Chapter 16

메모리를 만들어 봅시다

매일 아침에 일어날 때마다 텅 빈 머리에 기억이 채워지기 시작합니다. 어디서 있었는지, 어제 무엇을 했는지, 오늘 무엇을 하기로 했는지 기억이 나기 시작하지요. 이러한 기억들은 번뜩 떠오르기도 하고, 흐릿하게 떠오르기도 합니다. 그리고 몇 분이 지나면 일상적이지 않은 기억들에 대해서 "이런, 어제 양말을 신은 채로 잠자리에 들었군."과 같은 변명을 늘어놓기도 하지요. 하지만 대부분의 경우에는 일상이 평소와 대부분 비슷하여 연속성 있는 삶이 이루어집니다.

물론 사람의 기억이란 시간 순서대로 떠오르는 것은 아닙니다. 고등학교 때의 기하학 시간을 기억해 보면, 증명 종료(QED)가 어떤 것을 의미하는지 설명하시던 선생님의 모습보다는 앞자리에 앉았던 친구나 화재 대피 훈련을 하던 날이 먼저 떠오를 것입니다.

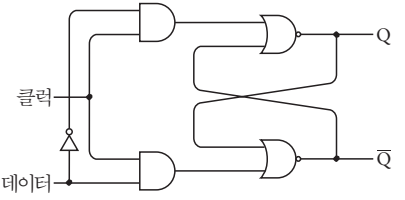
게다가 사람의 기억이란 것이 그다지 믿을 만한 것도 아닙니다. 글쓰기란

아마도 사람에게 부족한 기억력을 채워주기 위하여 개발되었을 것입니다. 지난밤 새벽 3시에 엄청난 영화 시나리오 아이디어가 떠올라서 갑작스럽게 일어났습니다. 이런 순간을 대비해서 침대 머리맡에 두었던 종이와 연필을 잡고서, 이 엄청난 아이디어를 잊어버리지 않기 위해서 허겁지겁 적어둡니다. 그 다음날 아침에 일어나서 그 멋진 아이디어를 읽고서 각본을 작성하기 시작하거나, ('소년이 소녀를 만나서, 차 추격전 하고, 폭발'. 뭐 이런 내용을 적었다면) 그렇지 않을 수도 있지요.

우리는 종종 글을 적어두고 나중에 읽어봅니다. 우리는 저축을 했다가 나중에 인출하기도 합니다. 우리는 어떤 것을 저장해두고 나중에 접근해서 살펴보기도 합니다. 메모리의 기능은 바로 위에 이야기한 두 작업의 시점을 온전히 이어주기 위해서 정보를 보관해 두는 것입니다. 우리가 정보를 저장하려고 할 때 우리는 다양한 형태의 메모리를 사용합니다. 종이는 글로 이루어진 정보를 저장하기에 적당하고, 자기 테이프는 음악이나 영화를 저장하는 데 유용하게 사용됩니다.

전신(telegraph)에서 사용되었던 릴레이도 논리 게이트 형태로 묶여서 플립플롭을 형성함으로써 정보를 저장할 수 있습니다. 앞에서 보았듯이 플립플롭은 1비트를 저장할 수 있지요. 그다지 많은 정보라 할 수는 없지만 단지 시작일 뿐입니다. 한 비트를 저장하는 방법을 알았으니 이제 둘, 셋 혹은 그 이상을 저장하는 것도 어렵지 않겠지요.

14장에서는 인버터 하나, AND 게이트 두 개와 NOR 게이트 두 개로 이루어진 레벨 트리거 D타입 플립플롭을 보았습니다.

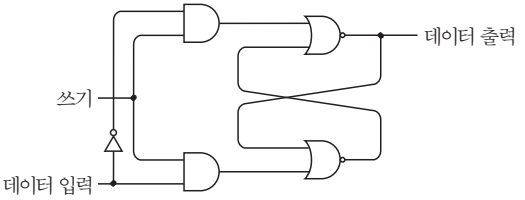


클럭 입력이 1인 동안에는 Q의 출력이 데이터 입력과 같아지게 됩니다. 하지만 클럭 입력이 0으로 바뀌면 Q는 직전의 데이터 입력 값을 유지하게 되는 것이지요. 그 이후에 데이터 입력이 바뀌는 것은 클럭이 다시 1의 값으로 바뀔 때까지 출력에 전혀 영향을 주지 못합니다. 플립플롭의 논리 표는 다음과 같습니다.

입력		출력	
D	Clk	Q	$\bar{Q}$
0	1	0	1
1	1	1	0
X	0	Q	$\bar{Q}$

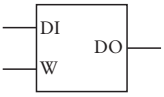
14장에서는 여러 가지 다른 형태의 회로를 이용하여 플립플롭의 여러 특성에 대하여 이야기하였지만, 이 장에서는 1비트 정보를 저장하는 것에 대해서만 이야기하도록 하겠습니다.

정보 저장이라는 목적이 좀 더 강조될 수 있도록 입력과 출력의 이름을 다음과 같이 바꾸어보도록 하겠습니다.

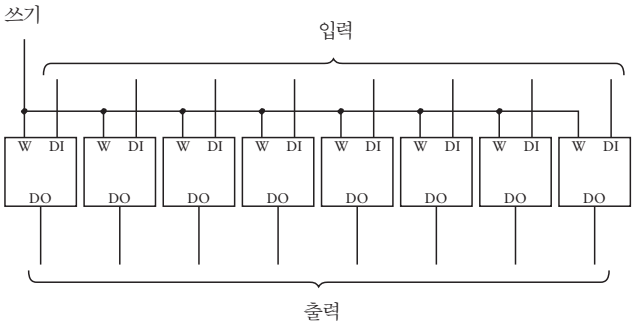


이는 플립플롭과 같은 것이지만, Q 출력 대신 데이터 출력이라는 이름을, 클럭 입력(14장의 앞부분에서는 '값 보존(Hold That Bit)'이라는 이름을 썼었지요)이 '쓰기(Write)'라는 이름으로 바뀐 것이지요. 종이에 정보를 적어두는 것과 마찬가지로 쓰기 신호는 데이터 입력이 회로에 저장될 수 있도록 만드는 것입니다. 보통 쓰기 신호는 0의 값을 가지고 있으므로 입력의 변화가 출력에 대하여 영향을 끼치지 못합니다. 입력 값을 플립플롭에 저장하기 위해서는 쓰기

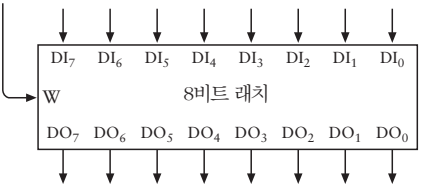
입력을 1로 바꾸었다 다시 0으로 변경하면 됩니다. 14장에서 잠깐 언급했지만 이런 형태의 회로의 경우 데이터 값의 변화를 막을 수 있기(latching) 때문에 래치라 부릅니다. 래치 역시 아래와 같이 각각의 구성요소를 자세히 그리지 않고 하나의 기호로 표현할 수 있습니다.



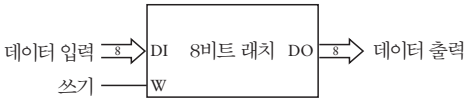
1비트짜리 래치를 여러 개 묶어서 여러 비트를 처리할 수 있는 래치를 만드는 것은 아주 쉽습니다. 아래 그림처럼 쓰기 신호만 같이 잘 묶으면 되지요.



위의 8비트 래치는 8개의 입력과 8개의 출력을 가지고 있으며, '쓰기'라는 이름의 입력을 가지고 있는데 이 값은 보통 '0'이 됩니다. 8비트 값을 이 래치에 저장하려면 쓰기 입력을 '1'로 바꾸었다 다시 '0'으로 변경하면 됩니다. 이 8비트 래치 역시도 다음과 같이 하나의 상자 형태로 그릴 수 있습니다.



물론 다음과 같이 1비트 래치와 좀 더 비슷하게 그릴 수도 있지요.



8개의 1비트 래치를 연결하는 다른 방법에 대해서도 소개하겠습니다. 이 방법은 앞에서 본 방법만큼 쉽지는 않습니다. 하나의 데이터 입력과 데이터 출력만을 필요로 한다고 가정해 봅시다. 하지만 한 신호가 서로 다른 8개의 시점(하루 내에서든, 며칠에 걸쳐서든)에 어떤 값을 가졌는지 저장하고 싶은 경우가 있을 것입니다. 또한 나중에 값을 확인해 볼 때도 하나의 데이터 출력 신호만을 살펴보면서 8개의 값을 확인하고 싶습니다.

다시 말하자면 하나의 8비트 값을 8비트 래치에 저장하는 것이 아니라, 별개의 의미를 가진 8개의 1비트 값들을 하나씩 래치에 저장하고 싶은 것이지요.

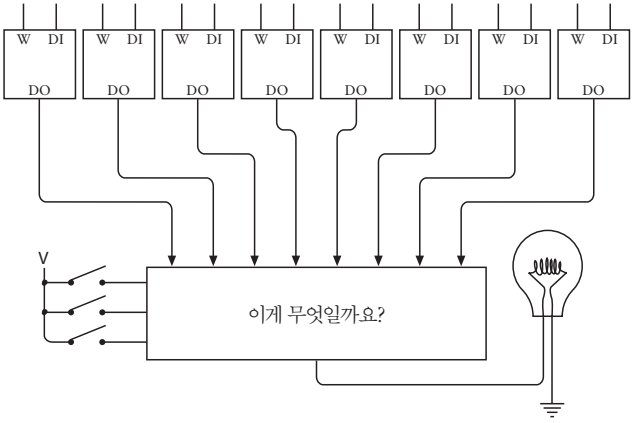
이렇게 해야 할 일이 뭐가 있을까요? 글쎄요. 아마도 전구가 하나밖에 없을 때도 이렇게 해야겠지요.

8개의 1비트 래치가 필요하다는 것은 이미 알고 있습니다. 지금은 어떻게 래치에 데이터가 저장되는지에 대한 부분은 신경 쓰지 말고 잠시 넘어가고, 8개의 래치에서 나오는 데이터 출력 신호가 어떤 방식으로 하나의 전구에 연결될 것인지 집중해보도록 합시다. 물론 전구의 위치를 손으로 하나씩 바꾸어가면서 각각의 래치에 어떤 값이 저장되어 있는지 확인하는 것도 가능하지만, 뭔가 좀 더 자동화된 방법이 좋겠지요. 사실 8개의 1비트 래치들 중에서 어떤 래치의 값을 전구로 보고 싶은지 정도는 선택할 수 있도록 스위치를 사용하는 것이 좋을 것 같습니다.

얼마나 많은 스위치가 필요할까요? 8개의 입력으로부터 선택을 하려고 한다면 세 개의 스위치가 필요합니다. 세 개의 스위치는 8개의 서로 다른 값

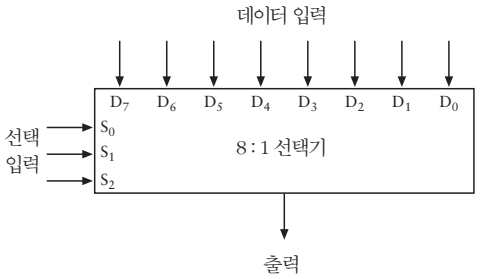
인 000, 001, 010, 011, 100, 101, 110, 111을 나타낼 수 있기 때문이지요.

아래와 같이 1비트 래치 8개, 스위치 세 개, 전구 한 개, 그리고 전구와 스위치 사이에 어떤 것이 하나 들어가야 할 것 같습니다.



앞에서 이야기한 '무엇'은 위쪽으로 8개의 입력을 받아들이고 왼쪽에서는 3개의 입력을 받아들입니다. 세 개의 스위치를 열거나 닫음으로써, 상자의 아래쪽에 있는 출력으로 8개의 입력 중에 어떤 값이 출력과 연결될 것인지 선택할 수 있게 되는 것이지요. 이 출력은 전구와 연결되어 전구를 켤 수 있습니다.

