 다른 연산일 경우에 캐리 출력은 어떤 값이 나올까요?

이제 32비트 ALU를 만들기 위한 고민을 시작하도록 합시다. 이를 위한 가장 단순하고도 명쾌한 답은 바로 1비트 ALU 32개를 나란히 나열하여 사용하는 것입니다. a, b 두 비트의 AND 연산이나 32개의 ALU를 나열하여 각각의 입력 (a1, b1), (a2, b2)…, (a32, b32)을 연산하여 나온 32개의 출력이 바로 32비트 연산 결과인 것이죠. 즉 a1~a32까지가 32비트의 새로운 A 입력이 되고, b1~b32까지가 32비트의 또 다른 B 입력이 되는 것입니다.



[그림 4-11] 1비트 ALU를 32개 나열해 만든 32비트 ALU 모듈

어떻습니까? 아주 그럴듯해 보입니다. 실제 AND나 OR 연산 등의 논리 연산을 함에 있어서는 아무 문제될 것 없는 훌륭한 32비트 ALU입니다. 하지만 ADD와 같은 산술 연산인 경우에는 어떨까요? 다음의 계산을 수행해보면서 파악해봅시다.

$$\begin{array}{r} 0000.....001101_2 \\ +\ 0000.....001001_2 \\ \hline 0000.....0?????_2 \end{array}$$

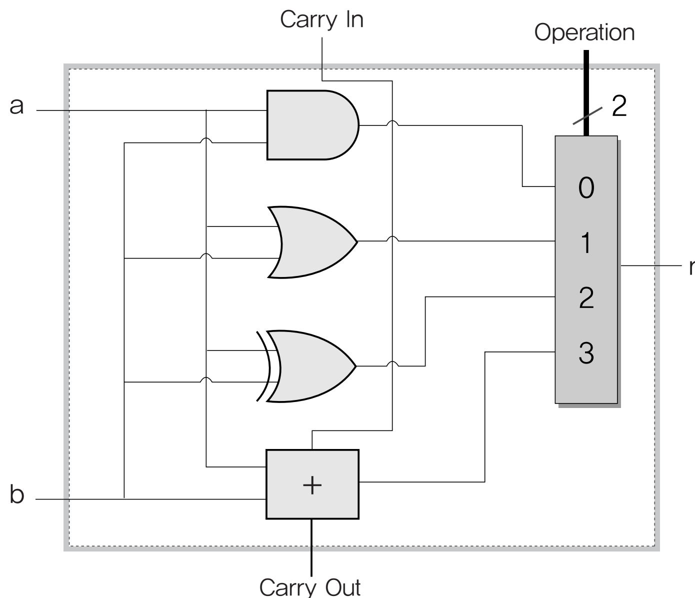
[그림 4-12] 2진수 덧셈 예제

우선 실제 답을 구해보면 $0000.....010110_2$이라는 것을 쉽게 알 수 있습니다. 하지만 위

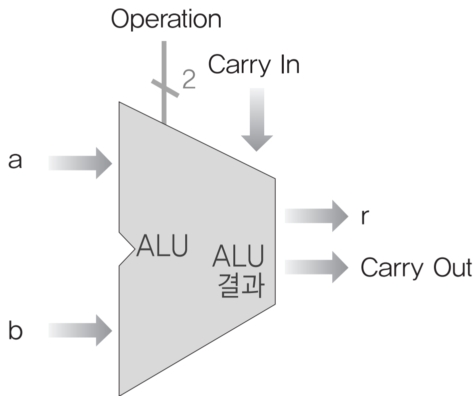
ALU를 이용하여 계산하면, 각 1비트 ALU들이 서로 독립적으로 계산을 하므로 엉뚱하게도 $0000.....000100_2$이라는 틀린 답이 나오게 됩니다. 이는 바로 자리 올림을 고려하지 않았기 때문에 발생하는 문제입니다. 가령 제일 하위 비트를 계산하면 $1_2 + 1_2 = 11_2$이 되어야 합니다만, 발생한 자리 올림, 즉 캐리(Carry)는 위 회로에서 각 1비트 ALU들끼리 전혀 연결되어 있지 않기 때문에 이런 문제가 발생하는 것입니다. 따라서 각 1비트 ALU들이 발생한 캐리(Carry)를 다시 상위비트에 반영할 수 있도록 회로가 수정되어야 합니다.