그럼 '이게 무엇일까요?' 블록이 정확하게 어떤 것일까요? 지금처럼 입력이 많지는 않았지만, 이와 아주 비슷한 회로를 이미 살펴본 적이 있습니다. 이 '무엇'은 14장에서 첫 번째로 새롭게 바꾼 덧셈기에서 우리가 살펴본 회로와 비슷합니다. 그때는 입력 값을 지정하기 위하여 사용된 스위치들과 래치의 출력 중에서 덧셈기로 들어가는 입력을 선택하기 위하여 비슷한 회로가 사용되었지요. 또한, 그때는 이를 2 : 1 선택기(2-Line-to-1-Line Selector)[1]라 불렀습니다. 여기서는 8 : 1 선택기(8-Line-to-1-Line Data Selector)[2]가 필요하지요.



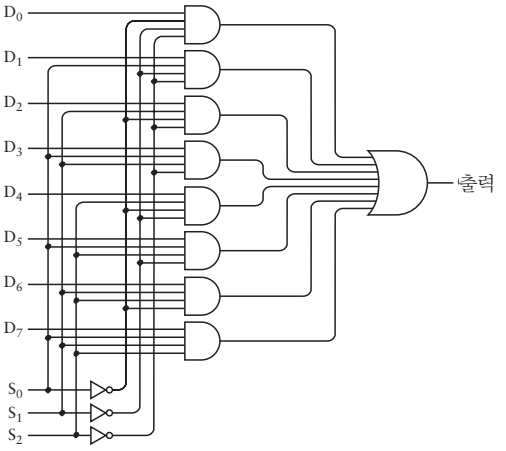
8 : 1 선택기는 8개의 데이터 입력(위쪽에 보이지요)과 3개의 선택(Select) 입력(왼쪽에 있지요)을 가지고 있습니다. 이 선택 입력은 어떤 입력이 출력에 나타날 것인지 선택하는 역할을 합니다. 예를 들어, 선택 입력이 '000'이라면, 출력은 D_0 입력과 동일하게 되며, 선택 입력이 111이라면, 출력은 D_7과 같아지고, 선택 입력이 101이라면 출력은 D_5의 값이 되는 것이지요. 다음은 이 회로의 논리표입니다.

입력			출력
S_2	S_1	S_0	Q
0	0	0	D_0
0	0	1	D_1
0	1	0	D_2
0	1	1	D_3
1	0	0	D_4
1	0	1	D_5
1	1	0	D_6
1	1	1	D_7

8 : 1 선택기는 다음 그림과 같이 인버터 3개, 4입력 AND 게이트 8개, 8입력 OR 게이트 1개로 이루어져 있습니다.

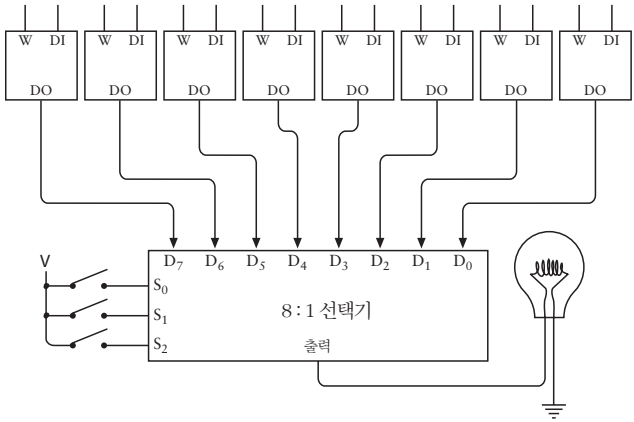
1 (옮긴이) 이러한 회로는 보통은 2 : 1 선택기보다는 2 : 1 MUX라는 이름이 더 널리 사용된다는 이야기도 했었습니다.

2 (옮긴이) 마찬가지로 8 : 1 MUX라는 표현이 더 자주 사용됩니다.



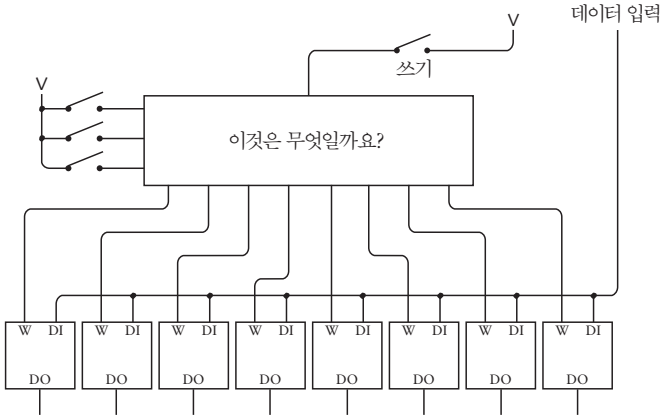
상당히 복잡하게 생긴 회로지만, 하나의 예제만 살펴보면 어떻게 동작하는지 쉽게 알 수 있을 것입니다. S_2가 1이고, S_1은 0, S_0는 1이라고 가정해 봅시다. 위에서 여섯 번째에 있는 AND 게이트가 S_0, $\bar{S}_1$, S^2의 입력과 연결되어 있으므로 모든 입력이 1이 됩니다. 다른 모든 AND 게이트의 입력은 좀 전에 설명한 값에서 입력 값이 1이 되는 경우가 없으므로 0을 출력하게 됩니다. 따라서 위에서 여섯 번째에 있는 AND 게이트와 연결된 D_5의 입력이 1이면 출력이 1이 되고, 이 값이 0이 되면 출력도 0이 됩니다. 이 출력 값은 오른쪽에 있는 OR 게이트에서도 마찬가지지요. 결과적으로 선택 입력이 101이 되는 경우에 출력은 D_5의 값과 같아지는 것입니다.

이제 원래 하려고 했던 일로 돌아와 봅시다. 처음에 의도했던 것은 8개의 1비트 래치들을 어떤 방식으로 연결하면 한 비트의 데이터 입력을 각각의 래치로 저장시킬 수 있는지 살펴보고, 어떤 방식을 이용하면 한 비트 데이터 출력으로 이를 꺼내볼 수 있는지 살펴보는 것이었지요. 일단 앞에서 8개의 1비트 래치의 출력들 중에 한 값을 선택하여 한 비트의 데이터 출력 신호에 보낼 수 있는 8 : 1 선택기를 살펴보았습니다.



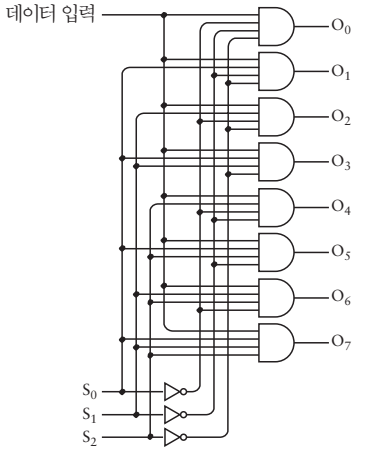
이제 절반이 끝난 것이지요. 출력 부분을 만들었으니 이제 입력 부분에 대해서 살펴보도록 합시다.

입력 부분에는 데이터 입력 신호와 쓰기 입력 신호가 있었습니다. 래치의 입력 부분에 있는 데이터 입력은 모두 한꺼번에 연결할 수 있습니다. 하지만 각 래치에는 서로 다른 값을 써야 하기 때문에 8개의 쓰기 입력은 같이 연결하면 안 되겠지요. 즉, 한 비트 쓰기 신호가 오직 하나의 래치에만 전달될 수 있도록 만들어야 합니다.



이런 작업을 수행하기 위해서는 앞에서 살펴본 8 : 1 선택기와 어떤 면에서는 비슷하지만 그 반대의 동작을 수행할 수 있는 회로가 필요합니다. 이를 3-8 디코더(3-to-8 Decoder)라 합니다.[3] 이와 비슷한 간단한 형태의 디코더를 이미 앞에서 살펴본 적이 있습니다. 11장에서 이상적인 색을 가진 고양이를 선택하기 위해서 스위치들을 연결했을 때 나왔었지요.

3-8 디코더는 8개의 출력을 가지고 있습니다. 이 회로는 하나의 출력을 제외하면 모두 0의 값을 출력하는 특성을 가지고 있습니다. 어떤 출력 신호가 특정한 값을 가질 것인지는 S_0, S_1, S_2에 의하여 선택되며, 선택된 출력의 값은 데이터 입력의 값과 같은 값을 지니게 됩니다.



앞의 경우와 마찬가지로 위에서 6번째 AND 게이트만이 S_0, $\overline{S}_1$, S_2에 해당하는 입력에 대응되며, 다른 AND 게이트들은 이 입력에 대응되지 않습니다. 따라서 선택 입력의 값이 101이라면 다른 AND 게이트의 출력은 무조건 0이 됩

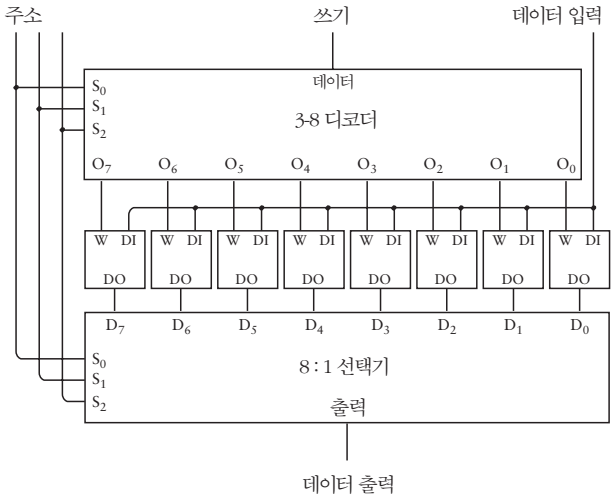
3 (옮긴이) 이런 회로는 사실 1비트 입력을 8곳의 출력에 배분하는 역할을 수행하므로 1-line to 8 line demultiplexer, 짧게 1 : 8 DEMUX라 부르는 것이 일반적입니다. 앞에서 살펴본 8 : 1 선택기(좀 더 자주 사용되는 표현은 8 : 1 MUX)의 반대가 되는 것이지요.

니다.

반면에 위에서 6번째에 있는 AND 게이트의 출력은 데이터 입력에 따라 0이 될 수도 1이 될 수도 있습니다. 다음은 이 회로의 완전한 논리표입니다.

입력			출력							
S_2	S_1	S_0	O_7	O_6	O_5	O_4	O_4	O_2	O_1	O_0
0	0	0	0	0	0	0	0	0	0	데이터
0	0	1	0	0	0	0	0	0	데이터	0
0	1	0	0	0	0	0	0	데이터	0	0
0	1	1	0	0	0	0	데이터	0	0	0
1	0	0	0	0	0	데이터	0	0	0	0
1	0	1	0	0	데이터	0	0	0	0	0
1	1	0	0	데이터	0	0	0	0	0	0
1	1	1	데이터	0	0	0	0	0	0	0

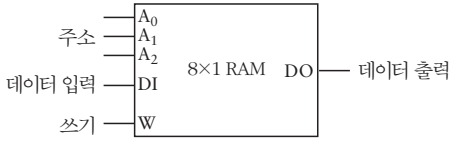
다음은 8개의 래치를 포함하고 있는 완전한 회로를 보여주고 있습니다.



세 비트의 선택 신호가 디코더와 선택기에 같이 들어가고 있으며, 이 신호에 주소(Address)라는 이름이 붙어 있다는 부분에 관심을 가질 필요가 있습

니다. 우편번호와 비슷하게 이 3비트 주소는 8개의 1비트 래치들 중에서 어떤 래치에 접근할 것인지 알려주는 역할을 합니다.[4] 주소는 어떤 래치에 데이터 입력 값이 저장되도록 쓰기 신호를 변경시킬지 판단하기 위하여 입력으로 사용됩니다. 그림 아랫부분에 있는 출력들에서 출력은 8 : 1 선택기를 조작하여 8개 래치의 출력 중에 한 값을 선택할 수 있도록 하기 위하여 사용됩니다.

이러한 형태를 가진 래치는 종종 읽기/쓰기 메모리, 좀 더 일반적으로는 임의 접근 메모리(Random Access Memory), 즉 RAM이라는 이름으로 더 많이 알려져 있습니다. 여기서 설명한 RAM의 구성은 개개의 1비트 값 8개를 저장할 수 있습니다. 이는 다음과 같이 표현할 수 있습니다.



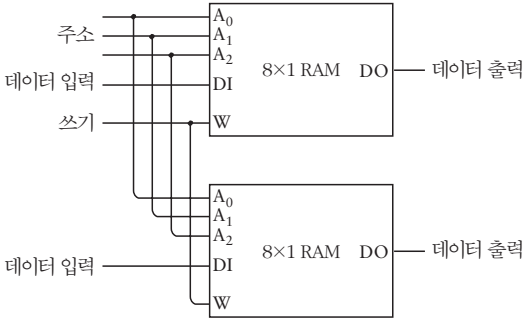
위의 회로는 정보를 보존할 수 있으므로 메모리라 불립니다. 새로운 값을 각각의 래치에 저장할 수 있고(즉, 값을 래치에 적고), 어떤 값이 저장되어 있는지 확인할 수 있기 때문에(즉, 값을 래치에서 읽을 수 있죠) 읽고/쓸 수 있는 메모리라 부릅니다. 주소 값만 바꾸면 8개의 래치 중 어디서든 값을 읽고 쓸 수 있기 때문에 임의 접근 메모리(random access memory)라 불립니다. 이와 반대로 몇몇 메모리는 값을 순차적으로 읽어야 합니다. 이러한 형태의 메모리에서 주소 101번지에 저장된 값을 읽기 위해서는 100번지의 값부터 읽어야만 합니다.[5]

4 (옮긴이) 보통 메모리 주소 값을 이야기할 때 보통 xxx번지라고 이야기하는 것도 비슷하지요.

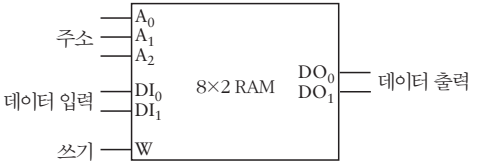
5 (옮긴이) 좀 오래된 예이기는 하지만, 카세트테이프를 생각해 보세요. 참고로 이런 속성을 가진 메모리 장치를 선형 메모리 장치(linear memory device)라 부르기도 합니다.

앞에서 설명한 RAM의 구성을 종종 RAM 배열(array)이라 부릅니다. 이러한 형태로 구성된 RAM 배열을 줄여서 8×1('eight by one'[6]이라 발음하죠) 형태라 이야기합니다. 배열에 8개의 값이 존재할 수 있으며 이는 각각 1비트라는 의미지요. 또한 두 값을 곱한 결과는 이 메모리 배열에 저장될 수 있는 비트 수를 의미합니다.

RAM 배열은 다양한 방법으로 조합될 수 있습니다. 예를 들어, 다음과 같이 두 개의 8×1 RAM 배열에 같은 주소가 공급되도록 배치할 수도 있지요.



주소와 쓰기 입력이 두 개의 8×1 RAM 배열에 공통적으로 연결되어 있으므로, 결과적으로 8×2 RAM 배열의 형태가 됩니다.

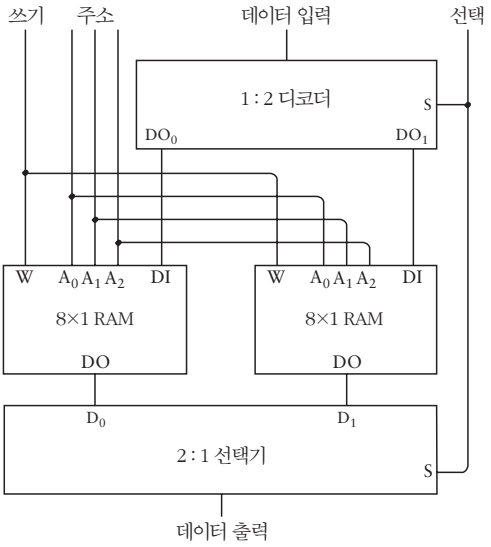


이 RAM 배열은 8개의 값을 저장한다는 점은 동일하지만 각각의 값이 2비트 크기를 가진다는 점이 다릅니다.

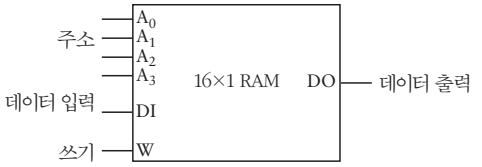
다음 그림과 같이 두 개의 8×1 RAM 배열을, 8비트 메모리에서 각각의 래

6 (옮긴이) 우리나라에서는 '팔 곱하기 일'이라 부릅니다.

치를 묶었을 때와 같은 방식으로 2 : 1 선택기와 1 : 2 디코더를 사용해서 묶는 방법도 있습니다.



선택 입력은 디코더와 선택기 모두에 전달되어 두 개의 8×1 RAM 배열 중에 한 개를 선택합니다. 이는 사실상 네 번째 주소 입력 비트와 같은 의미가 되므로 이를 통하여 16×1 RAM 배열이 만들어지는 것이지요.



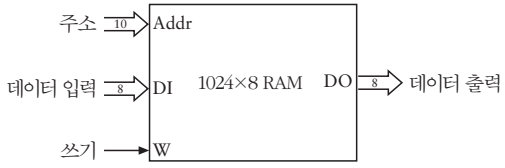
이 RAM 배열은 1비트 값 16개를 저장할 수 있습니다.

RAM 배열에 저장할 수 있는 값의 수는 주소 입력의 비트 수와 직접적인 연관성을 가지고 있습니다. 주소 입력이 없다면(1비트 래치와 8비트 래치가 이런 모양이지요) 하나의 값만이 저장될 수 있습니다. 주소 입력이 한 비트라면 두

개의 값을 저장하는 것이 가능하지요. 주소 입력이 두 비트라면 네 개의 값을 저장할 수 있습니다. 마찬가지로 주소 입력이 세 비트라면 8개의 값을, 네 비트라면 16개의 값을 저장할 수 있는 것이지요. 이러한 연관 관계는 다음 식으로 요약해 볼 수 있습니다.

$$\text{RAM 배열에 저장되는 값의 수} = 2^{\text{주소 입력의 비트 수}}$$

지금까지 작은 RAM 배열이 어떤 방식으로 만들어지는지를 보여드렸으므로, 큰 메모리를 만드는 방법을 상상하는 일도 어렵지 않을 것입니다. 예를 들자면 아래와 같은 메모리가 있습니다.



이 RAM 배열은 8비트 값을 모두 1024개 저장할 수 있도록 구성되어 있으므로 총 8196비트를 저장할 수 있지요. 이 메모리에는 10비트 주소 입력이 있으므로 총 2^{10}인 1024개의 값을 저장할 수 있습니다. 또한 8비트 데이터 입력과 8비트 데이터 출력이 각각 존재합니다.

다른 말로 하자면, RAM 배열은 1024바이트를 저장할 수 있습니다. 우체국으로 이야기하자면 1024개의 사서함이 있으며, 각각의 사서함에는 서로 다른 1바이트 값이 들어있다고 생각해 볼 수 있지요.

1024바이트는 킬로바이트(kilobyte)라 알려져 있는데 이 말이 가끔 혼동을 일으킵니다. kilo라는 접두사(그리스어로 1000을 의미하는 khillioi에서 유래되었습니다)는 대부분의 도량형에서 사용됩니다. 예를 들어 킬로그램(kilogram)은 1000그램을 의미하고, 킬로미터(kilometer)는 1000미터를 의미하지요. 하지만 앞에

서 이야기한 바와 같이 킬로바이트(kilobyte)는 1000바이트가 아닌 1024바이트를 의미합니다.

이러한 문제는 일반적인 도량형의 경우 10의 거듭제곱수에 기반하고 있는 반면에 이진수는 2의 거듭제곱수에 기반하고 있으며, 두 가지 수체계 간에 일치되는 값이 없다는 것에 기인합니다. 쉽게 이야기하면 10의 거듭제곱수는 10, 100, 1000, 10000, 100000 등인 반면에 2의 거듭제곱수는 2, 4, 8, 16, 32, 64 등이 되지요. 또한 10의 정수 거듭제곱수와 2의 정수 거듭제곱수 중에 같은 수는 없습니다.

하지만 두 수가 아주 근접하는 시점이 있기는 합니다. 생각하시는 것처럼 1000은 1024와 아주 가깝지요. 이 경우에 대해서 근사 등호 기호를 사용하여 다음과 같이 표현할 수 있습니다.

$$2^{10} \approx 10^{3}$$

이 관계에 마법과 같은 부분은 없습니다. 단지 2의 거듭제곱수 중 어떤 것은 10의 거듭제곱수 중 어떤 수에 근사 등가(approximately equal)가 된다는 것이지요. 이러한 특징에 의하여 사람들이 실제로는 1024바이트를 표현하고 싶을 때 그냥 편하게 킬로바이트라는 표현을 사용하는 것이지요.

킬로바이트는 K 또는 KB라는 약자로 사용되기도 합니다. 앞에서 보여드린 RAM 배열은 1024바이트 혹은 1킬로바이트, 1K, 1KB를 저장할 수 있다고 말할 수 있습니다.

1KB RAM 배열이 1000바이트 혹은 천 바이트를 저장한다고 이야기하면 안 된다는 점을 잊지 마십시오. 1000을 넘어가는 값인 1024를 의미하는 것이지요. 어떤 것을 이야기하고 있는 것인지 정확히 알고 있다면 1K 혹은 1킬로바이트라고 이야기하셔도 됩니다.[7]

1킬로바이트의 메모리는 8비트 데이터 입력과 8비트 데이터 출력, 10비트

주소 입력을 가지고 있습니다. 각각의 바이트는 10비트 주소 입력에 의하여 접근되므로 RAM 배열은 2^{10}바이트를 저장할 수 있습니다. 주소 입력이 한 비트 증가할 때마다 메모리의 양은 두 배 증가하게 됩니다. 다음의 예는 각 행마다 메모리가 두 배로 증가하는 것을 보여줍니다.

1킬로바이트 = 1,024바이트 = 2^{10}바이트 ≈ 10^3바이트
2킬로바이트 = 2,048바이트 = 2^{11}바이트
4킬로바이트 = 4,096바이트 = 2^{12}바이트
8킬로바이트 = 8,192바이트 = 2^{13}바이트
16킬로바이트 = 16,384바이트 = 2^{14}바이트
32킬로바이트 = 32,768바이트 = 2^{15}바이트
64킬로바이트 = 65,536바이트 = 2^{16}바이트
128킬로바이트 = 131,072바이트 = 2^{17}바이트
256킬로바이트 = 262,144바이트 = 2^{18}바이트
512킬로바이트 = 524,288바이트 = 2^{19}바이트
1,024킬로바이트 = 1,048,576바이트 = 2^{20}바이트 ≈ 10^6바이트

위의 배열에서 가장 왼쪽에 있는 킬로바이트 단위의 수는 2의 거듭제곱수라는 것에 주목하십시오.

1024바이트가 킬로바이트로 불리는 것과 마찬가지로 1024킬로바이트는 메가바이트(megabyte, 그리스어로 mega는 거대하다는 의미입니다)라 불립니다. 메가바이트는 MB라는 약자로 쓸 수 있습니다.[8] 이제 메모리의 용량을 계속하여 두 배로 증가시켜 봅시다.

7 (옮긴이) 이러한 모호성을 줄이기 위해서, 최근에는 kilobyte대신에 kibibyte라는 단위를 사용하여 1024바이트라는 것을 명확히 하려는 시도가 진행 중입니다. 약자로도 KB가 아닌 KiB로 적습니다. 마찬가지로 Megabyte 대신에 Mebibyte(약자로는 MiB)와 같은 단위를 사용하지요. 이는 현재 IEC 국제 표준이기도 하며, Linux 시스템 등에서 폭넓게 받아들여지고 있습니다. 자세한 사항은 위키백과나 http://physics.nist.gov/cuu/Units/binary.html 등을 참고하십시오.