우리가 앞에서 설계한 1비트 ALU에는 다행히도 캐리 출력이 있습니다. 그렇다면 이 캐리 출력을 인접한 상위 비트 ALU에 어떻게든 전달하기 위해서는 어디에 연결해야 할까요? 실제 덧셈 연산을 잘 살펴보면 두 비트의 값만 더하는 게 아니라 인접한 아랫자리에서 발생한 자리올림 값, 즉 캐리(Carry)도 같이 더해야 한다는 것을 알 수 있습니다. 하지만 우리가 설계한 ALU에는 캐리(Carry) 출력은 있는데, a, b와 함께 더해질 캐리의 입력이 없습니다. 그럼 이런 사실을 바탕으로 1비트 ALU를 살짝 수정해보도록 하겠습니다.

다음 그림은 캐리 출력(Carry Out)뿐 아니라 캐리 입력(Carry In)까지 고려한 1비트 ALU입니다.

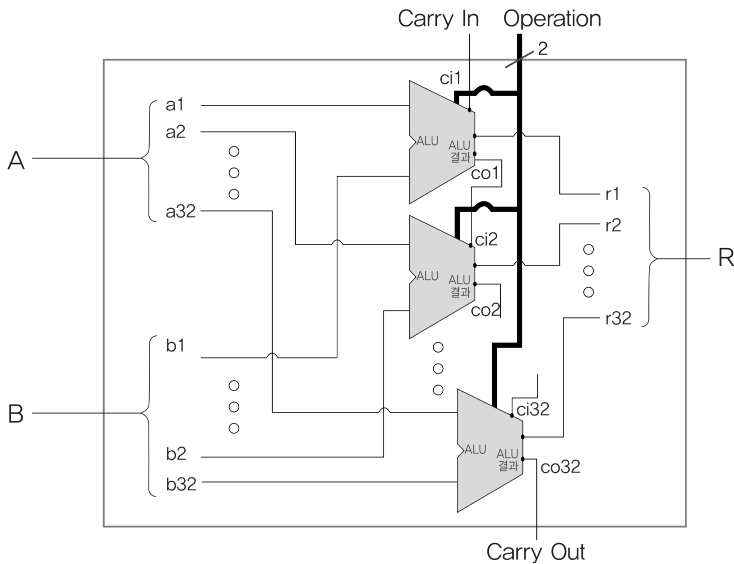


[그림 4-13] Carry 입출력이 모두 갖춰진 1비트 ALU – 내부 구조



[그림 4-14] Carry 입출력이 모두 갖춰진 1비트 ALU – 심볼

위 ALU를 이용하여 32비트 ALU를 설계할 때 앞서 설계한 32비트 ALU와 다른 점은 각 인접 비트끼리 캐리 입출력을 연결한다는 점입니다. 즉 하위 비트에서 발생한 캐리 출력은 상위 비트의 캐리 입력으로 연결해야 합니다.



[그림 4-15] 캐리 입출력이 모두 갖춰진 32 비트 ALU

위 회로는 기능적으로는 실제 사용되는 ALU로서도 아무 손색이 없습니다. 하지만 1비트 ALU에 비해 치명적인 단점이 있습니다. 바로 속도가 느리다는 점입니다. 이는 게이트의 전기적 특성에 의한 것인데, 가령 AND 게이트의 입력에 (1,1)을 주고 출력단에서 1이라는 결과가 나타나기까지 어느 정도 시간이 걸립니다. 이러한 게이트의 집합으로 이루어진 ALU도 마찬가지로 a, b의 입력