8 (옮긴이) 앞에 이야기한 새로운 표준에서는 Mebibyte라 부르고 MiB라 씁니다.

1메가바이트 = 1,048,576바이트 = 2^{20}바이트 ≈ 10^{6}바이트
2메가바이트 = 2,097,152바이트 = 2^{21}바이트
4메가바이트 = 4,194,304바이트 = 2^{22}바이트
8메가바이트 = 8,388,608바이트 = 2^{23}바이트
16메가바이트 = 16,777,216바이트 = 2^{24}바이트
32메가바이트 = 33,554,432바이트 = 2^{25}바이트
64메가바이트 = 67,108,864바이트 = 2^{26}바이트
128메가바이트 = 134,217,728바이트 = 2^{27}바이트
256메가바이트 = 268,435,456바이트 = 2^{28}바이트
512메가바이트 = 536,870,912바이트 = 2^{29}바이트
1,024메가바이트 = 1,073,741,824바이트 = 2^{30}바이트 ≈ 10^{9}바이트

그리스어로 gigas는 거인(giant)를 의미하며, 1024메가바이트(megabyte)는 기가바이트(gigabyte)라 불리며, GB라는 약자가 사용됩니다.[9]

이와 비슷하게 테라바이트(terabyte; teras는 괴물을 의미합니다)는 2^{40}바이트(10^{12}에 근사합니다), 즉 1,099,511,627,776바이트를 의미합니다. 테라바이트는 TB라는 약자를 사용합니다.[10]

킬로바이트는 천 바이트에 근사하며, 메가바이트는 백만 바이트에 근사하고, 기가바이트는 십억 바이트에 근사하며, 테라바이트는 1조 바이트에 근사하게 됩니다.

이보다 높은 숫자 영역은 잘 이용되지 않지만, 페타바이트(petabyte)는 2^{50}바이트, 즉 1,125,899,906,842,624바이트가 되며, 이는 10^{15}(1000조)에 근사합니다. 엑사바이트(exabyte)는 2^{60} 즉 1,152,921,504,606,846,976바이트이며, 10^{18}(100경)에 근사합니다.

약간 감을 가지기 위해서 예를 들자면, 제가 이 책을 처음 집필했던 1999년에 일반적인 가정용 컴퓨터에 설치되어 있는 RAM의 크기는 대략 32MB,

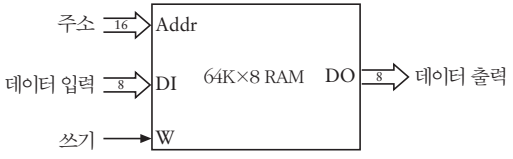
9 (옮긴이) 앞에 이야기한 새로운 표준에서는 Gibibyte라 부르고 GiB이라 씁니다.
10 (옮긴이) 앞에 이야기한 새로운 표준에서는 Tebibyte라 부르고 TiB라 씁니다.

64MB, 128MB 정도였습니다. (하드 드라이브의 용량을 이야기하는 것은 아니고 RAM에 대해서만 이야기 할 것이므로 그다지 혼동되지는 않을 것입니다.) 이는 각각 33,554,432바이트, 67,108,864바이트, 134,217,728바이트지요.

물론 사람들은 짧게 줄여서 말하기를 좋아합니다. 예를 들어, 65,536바이트 메모리를 가진 사람은 "64K를 가지고 있다"(1980년대에서 왔나보군요)고 이야기합니다. 또한, 33,554,432바이트 메모리를 가진 사람은 "난 32메가를 가지고 있어"라고 하고, 1,073,741,824바이트 메모리를 가진 사람은 "난 1기가를 가지고 있어"라고 이야기하겠지요.

가끔 사람들이 킬로비트나 메가비트(바이트가 아니라 비트라는 데 주의하세요)를 이야기할 때가 있기는 하지만, 흔치는 않습니다. 대부분 메모리에 대해서 이야기할 때는 비트가 아니라 바이트 단위로 이야기를 하지요. (물론 바이트에서 비트 단위로 바꾸는 것은 그냥 8만 곱하면 됩니다.) 킬로비트나 메가비트라는 단위가 대화에서 사용되는 경우는 대부분 데이터가 통신 회선을 통하여 전달될 때에 전송 속도의 단위로 '초당 킬로비트'나 '초당 메가비트'등의 값을 사용할 때일 것입니다. 예를 들어 56K 모뎀은 초당 56킬로비트의 데이터(킬로바이트가 아닙니다)를 전송할 수 있다는 것이지요.

이제 우리가 원하는 크기만큼 메모리를 만드는 방법을 알게 되었으니, 이 범주에서 더 이상은 너무 많이 나가지 않도록 하지요. 일단 아래와 같이 65,536바이트의 메모리를 구성했다고 가정해봅시다.

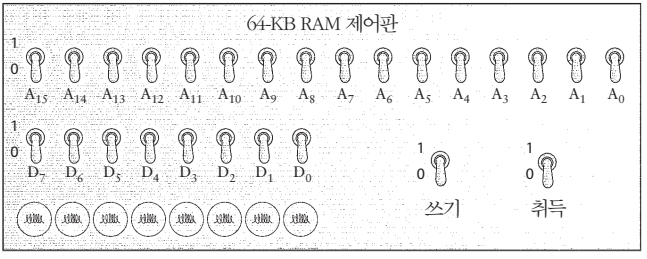


왜 32KB나 128KB가 아닌 64KB일까요? 65,536이 바로 2진수와 16진수 모두의 관점에서 딱 떨어지는 숫자이기 때문이지요. 이는 2^{16}이므로, 이 RAM

배열은 16비트 주소 입력을 가지고 있게 됩니다. 다른 말로 하자면 주소는 정확히 2바이트가 되고 이는 16진수로 주소의 범위가 0000h에서 ffffh가 됨을 의미합니다.

앞에서 은연중에 이야기했지만, 1980년 정도에 팔리던 개인용 컴퓨터의 메모리는 일반적으로 64KB였습니다. 물론 전신에서 사용했던 릴레이로 구성된 것은 아니지만 실제로 이런 메모리를 릴레이로 구성하지 못할 이유는 없습니다. 하지만 그 부분에 대해서 관심을 가지고 계실 것이라 생각하지 않습니다. 앞에서 만들었던 1비트 메모리를 만들기 위해서는 9개의 릴레이가 필요했으므로, 64K × 8 RAM 배열을 만들기 위해서는 대략 5백만 개의 릴레이가 필요합니다.[11]

메모리를 이용할 때 값을 메모리에 쓰고, 저장된 값을 읽을 수 있도록 관리해 줄 수 있는 제어판이 있다면 아주 도움이 될 것 같습니다. 이 제어판에는 다음 그림에서 표현한 것과 같이 주소를 입력하기 위한 16개의 스위치가 있으며, 메모리에 저장될 8비트 값을 입력하기 위한 8개의 스위치와 쓰기 동작을 제어하기 위한 스위치, 그리고 8비트 출력 값을 표시하기 위한 8개의

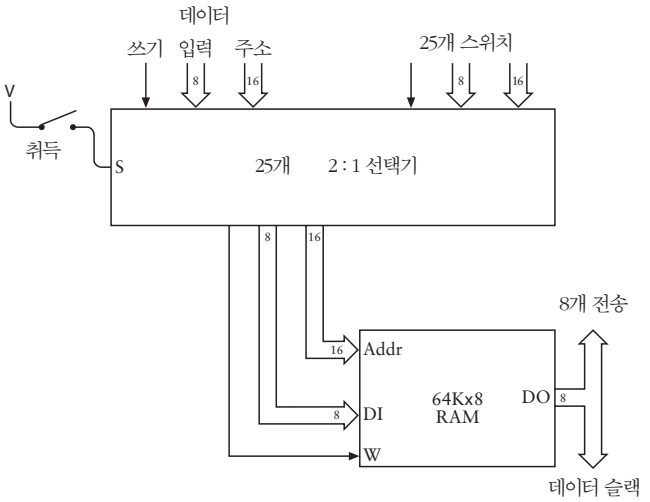


11 (옮긴이) 보통 사용되는 메모리는 속도와 복잡도에 따라 레지스터, 캐시 등에 많이 사용되는 SRAM, 보통 주 메모리로 사용되는 DRAM(요즘은 DDRn-SDRAM이지요)으로 구분되는데, 여기서 계속 나오고 있는 메모리인 래치는 레지스터에 가깝습니다(세월이 많이 지났으니 완전히 동일하지는 않습니다). 참고로 SRAM은 보통 4개 정도의 트랜지스터로 만들어지고(물론, 1T SRAM이라고 해서 트랜지스터 1개를 이용하는 특별한 형태도 있습니다만, 논외로 하고요), DRAM은 1개의 트랜지스터로 만들어집니다.

전구가 사용됩니다.

그림에 보이는 모든 스위치는 꺼짐(off, 0)의 위치에 있습니다. 또한 제어판에 취득(takeover)이라는 언급하지 않은 스위치 하나를 포함시켰습니다. 이 스위치는 다른 회로에서 제어판에 붙어 있는 메모리를 제어할 수 있도록 만들어 둔 것입니다. 이 스위치가 그림에서와 같이 '0'인 경우에는 제어판에 있는 모든 스위치가 동작하지 않게 됩니다. 이 스위치가 1인 경우에만 제어판은 메모리에 대한 제어권을 '취득'하게 되는 것이지요.

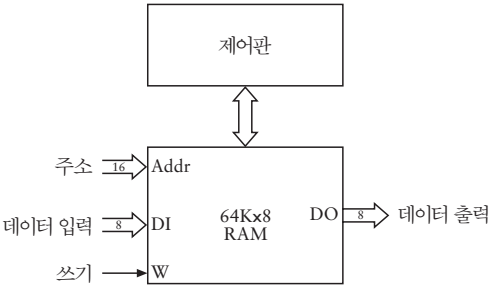
이 동작은 다수의 2 : 1 선택기로 이루어집니다. 2 : 1 선택기는 주소 신호 부분에 16개, 데이터 입력 스위치에 8개, 쓰기 스위치에 1개가 사용되므로 합쳐서 25개의 2 : 1 선택기가 사용됩니다. 아래에 대략적인 회로가 있습니다.



취득(Takeover) 스위치가 그림에서와 같이 열려 있는 경우에는, 64K × 8 RAM 배열로 들어가는 주소, 데이터, 쓰기 입력이 2 : 1 입력기의 좌상단에 있는 외부 입력 값으로 선택됩니다. 취득 스위치가 닫히는 경우에는 제어판의 스위치로부터 주소, 데이터, 쓰기 입력을 받아들이게 됩니다. 두 경우 모두

RAM 배열에서 출력되는 데이터 출력 신호는 8개의 전구로 전달되며, 다른 곳으로도 연결될 수 있습니다.

이러한 제어판이 있는 64K×8 RAM 배열은 다음과 같은 형태로 그릴 수 있습니다.



취득 스위치가 닫힌 경우에는 16개의 주소 스위치를 이용하여 65,536 주소들 중 어떤 곳이든 선택할 수 있습니다. 전구에는 해당 주소로 지정한 메모리에 지금 저장된 8비트 값이 출력될 것입니다. 8비트 데이터 입력 스위치는 새로운 값을 지정하기 위하여 사용할 수 있으며, 이 값은 쓰기 스위치를 이용하여 메모리에 저장될 수 있습니다.

64K×8 RAM 배열과 제어판을 이용하여 65,536개의 8비트 값을 언제든지 보존하고 꺼낼 수 있게 되었습니다. 하지만, 메모리에 저장된 값을 사용하고 다른 값을 메모리에 저장할 수 있도록 하기 위해서는, 다른 어떤 것(대부분의 경우에는 다른 회로가 되겠지요)과 연결할 수 있어야 한다는 점을 잊으면 안 됩니다.