이 주어지고 출력이 입력에 맞게 변하기까지 시간이 걸리죠. 그런데 위에서 설계한 32비트 ALU의 경우 입력이 주어지면 우선 제일 하위 비트의 ALU 출력이 변하고 이 변화된 출력 중 캐리가 인접한 상위 비트 ALU의 캐리 입력으로 전달되는 시간이 걸린 후 두 번째 캐리 아웃이 출력됩니다. 이런 식으로 제일 하위비트의 캐리가 물결 타듯 상위 ALU로 전달되어 마지막 32번째 ALU까지 캐리가 전달되려면 1비트 ALU 때의 32배의 시간이 걸리는 것이죠. 결국 동시에 할 수 있는 계산량은 32배가 되었지만 걸리는 시간도 32배가 되었기 때문에 큰 의미가 없습니다. 이런 방식의 구성을 마치 물결치듯이 캐리가 전달되어간다 하여 리플 캐리(Ripple Carry)라고 합니다.

리플 캐리 방식은 속도상의 문제로 인해 실제로는 잘 사용되지 않습니다. 대신 발생할 캐리를 미리 입력값에서 한꺼번에 계산하는 캐리 룩 어헤드(Carry Look Ahead)라는 방식이 사용됩니다만, 여기서는 다루지 않겠습니다. 여러분은 다만 ALU라는 것이 무엇이고 어떠한 형태로 동작한다는 개념적인 부분만 확실히 이해하면 됩니다.

Section 05 ALU에서의 뺄셈 구현

이제 뺄셈 연산에 대해 생각해보겠습니다. 결론부터 얘기하자면 2진수에서 뺄셈 연산은 덧셈 연산을 살짝 수정하는 것만으로 구현할 수 있습니다. 아주 간단한 발상의 전환입니다만 실질적으론 매우 유용한 방법이죠. 바로 뺄셈 대신 음수의 덧셈으로 바꾸는 것입니다. 가령 5 – 3을 5 + (–3)과 같이 바꾸어 생각하는 것이죠.

이를 위해 필연적으로 등장하는 것이 음수라는 개념입니다. 그럼 우리가 흔히 사용하는 음수를 컴퓨터에서는 어떻게 나타낼까요? 바로 보수라는 개념을 사용합니다.

두 개의 자연수가 합이 N이 되는 관계에 있을 때 이 두 수를 'N의 보수관계에 있다'라고 부릅니다. 가령 십진법에서 3에 대한 9의 보수는 6입니다. 그럼 보수가 2진수에서 어떻게 음수를 나타내는지 알아보죠.

우선 어떤 숫자를 4비트의 2진수로 나타내는 경우에 대해서 생각해보겠습니다. 4비트 2진수로는 총 2^4 = 16 즉, 16가지 숫자를 표현할 수 있습니다. 우리는 흔히 이 16가지 숫자를 0~15까지의 양수로만 맵핑하여 사용합니다만, 음수를 표현해야 할 경우에는 절반 정도, 즉 0을 제외한 7~8개 정도는 음수 부분으로 할당해야 합니다. 또한 음수라고 정해진 수는 쌍을 이루는 양수와 더하

면 0이 되는 성질을 가져야만 합니다. 이러한 두 가지 전제를 만족시키기 위해서 보수의 성질을 이용하면 문제가 쉽게 해결됩니다. 즉, 두 수를 더해서 특정 숫자가 될 때 이 두 수를 그 특정 숫자의 보수 관계에 있다고 하므로 4비트로 표현 가능한 0~15 중 가장 끝에 있는 수, 즉 0000_2이나 1111_2를 하나 골라 이를 0이라 하고 그 수의 보수 관계에 있는 수를 추려서 이 중 절반을 양수, 절반을 음수로 취하면 위 조건을 만족합니다. 이 때 1111_2을 0으로 삼고 모든 숫자들에 대해 1111_2의 보수를 취하는 경우는 더해진 합이 1111_2이고 이 때 각 비트가 1이 되므로 1의 보수라 하고, 0000_2를 기준으로 하는 경우에는 그 합이 각각 0이므로 2의 보수가 됩니다(2진법에서는 한 자리만 놓고 보면 1 + 1 = 0이 되므로 2의 보수라 합니다). 이 중 우선 1의 보수인 경우에 대해서 생각해봅시다.