메모리와 관련해서 잊지 말아야 할 아주 중요한 것이 하나 더 있습니다. 11장에서 논리 게이트의 개념을 소개하면서 더 이상 게이트를 구성하는 각각의 릴레이를 그리지는 않았었지요. 특히 각 릴레이들에 모두 공급되는 전원에 대해서는 더 이상 나타내거나 설명한 적이 없습니다. 릴레이의 값이 변

할 때마다 전자석에 말려 있는 코일을 통하여 전기가 흐르게 되고, 이것이 금속 접점을 원하는 자리에 머무르게 합니다.

만일 여러분이 좋아하시는 65,536개의 바이트 값이 가득 찬 64K×8 RAM 배열의 전원을 끄면 어떤 일이 벌어질까요? 모든 전자석이 자성을 잃으면서 큰 소리를 '텅'하고 내면서 게이트가 천이되지 않은 상태로 되돌아가게 됩니다. RAM에 있던 자료는 어떻게 될까요? 완전히 사라져 버리게 되는 것입니다.

이것이 바로 임의 접근 메모리(RAM)가 휘발성 메모리(volatile memory)라 불리는 이유입니다. 이러한 형태의 메모리는 내용을 보존하기 위하여 일정한 전기의 공급이 필요합니다.

Chapter 17

혼자서 움직이는 컴퓨터

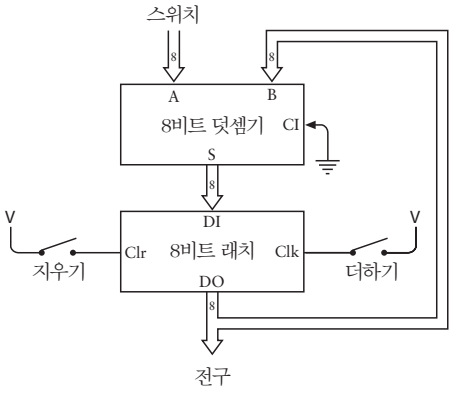
인류는 놀랄 만큼 창의적이고 부지런한 부분과 아주 게으른 부분을 같이 가지고 있습니다. 사람들이 일반적으로 일하는 것을 싫어한다는 것은 자명합니다. 일에 대한 반감이 가끔은 극단적인 형태로 나타나서 가끔 단 몇 분을 절약해 줄 수 있는 기계를 만들기 위해서 아주 많은 시간을 소모하기도 합니다. (물론, 뭔가를 만들어내는 재미가 있기는 하지요.) 단순히 즐거움 속에서 만족하면서 사는 것보다 새로 만든 괴상한 기계들이 잔디밭을 정리하는 것을 구경하면서 그물 침대에 편안히 누워있는 것을 꿈꿀 때도 있습니다.

이 장에서 자동으로 잔디밭을 정리해주는 기계를 만들기 위한 계획을 보여드리는 것은 아닙니다. 하지만, 이 장에서는 그 동안에 본 것보다 훨씬 복잡한 형태를 가진, 혼자서 덧셈과 뺄셈을 할 수 있는 기계를 만들어 볼 것입니다. 그렇게 대단한 기계를 만드는 것처럼 보이지 않는다는 사실은 잘 알고 있습니다. 하지만, 이 장에서 최종적으로 만들게 될 장치는 아주 많은 부분에

적용할 수 있기 때문에, 장치에서 기본적으로 지원하는 덧셈과 뺄셈을 사용하는 문제들뿐만 아니라 그보다 훨씬 더 많은 일을 처리할 수 있습니다.

물론, 정교하다는 것은 상당한 복잡도를 수반하므로 여기서는 많은 부분을 대략적으로만 설명할 것입니다. 세세한 부분은 대충 훑어보고 넘어가셔도 됩니다. 이 책의 내용에서 수학 문제를 풀기 위한 전자공학 혹은 기계공학의 배경 지식이 전혀 필요치 않다고 이야기하기는 쉽지 않습니다. 하지만, 조금 힘들더라도 믿고 끝까지 읽어나가면 이 장이 끝날 때에는 실제적으로 컴퓨터라 불릴 만한 것을 만들어 낼 수 있을 것입니다.

14장의 마지막 부분에서 덧셈기에 대해서 살펴보았습니다. 이때 보았던 덧셈기는 8개의 스위치로 입력된 숫자들을 이용하여 덧셈을 수행하고, 8비트 래치를 이용하여 그 중간 합을 보존합니다.



기억하시겠지만 8비트 래치는 8비트의 값을 저장하기 위하여 플립플롭을 사용했습니다.[1] 이 장치를 사용하기 전에, 그 내부에 저장된 값을 모두 0으로 만들기 위해 잠깐 동안 지우기(Clear) 버튼을 눌러야 합니다. 그 이후에

1 (옮긴이) 이 책에서의 표현과는 약간 차이가 있지만, 전자 엔지니어들은 보통 레벨 트리거나 엣지 트리거 속성에 의하여 래치와 플립플롭을 구분한다는 말씀도 드렸습니다.

스위치로 첫 번째 숫자를 입력하면, 덧셈기는 입력한 숫자에 래치에 있는 값인 0을 더하고, 그 결과 값이 전구로 출력됩니다. 이제 스위치를 이용하여 두 번째 숫자를 입력하고 더하기(Add) 버튼을 다시 한번 누르면, 그 합이 래치에 저장되고 결과는 전구에 나타나게 됩니다. 이러한 방식으로 일련의 숫자들을 더할 수 있으며, 중간 중간의 합은 전구를 통하여 확인할 수 있습니다. 물론, 이 덧셈기에는 전구가 8개 밖에 없기 때문에 255 이상의 수를 표시할 수 없다는 제한이 있기는 합니다.

14장에서 이 회로를 보여드렸을 때 레벨 트리거 속성을 가진 래치에 대해서만 이야기했지요. 레벨 트리거 래치에서 어떤 데이터를 저장하려면 클럭(Clock) 입력을 1로 만들었다 0으로 바꾸어야 합니다. 래치에서는 클럭 입력이 1인 동안에만 데이터 입력에서 일어나는 값의 변화가 출력 부분에 반영되지요. 그 후에 엣지 트리거 래치[2]에 대하여 다루었습니다. 이러한 형태의 래치는 클럭이 0에서 1로 변경되는 짧은 순간에 입력 값을 저장하게 됩니다. 엣지 트리거 래치가 좀 더 다루기 쉽기 때문에, 이 장에서 나오는 모든 래치는 엣지 트리거 속성을 가지고 있다고 가정하도록 하겠습니다.

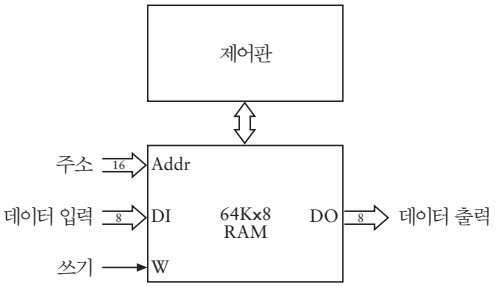
지금까지 더해진 숫자들의 중간 합을 보존하고 있는 래치를 보통 누산기(累算機, Accumulator)라 부릅니다. 이 장의 뒤에서 살펴보겠지만 누산기는 단순히 덧셈을 누적하는 것 이상의 역할을 하게 됩니다. 누산기는 보통 일련의 수를 더하거나 뺄 때 첫 번째 값을 저장하고 있는 래치를 의미합니다.

위에서 살펴본 덧셈기에는 좀 심각한 문제가 하나 있습니다. 100개의 이진수를 더해야 한다고 생각해 봅시다. 덧셈기 앞에 앉아서 묵묵히 각각의 숫자를 입력하고 그 합을 누적해 나가야겠지요. 만일 다 입력하고 난 후에야 숫자 몇 개를 잘못 입력했다는 것을 발견했다면, 별수 없이 처음부터 다시

2 (옮긴이) 일반적으로 플립플롭이라 불립니다.

숫자를 입력해야만 합니다.

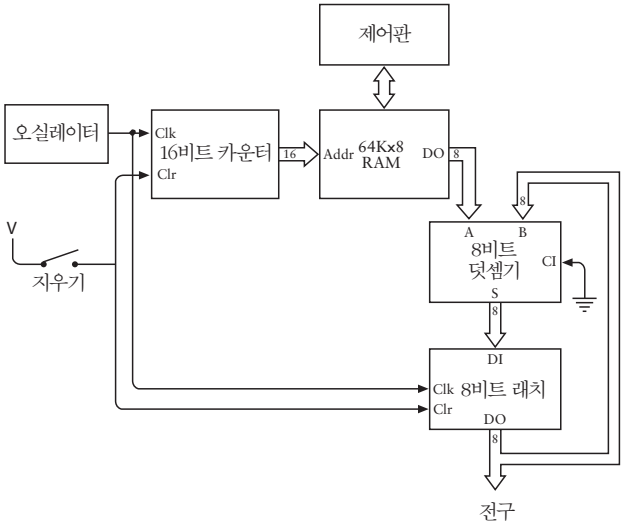
사실 그렇게 하지 않아도 됩니다. 앞에서 살펴보셨듯이 64KB 정도 되는 RAM 메모리를 만들려면 대략 5백만 개의 릴레이가 필요하지요. 또한, 메모리와 제어판을 연결한 후 취득(Takeover)이라는 이름의 스위치를 닫아서 RAM 배열에 대한 제어권을 취득하고, 스위치 제어에 따라 읽기/쓰기 동작을 할 수 있었습니다.



만일 100개의 숫자를 직접 덧셈기에 입력하는 대신 RAM 배열에 입력해 두었다면, 숫자를 몇 개 잘못 입력했더라도 수정하기 훨씬 쉬웠을 것입니다.

그럼 이제 RAM 배열과 누적 덧셈기(accumulating adder)를 연결하는 방법을 알아낼 차례입니다. 스위치의 출력 대신에 RAM 데이터 출력 신호를 덧셈기의 입력 신호로 사용할 수 있다는 것은 자명합니다. 하지만, (14장에서 했던 것과 같이) 16비트 카운터로 RAM 배열의 주소 입력을 제어할 수 있다는 점이 명확하지는 않습니다. 이 회로에서는 RAM에 대한 데이터 입력과 쓰기 신호가 필요하지 않습니다.

이 회로가 이전에 개발했던 연산기들보다 간단하지는 않습니다. 이 회로를 이용하기 전에 지우기 스위치를 닫아 래치의 내용을 지워서 16비트 카운터의 출력을 0000h로 만듭니다. 그 후에 RAM 제어판의 취득 스위치를 닫습니다. 이제 더하려는 8비트 숫자를 0000h번지에서부터 입력해나가면 됩니다. 만일 100개의 숫자를 더해야 한다면, 0000h에서 0063h까지의 주소에 숫



자들이 저장될 것입니다. (RAM 배열에서 사용하지 않는 부분은 00h로 두면 되겠지요.) 이제 RAM 제어판에 있는 취득 스위치를 열고(이때부터 제어판이 RAM 배열에 대한 제어권을 가지지 않지요), 지우기 스위치를 열어둡니다. 그리고 앉아서 전구가 반짝이는 것을 지켜보시기만 하면 됩니다.

이제 어떻게 동작하는지 알아봅시다. 처음 지우기 스위치를 열었을 때에는 RAM 배열에 대한 주소는 0000h입니다. RAM 배열의 0000h번지에 저장된 8비트 데이터 값은 덧셈기의 입력이 됩니다. 덧셈기의 다른 입력은 00h가 되는데, 이는 누산기가 0으로 초기화되었기 때문이지요.