1의 보수에선 두 수의 합이 1111_2이 되어야 하므로, 모든 경우에 대해 보수 관계를 표시하면 아래 표와 같습니다.

Index	숫자 1	숫자 2	합
0	0000	1111	1111
1	0001	1110	1111
2	0010	1101	1111
3	0011	1100	1111
4	0100	1011	1111
5	0101	1010	1111
6	0110	1001	1111
7	0111	1000	1111
8	1000	0111	1111
9	1001	0110	1111
10	1010	0101	1111
11	1011	0100	1111
12	1100	0011	1111
13	1101	0010	1111
14	1110	0001	1111
15	1111	0000	1111

[표 4-3] 4비트 2진수에서 1의 보수표

위 테이블은 1111_2의 보수 관계를 나타낸 표이므로 각 행의 양 숫자를 더하면 항상 1111_2입니다. 따라서 1111_2를 의미상 0으로 삼으면 같은 행에 있는 숫자 1, 2는 절대값이 같고 부호가 다른 두 숫자로 생각할 수 있습니다. 또한 이를 자세히 관찰해보면 인덱스 0~7번의 좌측에 해당하는 부분이 인덱스 8이 넘어가면서 우측에 다시 나타나고 있습니다. 즉, 1000_2를 기준으로 두 그룹으로

나뉘며 이 중 한 그룹을 양수, 나머지 한 그룹을 음수로 사용할 수 있는 것이죠. 한편 0에 대한 보수는 결국 또 0이 되게 되므로 0으로 삼은 1111_2에 대한 보수인 0000_2역시 의미상 0이 됩니다. 그리고 왼쪽의 음영 부분(인덱스 1~7)을 양수 1~7로 삼으면 이에 해당하는 우측 숫자(숫자 2)라인들은 (–1) ~ (–7)이 됩니다.

이번에는 2의 보수에 대해 생각해보죠.

2의 보수에서는 두 수의 합이 0000_2(실제로는 10000_2이지만 4비트뿐이므로)이 되어야 하며 아래 표와 같은 보수 관계를 가집니다.

Index	숫자 1	숫자 2	합
0	0000	0000	0000
1	0001	1111	0000
2	0010	1110	0000
3	0011	1101	0000
4	0100	1100	0000
5	0101	1011	0000
6	0110	1010	0000
7	0111	1001	0000
8	1000	1000	0000
9	1001	0111	0000
10	1010	0110	0000
11	1011	0101	0000
12	1100	0100	0000
13	1101	0011	0000
14	1110	0010	0000
15	1111	0001	0000

[표 4–4] 4비트 2진수에서 2의 보수표

이번에는 기준 값인 0000_2을 0으로 삼으면 공교롭게도 그에 대한 보수가 또다시 자기 자신이 되므로 1의 보수에서처럼 의미상 0이 두 번 나타나지 않습니다. 또한 1의 보수에서처럼 1000_2부터 다시 왼쪽에 나타났던 숫자가 나타나므로 1~7번 인덱스까지의 왼쪽 숫자를 양수로 보고 8~15까지를 음수로 볼 수 있습니다. 이 때 눈치 빠른 독자는 1000_2은 어차피 보수 관계에 있는 수가 똑같은 1000_2이므로 굳이 음수로 삼지 말고 양수 8로 삼아도 될 것이라 생각할 겁니다. 사실 그렇게 해도 상관 없습니다. 그런데 여기서 또 중요한 사실이 하나 있습니다. 가만히 테이블을 살펴보면 인덱스 1~7번까지의 좌측 숫자, 즉 양수가 확실한 녀석들의 최상위 비트는 모두 0이고 이의 음수 짝에 해당하는 오른쪽 값들은 모두 최상위 비트가 1로 셋팅되어 있다는 점입니다. 이는 1의