오실레이터(oscillator)에서는 0과 1사이를 매우 빠르게 천이(遷移, transition)하는 신호인 클럭 신호를 제공해줍니다. 지우기 스위치가 열린 이후에는 클럭이 0에서 1로 바뀔 때마다 다음 두 가지 일이 동시에 일어납니다. 덧셈기에서 출력되는 결과가 누산기에 저장되고, 동시에 16비트 카운트가 증가되어 RAM 배열에서 다음 값에 대하여 주소 지정이 일어나는 것이지요. 따라서 지우기 스위치가 열린 후 처음으로 클럭이 0에서 1로 변했다면, 래치에는 더하

려는 첫 번째 값이 저장될 것이고, 카운터는 0001h로 증가할 것입니다. 다음 클럭이 변할 때는 래치에 첫 번째 값과 두 번째 값의 합이 저장될 것이고, 카운터의 값은 0002h로 증가하게 됩니다. 그 뒤에도 마찬가지지요.

물론, 제대로 동작하려면 몇 가지 가정이 있어야 합니다. 일단 오실레이터의 동작 속도는 그 이외 회로의 동작 속도보다 느려야 합니다. 클럭이 한 번 움직일 때마다 덧셈기의 출력에서 제대로 된 덧셈의 결과가 만들어지기 위하여 수많은 릴레이의 값이 변해야 하는데, 모든 릴레이의 값이 변하기 전에 클럭이 다시 바뀌면 문제가 발생하겠지요.

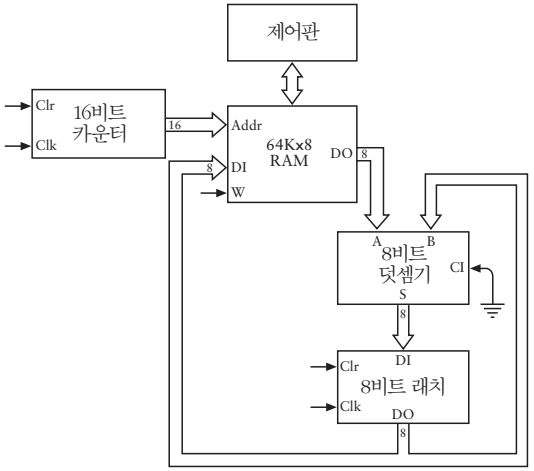
이 회로에서의 문제는 회로의 동작을 중지시킬 방법이 없다는 것입니다. 물론 어느 시점이 되면 전구에서의 깜빡임은 중단되게 되는데, 이는 RAM 배열에 나머지 숫자들이 모두 00h 값을 지니기 때문이지요. 이때 이진 합 결과를 읽을 수 있습니다. 하지만, 언젠가는 카운터가 FFFFh에 도달해서 다시 0000h로 돌아오게 되며, 이로 인하여 자동으로 덧셈을 수행하는 덧셈기에서는 첫 데이터부터 다시 덧셈을 수행해 나가게 됩니다.

이 덧셈기에는 다른 문제들도 있습니다. 8비트 수만 처리할 수 있으며, 덧셈만 사용할 수 있지요. 이는 덧셈을 위하여 RAM에 저장되어 있는 각각의 숫자가 255로 제한될 뿐 아니라, 그 결과도 255를 넘어갈 수 없음을 의미합니다. 이 덧셈기의 경우 따로 수를 뺄 수 있는 방법이 없지만, 2의 보수를 이용해서 음수를 사용할 수는 있습니다. 하지만 이 경우에 다룰 수 있는 수는 -128에서 127까지로 제한되겠지요. 좀 더 큰 수(예를 들어 16비트 숫자)를 더할 수 있도록 만드는 가장 쉬운 방법은 RAM 배열, 덧셈기, 래치의 데이터 크기를 두 배로 늘리고, 전구도 8개 더 붙이는 것이지요. 하지만, 이 정도의 투자를 할 생각은 전혀 없습니다.

물론 이런 말을 쉽게 하는 이유는 어떤 방법으로 이 문제를 해결해야 하는지 알고 있기 때문이지요. 일단 여기서는 우선 다른 문제들을 해결하는 데

집중하도록 하겠습니다. 만일 숫자 100개를 전부 더해서 하나의 합을 구하는 것이 아니라면 어떻게 해야 할까요? 스스로 연산을 수행하는 자동화된 덧셈기(automated adder)를 이용해서 50쌍의 숫자들을 더하여 서로 다른 50개의 합을 구하려면 어떻게 해야 할까요? 혹은 숫자 2개, 10개, 혹은 100개를 자유자재로 더할 수 있는 다용도의 덧셈기를 만들려면 어떻게 해야 할까요? 또한, 결과를 읽기 쉽도록 만들려면 어떻게 해야 할까요?

앞에서 보여드린 자동화된 덧셈기의 경우 래치에 연결된 전구를 이용해서 중간 합을 표시합니다. 만일 50쌍의 숫자를 더해서 50개의 합을 얻어내야 한다면 좋은 방법이 아닐 것입니다. 그것보다는 연산 결과가 RAM에 다시 기록되게 만들고, 이후에 RAM 제어판을 통하여 결과를 하나씩 확인할 수 있다면 편할 것 같습니다. 물론, 제어판에 결과 확인을 위한 전구가 달려 있다면 더욱 편하겠지요.

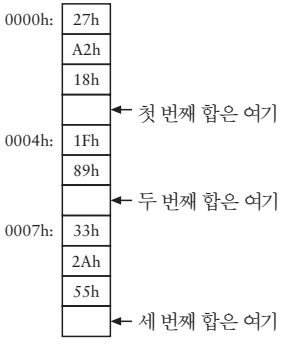


달리 이야기하면, 래치에 붙어 있는 전구를 떼어낼 수 있을 것 같습니다. 그 대신 그림과 같이 덧셈의 결과가 쓰일 수 있도록 래치의 출력이 RAM의 데이터 입력 부분에 연결되어야만 합니다.

앞의 자동화된 덧셈기 그림에서 오실레이터와 지우기 스위치 부분은 제외하였습니다. 카운터와 래치에 입력되는 지우기와 클럭 신호가 어디에서부터 오는지 명확하지 않거니와 지금은 그 부분이 별로 중요하지도 않기 때문이지요. 그보다 RAM 데이터 입력을 사용하기 위하여 RAM에 대한 쓰기 신호를 제어할 수 있어야 한다는 것이 중요합니다.

일단은 회로에서 명확하지 않은 부분을 걱정하기보다 우리가 알아내야 하는 문제에 좀 더 집중해 보겠습니다. 여기서 우리가 만들려고 하는 것은 단순히 주어진 숫자들을 더하기만 하는 데 제한되지 않는 자동화된 덧셈기입니다. 즉, 얼마나 많은 숫자를 더할 수 있을 것이고, 얼마나 많은 합이 만들어져 RAM에 저장되고, 이를 나중에 확인할 수 있을지와 무관하게 잘 동작하는 덧셈기를 만들려고 하는 것이지요.

예를 들어, 일단 세 개의 숫자를 더하고, 두 개의 숫자를 더한 다음에, 또 다시 세 개의 숫자들 더해야 한다고 가정해 봅시다. 보통 0000h번지에서부터 RAM에 데이터를 입력해 나가게 될 것이므로, 메모리 내부에는 다음과 같은 내용이 저장될 것이라 생각할 수 있습니다.



이 책에서는 위의 그림에서 보이는 것과 같이 메모리 영역을 표현할 것입니다. 위의 그림에서 상자들은 메모리의 내용을 나타내며, 각각의 상자는 메

모리의 한 바이트씩을 나타냅니다. 각 상자의 주소는 왼쪽에 표기되며, 주소의 경우 순차적으로 증가하므로 모든 상자에 주소를 적어둘 필요는 없습니다. 주소가 적혀 있는 상자와 주소를 알아보려는 상자의 위치를 통해서 이를 확인할 수 있는 것이지요. 메모리에 대한 메모가 필요할 때는 이를 상자의 오른쪽에 적어둡니다. 위의 그림에서 각각의 메모는 자동화된 덧셈기에서 발생하는 세 개의 결과가 빈 상자에 저장되도록 만들 것이라는 점을 알려줍니다. (그림에서는 빈 상자를 그렸지만, 실제 메모리의 경우 정말로 비어 있을 필요는 없지요. 실제로 메모리에는 항상 어떤 값이 들어가 있으며, 초기화되지 않았을 때는 아무런 쓸모가 없는 쓰레기 값이라도 들어가 있습니다.)

귀찮게 이것저것 하느니 그냥 16진수 연산을 해서 상자 안에 답을 채워 넣고 싶은 생각이 들지도 모르겠습니다. 하지만, 여기서는 연산을 보여드리려 한 것이 아니라 우리를 대신해서 덧셈을 수행할 수 있는 자동화된 덧셈기를 만들려고 하는 것을 잊으시면 안 됩니다.

처음에 만들었던 덧셈기의 경우 RAM에 있는 데이터와 누산기(accumulator)라 불리는 8비트 래치의 내용을 더하는 일만 할 수 있었던 반면, 이번에 만들 버전에서는 아래에서 설명하는 서로 다른 네 가지 동작을 포함하도록 해야 합니다. 덧셈을 수행하기 위한 첫 동작은 메모리에서 누산기로 한 바이트의 데이터를 가지고 오는 것입니다. 이를 로드(Load)라 부르겠습니다. 두 번째로 해야 하는 동작은 메모리에 있는 한 바이트와 누산기에 있는 내용을 더하는(Add) 것입니다. 세 번째로 수행해야 하는 동작은 누산기에 들어가 있는 합을 메모리에 저장(Store)하는 것이며, 마지막으로 적절한 시점에서 자동화된 덧셈기를 정지(Halt)시킬 수 있는 방법이 필요합니다.

앞에서 이야기한 예제를 기준으로 자동화된 덧셈기의 동작을 조금 더 자세히 이야기하자면 다음과 같습니다.

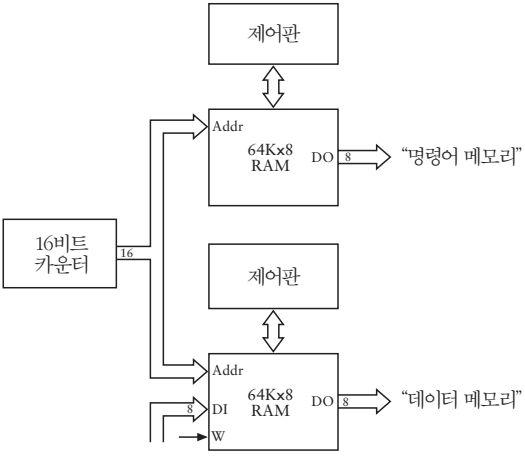
- 0000h 주소에 있는 값을 로드해서 누산기에 저장
- 0001h 주소에 있는 값과 누산기의 값을 더함
- 0002h 주소에 있는 값과 누산기의 값을 더함
- 누산기에 있는 값을 0003h 주소에 저장
- 0004h 주소에 있는 값을 로드해서 누산기에 저장
- 0005h 주소에 있는 값과 누산기의 값을 더함
- 누산기에 있는 값을 0006h 주소에 저장
- 0007h 주소에 있는 값을 로드해서 누산기에 저장
- 0008h 주소에 있는 값과 누산기의 값을 더함
- 0009h 주소에 있는 값과 누산기의 값을 더함
- 누산기에 있는 값을 000Ah 주소에 저장
- 자동화된 덧셈기의 동작을 중지시킴

처음 만들었던 자동화된 덧셈기와 마찬가지로 메모리 안에 있는 각각의 바이트 값은 0000h에서부터 순서대로 주소가 매겨진다는 점을 주의 깊게 보십시오. 처음에 만들었던 자동화된 덧셈기는 단순히 주소 값이 지정된 번지의 메모리에서 데이터를 가지고 와서 누산기에 있는 내용과 더할 수 있었습니다. 물론, 여전히 몇몇 경우에는 그런 작업이 필요하지요. 하지만, 가끔은 메모리에서 데이터를 직접 로드하거나 누산기에 있는 값을 메모리로 저장할 필요가 있습니다. 또한, 모든 작업이 끝난 다음에 RAM의 내용을 확인해 볼 수 있도록 자동화된 덧셈기의 동작을 중단시킬 필요도 있습니다.