보수 테이블을 보아도 명확히 드러나는 점인데, 이 때문에 양수와 음수를 판단하는데 최상위 비트 하나만 보고 판단할 수 있는 매우 큰 장점을 지니게 됩니다. 그런데 만약 8번의 1000_2을 양수 8로 취급하면 이런 장점을 활용할 수 없죠. 따라서 1000_2은 8이 아닌 –8로 취급하는 것이 일반적입니다.

이제 1의 보수와 2의 보수 둘 중에 어떤 것이 과연 컴퓨터 세계에서 편리한지 알아보겠습니다. 우선 1의 보수는 좌우측 숫자, 즉 부호가 다른 숫자들의 모든 비트가 서로 반전되어 있습니다. 따라서 앞서 설계한 ALU에서 뺄셈 연산을 하기 위해서는 단순히 b 대신에 반전된 b를 입력으로 사용해 ADD 연산할 수 있습니다. 하지만 이 때 주의할 점이 하나 있습니다. ADD 연산 결과 최상위 비트에서 자리 올림, 즉, 캐리가 발생하면 결과값에 1을 더해줘야만 합니다. 예를 들어 2 – 5의 경우는 2 + (–5) = 0010_2 + 1010_2 = 1100_2 = –3이 되므로 캐리가 발생하지 않은 경우는 그대로 사용할 수 있지만, 5 – 2 = 0101_2 + 1101_2 = $(1)0010_2$ = 2와 같이 캐리가 발생하게 되는 경우는 결과가 다르게 나옵니다. 이 때는 결과값에 1을 더해야만 정확한 값 3을 얻을 수 있습니다.

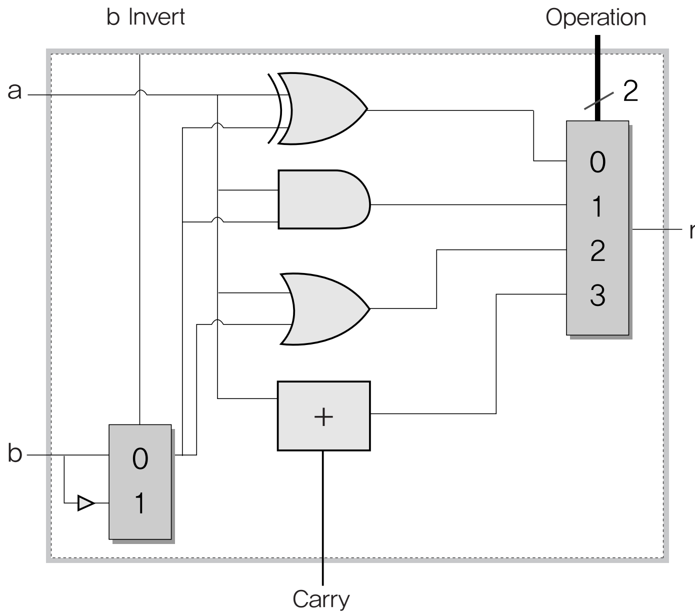
반면 2의 보수는 어떨까요? 2의 보수는 일견 왼쪽 숫자와 비교해 별 규칙이 없는 듯하지만, 조금만 더 자세히 살펴보면 다음과 같은 규칙을 알 수 있습니다. 왼쪽의 모든 비트를 반전한 뒤 1을 더하면 바로 2의 보수가 됩니다. 1의 보수에 비해서 다소 복잡해 보이죠? 하지만 이는 아주 간단하게 해결될 수 있습니다. 1의 보수와 마찬가지로 b를 반전하여 입력으로 주고 똑같이 ADD 연산을 합니다. 단, 이 때 캐리 입력을 1로 주고 합니다. 어차피 캐리 입력은 사용하기 위해 있는 것이고 평상시에는 최하위 캐리 입력은 0으로 입력해야 했지만, 뺄셈 연산일 경우에는 0 대신 최하위 캐리 입력에 1을 입력하고 반전된 b입력으로 ADD 연산을 하면 결국 별도의 하드웨어 추가 없이 쉽게 뺄셈이 가능합니다. 또한 1의 보수에서와는 달리 캐리 발생에 따른 추가 작업(1을 더해야 하는 등)이 없게 되므로 실질적으론 더 간단해집니다.