어떤 방법으로 이런 동작들을 처리할 수 있을까요? 단순히 RAM에 숫자를 잔뜩 입력하고 나서 자동화된 덧셈기에서 제대로 연산을 할 것이라 기대하기는 어렵습니다. 자동화 덧셈기가 RAM에 있는 숫자들을 이용해서 처리해야 하는 로드, 더하기, 저장하기, 중단 등의 동작을 나타내는 숫자 부호가

필요합니다.

아마도 이런 부호를 저장하는 가장 편한 방법(가장 하드웨어가 적게 드는 방법은 아닐지라도)은 별도의 RAM을 사용하는 것입니다. 두 번째 RAM은 첫 번째 RAM과 동시에 접근됩니다. 두 번째 RAM에는 더해야 하는 숫자들이 아니라 자동화된 덧셈기에서 원래 RAM의 동일한 주소에 있는 숫자들을 사용하여 수행해야 하는 동작들이 저장되어 있습니다. 이 두 개의 RAM은 각각 데이터 메모리(Data, 원래의 RAM)와 명령어 메모리(Code, 새로운 RAM)라 이름을 붙일 수 있겠습니다.



새로 만든 자동화 덧셈기는 이미 덧셈의 결과를 (데이터라 이름 붙여진) 원래의 RAM으로 저장할 수 있지요. 하지만, (명령어 메모리라 이름 붙인) 새로운 RAM에는 오로지 제어판을 통해서만 값을 저장할 수 있습니다.

새롭게 만든 자동화된 덧셈기에서 수행해야 하는 네 가지 동작을 위해서 명령어가 네 개 필요합니다. 이러한 명령어를 위하여 어떤 값을 할당해도 관계는 없습니다. 아래와 같이 할당할 수도 있지요.

동작	부호
로드	10h
저장	11h
더하기	20h
중단	FFh

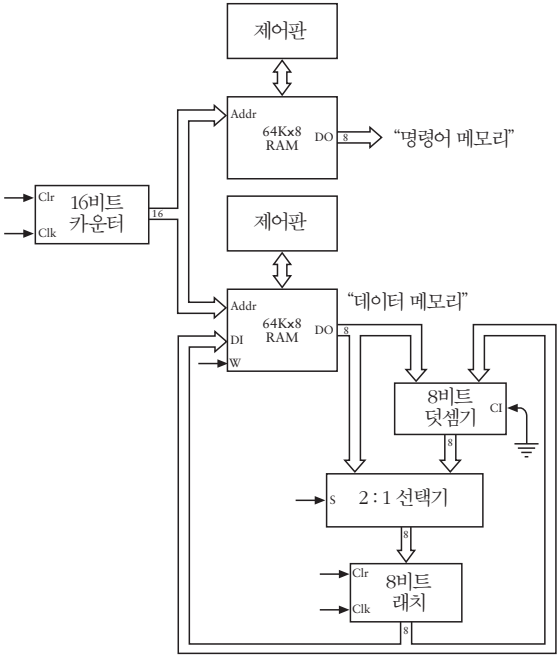
따라서 앞의 예제에서 보여드렸던 세 쌍의 덧셈을 수행하려면 제어판을 이용해서 명령어 RAM에 다음과 같은 값을 넣어야 합니다.

0000h:	10h	로드
	20h	더하기
	20h	더하기
	11h	저장
0004h:	10h	로드
	20h	더하기
	11h	저장
0007h:	10h	로드
	20h	더하기
	20h	더하기
	11h	저장
000Bh:	FFh	중단

이 명령어 RAM의 내용과 (330쪽에서 보여드렸던) 덧셈에 사용될 데이터가 있는 RAM의 내용을 서로 비교해 볼까요. 명령어 RAM에 있는 각 명령어는 데이터 RAM의 같은 주소에 있는 값이 누산기로 로드될 것인지, 더해질 것인지, 혹은 누산기의 값이 메모리로 저장될 것인지 등의 동작을 나타냅니다. 이러한 형태로 사용된 숫자 부호를 보통 명령어 부호, 또는 동작 부호, 혹은 아주 간단하게 동작코드(opcode)라 부릅니다. 이 숫자 코드들은 회로가 어떤 동작(operation)을 수행할 것인지 지시하는(instruct) 역할을 합니다.

앞에서도 언급되었지만, 처음에 만든 자동화된 덧셈기에 있던 8비트 래치의 출력은 데이터 RAM의 입력과 연결될 필요가 있습니다. 이는 저장(Store) 명령의 동작을 위한 것이지요. 물론, 다른 부분들도 변경되어야 합니다. 원

래 8비트 덧셈기의 출력은 8비트 래치의 입력으로 연결되어 있었습니다. 하지만, 이제는 로드(Load) 명령어를 처리하기 위해 데이터 RAM의 출력이 가끔은 8비트 래치의 입력과 연결되어야 합니다. 이러한 동작을 처리하기 위해서 2 : 1 데이터 선택기 한 개가 필요합니다. 이를 위하여 수정된 자동화 덧셈기는 아래 그림과 같습니다.



위의 그림에서 몇 부분이 빠져있기는 하지만, 여러 구성요소들을 연결하는 8비트 데이터 패스는 모두 그려져 있습니다. 16비트 카운터는 두 RAM에 주소를 제공하는 역할을 합니다. 데이터 RAM의 출력은 일반적으로 8비트 덧셈기의 입력과 연결되어 덧셈 명령에서 사용됩니다. 하지만 데이터 RAM의 출력(로드 명령의 경우)이나 덧셈기의 출력(더하기 명령의 경우)이 모두 8비트 래치의 입력으로 연결될 수도 있습니다. 이 경우에 2 : 1 선택기(MUX)가 사용

되는 것이지요. 래치의 출력은 보통 덧셈기의 입력으로 되돌아가지만, '저장' 명령어를 위하여 데이터 RAM의 입력으로도 사용됩니다.

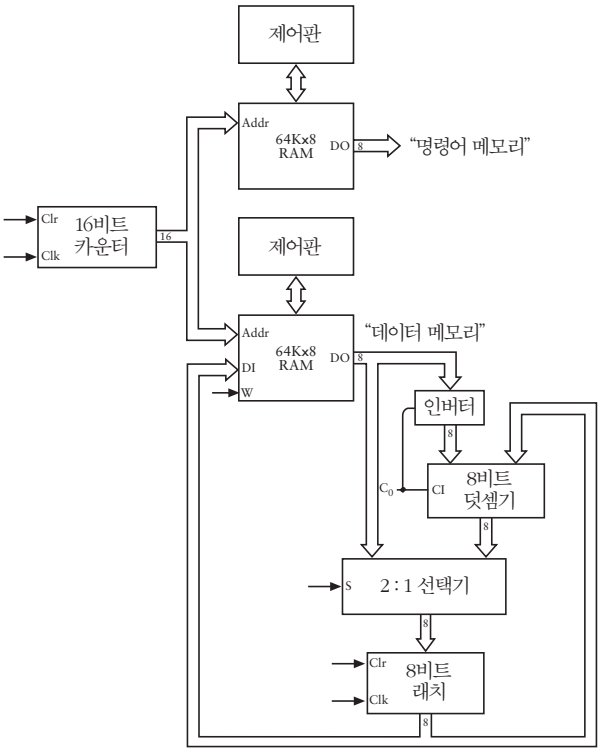
이 그림에는 보통 제어신호(control signal)라 불리는 각각의 구성요소를 제어하기 위한 신호들이 빠져 있습니다. 이 신호들에는 16비트 카운터에 대한 클럭, 지우기(Clear) 입력, 8비트 래치에 대한 클럭, 지우기 입력, 데이터 RAM에 대한 쓰기(Write) 입력, 그리고 2 : 1 선택기의 선택(Select) 입력 등이 포함됩니다. 이 신호들의 일부가 명령어 RAM에서 출력된 명령어에 따라 발생한다는 점은 자명하지요. 예를 들어, 명령어 RAM에서 '로드' 명령을 읽어왔다면 2 : 1 선택기의 Select 입력은 '0' 값(데이터 RAM의 출력을 선택한다는 거죠)을 가지고 있어야 합니다. 또한, 데이터 RAM에 대한 '쓰기' 신호 값은 동작코드의 값이 '저장' 명령인 경우에만 '1'로 지정됩니다. 이러한 제어신호들은 논리 게이트를 다양하게 조합해서 만들 수 있겠지요.

하드웨어를 많이 추가하지 않아도, 명령어를 하나 추가해서 누산기에 있는 값을 뺄 수 있는 회로를 만들 수 있습니다. 우선적으로 해야 할 일은 명령어를 추가해서 명령어 표를 확장하는 것입니다.

동작	부호
로드	10h
저장	11h
더하기	20h
빼기	21h
중단	FFh

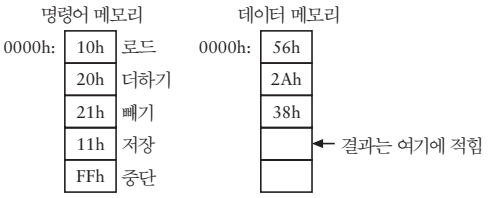
'더하기'와 '빼기'를 위한 명령어의 차이는 명령어 부호의 최하위 비트(LSB; least-significant bit)인 C_0의 값이 다르다는 점입니다. 명령어 부호 값이 21h인 경우 회로는 Add 명령의 경우와 대부분 같은 동작을 하지만, 데이터 RAM의 출력이 덧셈기로 들어가기 전에 반전(invert)되고, 덧셈기의 자리올림 입력이 '1'로 설정된다는 점에서 차이가 있습니다.[3] 인버터를 포함하는 새롭

게 변형된 자동화 덧셈기에서는 C_0 신호를 이용하여 이 동작을 수행하게 됩니다.



56h와 2Ah를 더하고, 그 합에서 38h를 빼는 예를 들어보겠습니다. 다음과 같은 명령어들과 데이터를 각각 두 RAM에 넣어두고 구동시키면 될 것 같습니다.

3 (옮긴이) 즉, 음수로 만들어 주는 거죠.



첫 번째 로드 명령이 끝난 후에 누산기에는 56h 값이 저장되며, 그 이후에 '더하기' 명령을 수행하면 누산기에는 56h와 2Ah의 합인 80h 값이 저장됩니다. '빼기' 명령은 데이터 RAM의 다음 값인 38h를 비트 단위로 반전(invert)하여 가지고 오는 것이지요. 반전된 값인 C7h가 80h에 더해지는데 이때 덧셈기의 자리올림수(carry)는 '1'이 됩니다.

```
  C7h
+ 80h
+  1h
-----
  48h
```

그 결과는 48h가 되는데, 이는 십진수로 86 더하기 42 빼기 56의 결과가 72가 되는 것과 같지요.

아직 충분히 언급되지는 않았지만 이 덧셈기에 남아있는 문제점은 덧셈기나 다른 부분, 이와 연결된 모든 부분이 겨우 8비트 데이터 폭을 가지고 있다는 점입니다. 앞에서 8비트 덧셈기 두 개를 붙여서 16비트 장치를 만드는 방법(물론, 다른 장치들도 비슷한 방법으로 확장해야겠지요)에 대해서 설명한 적이 있습니다.

하지만, 좀 더 비용이 적게 드는 방법이 있을 것 같습니다. 아래 예와 같이 두 개의 16비트 숫자를 더한다고 가정해볼까요.

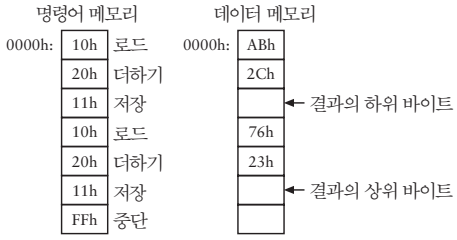
```
  76ABh
+ 232Ch
-------
```

위의 16비트 덧셈은 결국 아래 식과 같이 오른쪽의 바이트(보통 낮은 자리의 바이트라 부릅니다)를 따로 더하고, 왼쪽의(높은 자리의 바이트라 부르지요) 바이트를 따로 더하는 것과 동일합니다.

```
  ABh
+ 2Ch
------
  D7h
```

```
  76h
+ 23h
------
  99h
```

결과로 생성되는 16비트 수인 99D7h는 다음과 같이 두 개로 나누어 메모리에 저장할 수 있습니다.