그렇다면 이번에는 다른 측면에서 관측해보겠습니다.

4비트로 나타낼 수 있는 숫자는 총 16가지입니다. 이를 1의 보수에서는 양수 1~7 과 음수 –1 ~ –7, 그리고 두 개의 0 (0000_2, 1111_2)으로 나눠 사용합니다. 하지만 2의 보수에서는 양수는 1의 보수에서처럼 1~7까지만 0이 하나이며 대신 음수를 –8까지 하나 더 쓸 수 있습니다. 이는 효율적인 측면에서 따졌을 때 무척 큰 부분이며 결국 이러한 하드웨어적인 구현의 편의성과 숫자 사용의 효율성 등의 이유로 컴퓨터 세계에서는 2의 보수의 사용이 아주 보편화되어 있습니다.

자, 이제 2의 보수를 사용하여 뺄셈까지 구현한 ALU를 작성해 보죠. 우선 현재까지 사용하고 있는 연산의 종류에 뺄셈을 추가하게 되면 AND, OR, XOR, ADD에 새로 추가되는 뺄셈인 SUB까지 총 5종류가 됩니다.

그렇다면 지금까지의 4입력 MUX 대신 3비트의 동작입력과 이에 따라 총 8개까지 입력이 가능해진 MUX를 사용해야 할까요? 그렇지 않습니다. SUB 연산은 별도로 모듈이 추가되는 것이 아니라 2의 보수를 사용해 단순히 b 입력을 반전시키고 ADD 연산을 하면 됩니다. 따라서 SUB 연산을 할 경우엔 동작입력을 ADD에 해당하는 10_2을 넣어주고 b 입력을 반전해 넣어주면 됩니다. 이를 위해 b 입력을 이등분하여 이 중 하나엔 NOT 게이트를 추가하여 둘 중 하나를 선택할 수 있도록 2입력 MUX를 사용해야 합니다. 따라서 수정된 1비트 ALU는 아래와 같이 구성됩니다.



[그림 4–16] SUB까지 구현된 1비트 ALU

Section 06 인스트럭션 맛보기

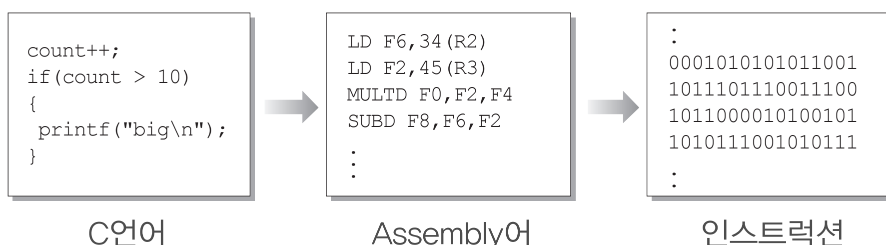
자, 이제 1부의 가장 중요한 테마인 인스트럭션(Instruction)이라는 것에 대해 잠깐 짚고 넘어갑시다. 비록 아직 우리가 CPU 전체를 구현한 건 아니지만, 이쯤에서 인스트럭션이라는 단어의 윤곽 정도는 잡고 가는 것이 좋기 때문이죠. 인스트럭션은 뒤에서 다시 자세히 다루도록 하겠습니다.

인스트럭션은 말 그대로 '지시' 내지는 '명령'이라는 뜻입니다. 이 책을 읽고 있는 독자라면 아

마 C언어는 물론이고 인스트럭션이니 어셈블리(Assembly)어(語) 등에 대해서 잘은 몰라도 들어 본 적은 있을 겁니다. 그리고 아마 그 대략적인 개념도 알고 있을 거라 믿습니다.

C 언어는, 혹자는 농담으로 알지만, B 언어라는 것을 개량한 고급 언어입니다. 그리고 이러한 C 언어로 작성된 소스코드는 컴파일러를 통해 어셈블리어로 바꿉니다. 그리고 어셈블러라는 것을 사용하면 최종적으로 이러한 어셈블리어는 인스트럭션의 집합인 바이너리(Binary) 코드로 바뀌게 됩니다.

인스트럭션은 CPU가 이해하는 유일한 언어이며, 0101 같은 2진수로 구성되어 있습니다. 이러한 인스트럭션에 1:1로 대응시켜 2진수를 사람들이 알아보기 쉬운 단어 형태로 바꾸어 놓은 것을 어셈블리(Assembly) 코드라고 합니다. 즉, 인스트럭션과 어셈블리 코드는 단순히 영어를 알아보기 힘든 필기체로 쓰느냐 활자체로 쓰느냐 정도의 차이밖에 없으며, 결국은 둘 다 CPU의 언어라 할 수 있죠(실제로는 어셈블리 코드에도 약간의 고급언어적인 측면이 있긴 합니다).



[그림 4-17] 컴파일 과정

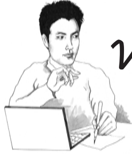
이제 우리가 여기서 알고자 하는 것은 이러한 인스트럭션이 도대체 무엇인가 하는 것입니다. 한국어가 알타이 어족이듯이 각 나라의 언어도 그 모체가 되는 말이 있으며, 최초의 언어라는 것은 나름대로 어떤 이유에 의해서 생겼을 겁니다. 마찬가지로 CPU의 언어라고 하는 인스트럭션은 도대체 무슨 기준으로 만들어진 언어일까요? 실제로 막상 소프트웨어만 다루는 엔지니어들은 이런 개념까지는 잘 모르고 있는 경우가 많이 있습니다만, 적어도 그 생성 원리 정도에 대해서는 알고 있어야 진정한 프로그래머라 할 수 있기 때문에 우리가 지금 이렇게 CPU에 대해 공부하고 있는 것입니다.

지금까지 우리가 다루었던 것이 CPU 전체는 아니더라도 상당히 큰 부분을 차지하고 실질적인 연산을 다루는 것이 ALU입니다. 이런 ALU에 연산하고자 하는 데이터가 연결되었다고 가정합시다. 가령 자판기의 현재 금액과 지금 새로 넣은 동전 금액의 합을 구하기 위해 두 금액이 각각 데이터 입력1과 2로 연결되었고 이 둘을 더한 결과를 자판기 금액 표시판에 나타내려 합니다. 이 때 더하기 연산을 하기 위해선 어떻게 해야 할까요?

바로 동작입력(Operation)에다가 덧셈 연산에 해당하는 값을 주는 것입니다. 처음 든 예처럼 건전지를 연결해서 신호를 주는 등 여러 가지 방법이 있을 것입니다. 다만 여기서는 그 방법보다, 입력값에 주목합시다. 그 입력값이 다름 아닌 인스트럭션인 것입니다. C언어로 컴파일을 하는 등의 고상한 과정이 아니라 애니악처럼 파이프를 뽑았다 꽂았다 하며 연결하는 것처럼 무식하게 하더라도, 어찌되었건 ALU로 행하고자 하는 연산을 선택하는 동작입력의 2진수 값이 바로 인스트럭션인 것이죠.

그렇다면 우리가 위에서 만든 ALU는 2비트의 동작입력이 있으므로 00_2, 01_2, 10_2, 11_2이라는 4가지 인스트럭션을 지니는 것일까요? 아닙니다. 가장 마지막에 추가한 뺄셈, 즉 SUB 연산의 경우 동작입력은 ADD와 똑같이 11_2이 되지만, b 입력을 반전하기 위한 MUX의 동작입력이 1이 되어야 합니다. 마찬가지로 나머지 4개의 연산은 반전되지 않은 b 입력을 받아야 하므로 b 입력 MUX에 0의 동작입력을 줘야 합니다. 결과적으로 외부에서 전체 ALU를 하나의 모듈로 보았을 때 b 입력 MUX의 동작입력을 최상위 비트로 본다면 AND는 (0 00), OR은 (0 01), XOR은 (0 10), ADD 는 (0 11), 그리고 SUB는 (1 11) 이렇게 총 5종류의 동작입력, 즉 인스트럭션을 갖는 것입니다.

CPU에 대해 좀 아는 사람은 지금까지의 얘기를 듣고 엉터리라고 생각할지도 모릅니다. 왜냐하면 실제로 인스트럭션을 구성하는 요소는 크게 세가지로 위와 동작입력과 같은 컨트롤 명령어와 실제로 연산을 하기 위한 두 값(오퍼랜드(Operand)라고 합니다)으로 이루어져 있기 때문입니다. 앞의 자판기 예에서는 두 입력값이 자판기 투입 금액과 현재 금액이라고 못을 박았기 때문에 인스트럭션을 이루는 부분은 동작입력의 3비트 뿐인 것이죠. 하지만 여러분은 속는 셈치고 일단 여기까지는 인스트럭션이 바로 ALU의 동작입력에 해당하는 부분이라고 생각해 두십시오. 앞으로 이 간단한 인스트럭션들을 점점 확장해가게 될 테니까요.



Vitamin Info

ALU의 NOT 연산 설계

이번 장에서 언급한 ALU에서는 논리 연산 중 NOT에 대한 언급이 없습니다. 가장 흔히 쓰이고 필수적인 연산 중 하나가 바로 NOT, 즉 비트 반전 연산인데 이를 구현하려면 위 ALU를 다시 설계해야 할까요?

정답은 아니오입니다. 물론 NOT 기능을 추가해서 설계해도 전혀 상관은 없습니다만, 기존 ALU에서 데이터 입력 중 한 곳에는 반전하고자 하는 데이터를 연결한 다음 나머지 한 쪽 입력에는 상수 1을 주고 XOR 연산을 하면 결과는 반전된 비트가 됩니다. 한 번 노트에 차곡차곡 진리표를 적어보세요.

입력	1과 XOR 결과
0	1
1	0

이것만은 알고 갑시다.

1. 전자 회로를 크게 아날로그 회로와 디지털 회로로 나눌 수 있다. 디지털 회로는 다른 말로 논리회로라고 할 수 있으며 이를 다시 세분화하면 조합 논리회로와 순차 논리회로로 나눌 수 있게 된다. 그렇다면 대표적인 디지털 회로라 할 수 있는 CPU도 논리회로일까? 그렇다면 조합 논리회로인가 순차 논리회로인가?

2. CPU 를 구성하는데 필수적인 몇 가지 장치들이 있다. 이 장치 들은 조합 논리회로와 순차 논리회로로 이루어져 있다. 이 중 연산을 담당하는 핵심 구성 요소는 무엇일까? 또한 이 장치는 조합 논리회로일까 순차 논리회로일까?

3. 컴퓨터가 2진수 체계를 채용한 데는 몇 가지 이유가 있다. 가장 대표적인 이유가 사람의 논리성이 흑백 논리의 2분법적이라는 것과 전기적으로 구분하기가 용이하다는 점에서이다.이러한 이진수를 채택하면서 대두되는 문제점 중 하나가 바로 음수의 표현인데 몇 가지 음수 표현 방법이 있으며 가장 대표적인 것이 2의 보수 혹은 1의 보수 방식이다. 현대의 컴퓨터는 대부분 2의 보수 방식을 채택하고 있는데 어떤 이유에서일까? 대표적인 이유 두 가지를 말하라.