결과적으로 0002h 번지에는 D7h가 저장되고, 99h는 0005h 번지에 저장됩니다.

물론, 이런 방법이 항상 가능한 것은 아닙니다. 앞의 예에서 선택한 수에서는 정상적으로 동작하지만, 76ABh와 236Ch를 더하는 경우라면 어떻게 될까요? 이 경우에 아랫자리 바이트들을 더하면 자리올림수가 발생합니다.

```
  ABh
+ 6Ch
------
 117h
```

이 자리올림수는 윗자리에 있는 두 바이트들의 합에 같이 더해져야 합니다.

```
   1h
+ 76h
+ 23h
------
   9Ah
```

따라서 최종적인 결과는 9A17h가 되는 것이지요.

앞에서 만든 자동화된 덧셈기에서 16비트 숫자를 정확하게 더할 수 있도록 회로를 확장할 수 있을까요? 네, 당연히 할 수 있습니다. 8비트 덧셈기의 자리올림 출력을 저장해 두었다가 다음 덧셈에서 자리올림 입력으로 사용할 수 있도록 하면 됩니다. 어떻게 이 한 비트를 저장할 수 있을까요? 물론 1비트 래치를 사용하면 되며, 이를 자리올림 래치라고 부릅니다.

자리올림 래치를 사용하려면 뭔가 다른 명령어가 필요합니다. 이 명령을 '자리올림을 이용하여 더하기(Add with carry)'라고 부르도록 하겠습니다. 8비트 숫자를 더할 때는 일반적인 '더하기(Add)' 명령을 사용하시면 됩니다. 이 때 덧셈기의 자리올림 입력은 0이 되며, 덧셈기의 자리올림 출력은 자리올림 래치에 저장되는 것이지요. (나중에 사용하지 않게 되더라도 일단 저장되는 것입니다.)

만일 두 개의 16비트 숫자를 더하려면, 일단 낮은 자리의 바이트를 처리하기 위하여 일반적인 더하기 명령을 사용해야 합니다. 앞에서 이야기한 것처럼 덧셈기의 자리올림 입력으로 0이 입력된 상태에서 덧셈이 수행되며, 덧셈기의 자리올림 출력은 자리올림 래치에 저장됩니다. 윗자리의 바이트들을 서로 더할 때는 '자리올림을 이용하여 더하기' 명령을 사용하면 됩니다. 이 명령이 사용됨으로써 두 수가 더해질 때 덧셈기의 자리올림 입력으로는 자리올림 래치의 출력이 사용됩니다. 따라서 첫 덧셈에서 자리올림이 발생하였다면, 두 번째 덧셈 과정에서 이 값이 사용됩니다. 만일 자리올림이 발생하지 않았다면 자리올림 래치의 출력은 0이 되겠지요.

만일 16비트 숫자를 빼려면 '빌림을 사용하여 빼기(subtract with borrow)'라 불리는 새로운 명령어가 있어야 합니다. 보통 빼기 명령의 수행 과정에서는 빼는 수의 값을 반전시키고, 덧셈기로 입력되는 자리올림 값을 1로 만듭니다. 뺄셈에서는 자리올림 출력이 1이 되는 것이 정상이므로 그냥 무시됩니다. 하지만, 16비트 숫자를 뺄 때, 자리올림 출력은 자리올림 래치에 저장되어야 할 필요가 있습니다. 두 번째 뺄셈에서 자리올림 래치에 저장된 결과를 사용해야 하기 때문이지요.

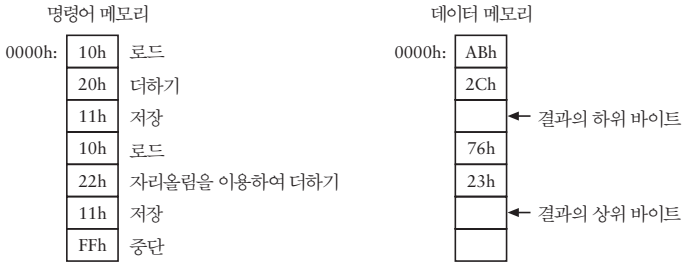
새롭게 '자리올림을 이용하여 더하기' 명령과 '빌림을 사용하여 빼기' 명령이 추가되어 모두 7개의 명령을 가지게 되었습니다.

명령	부호
로드	10h
저장	11h
더하기	20h
빼기	21h
자리올림을 이용하여 더하기	22h
빌림을 이용하여 빼기	23h
중단	FFh

빼기나 빌림을 이용하여 빼기 명령어를 수행할 때 덧셈기로 입력되는 숫자는 비트 단위로 반전됩니다. 또한, 덧셈기의 자리올림 출력은 자리올림 래치의 입력으로 들어갑니다. 더하기, 빼기, 자리올림을 이용하여 더하기, 빌림 등의 명령을 사용할 때 자리올림 래치에는 클럭이 공급됩니다. 뺄셈 연산이 수행될 때나 '자리올림을 이용하여 더하기' 또는 '빌림을 이용하여 빼기' 명령이 수행되고 있으며, 자리올림 래치의 출력이 1이 될 때 8비트 덧셈기의 자리올림 입력은 1이 됩니다.

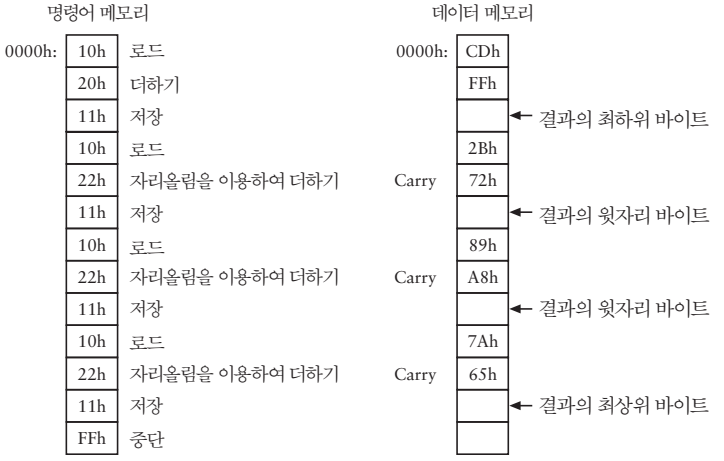
'자리올림을 이용하여 더하기(Add with Carry)' 명령의 경우 8비트 덧셈기의 자리올림 입력은 이전에 수행된 '더하기' 또는 '자리올림을 이용하여 더하

기(Add with Carry)' 명령에서 자리올림이 발생한 경우에만 1이 됩니다. 따라서 '자리올림을 이용하여 더하기' 명령은 덧셈기보다 비트수가 큰 수를 더해야 하는 경우에 실제 연산에서 자리올림이 발생하는지에 관계없이 항상 사용하면 됩니다. 16비트 덧셈을 위한 프로그램은 다음과 같습니다.



위의 프로그램은 어떤 수에 대한 덧셈에서든 잘 동작합니다.

새로운 명령어들을 이용하여 우리가 만든 장치의 데이터 범위를 아주 크게 확장시킬 수 있습니다. 더 이상 8비트만 더할 수 있는 것은 아니지요. '자리올림을 이용하여 더하기' 명령을 반복함으로써, 16비트, 24비트, 32비트, 40비트 등의 큰 값을 더할 수 있습니다. 만일 32비트 값인 7A892BCDh와



65A872FFh를 더하고 싶다면, 하나의 '더하기' 명령과 세 개의 '자리올림을 이용하여 더하기'를 이용하면 됩니다.

여기서 가장 귀찮은 일은 이 값들을 메모리로 입력하는 과정입니다. 이진수를 표현하기 위하여 스위치를 조작해야 할 뿐 아니라 연속된 주소에 수를 넣을 수도 없지요. 예를 들어, 7A892BCDh를 입력하려면 각각 가장 아래 자리 바이트부터 0000h, 0003h, 0006h, 0009h번지 순으로 저장됩니다. 최종 결과를 얻어내기 위해서는 0002h, 0005h, 0008h, 000Bh번지에서 데이터를 하나씩 가지고 와야 합니다.

게다가 지금 만든 자동화된 덧셈기의 형태에서는 연산의 결과를 뒤에서 재사용할 수도 없습니다. 8비트 수를 3개 더하고, 그 합에서 8비트 수를 뺀 다음에 값을 저장해야 한다고 생각해 봅시다. 일단 로드 명령, 덧셈 2개, 뺄셈 1개, 저장 명령 1개가 필요합니다. 프로그램 뒷부분에서 처음으로 연산했던 두 수의 합에서 어떤 수를 빼려고 하면 어떻게 해야 할까요? 안타깝게도 앞에서 연산된 결과를 프로그램의 뒤에서 접근할 방법이 없으므로 처음부터 다시 계산해야 합니다.

이 문제는 우리가 만든 자동화된 덧셈기에서 명령어 메모리와 데이터 메모리를 0000h번지부터 순차적으로 동시에 접근하도록 만들었기 때문에 발생한 것이지요. 명령어 메모리에서 각 명령은 같은 주소의 데이터 메모리에 대응하게 되어 있습니다. 즉, '저장' 명령어는 어떤 값을 같은 주소의 데이터 메모리로 저장하는 동작을 수행하게 되므로, 이후에 누산기로 다시 읽어낼 방법이 없는 것이지요.

이러한 문제를 해결하기 위해서는 자동화된 덧셈기의 근본부터 고쳐나가야 합니다. 물론, 이 과정은 약간 복잡하게 보일 수 있지만, 이를 통하여 아주 유연하게 데이터에 접근할 수 있게 되리라 기대할 수 있습니다.

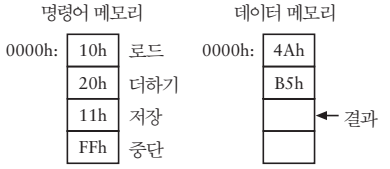
자. 시작해볼까요? 자동화된 덧셈기에는 7종류의 명령어가 있습니다.

명령	부호
로드	10h
저장	11h
더하기	20h
빼기	21h
자리올림을 이용하여 더하기	22h
빌림을 이용하여 빼기	23h
중단	FFh

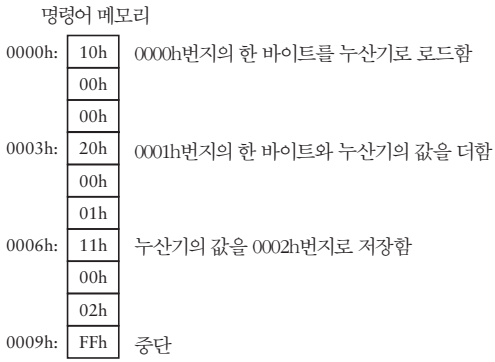
각 명령어는 메모리에서 1바이트씩 차지합니다. 이제 '중단(Halt)' 명령을 제외한 모든 명령어를 메모리에서 3바이트씩 차지하도록 바꿀 예정입니다. 명령어의 첫 번째 바이트는 명령어의 의미를 나타내고, 그 뒤의 2바이트는 16비트 메모리 주소를 나타냅니다. '로드(Load)' 명령의 경우에 명령 뒤에 나오는 주소는 데이터 RAM에서 누산기로 로드할 데이터가 있는 주소를 나타내는 것이지요.

'더하기', '빼기', '자리올림을 이용하여 더하기', '빌림을 이용하여 빼기' 명령에서 이 주소는 덧셈 혹은 뺄셈 연산과정에서 누산기에 있는 값과 연산을 수행할 데이터의 위치를 나타내게 됩니다. '저장' 명령에서 이 주소는 누산기의 값이 어디에 저장될 것인지 나타냅니다.

현재 자동화된 덧셈기에서 수행할 수 있는 가장 간단한 작업인 두 수를 더하는 예를 들어보겠습니다. 이를 수행하기 위해서는 명령어와 데이터 RAM을 아래와 같이 구성하면 됩니다.



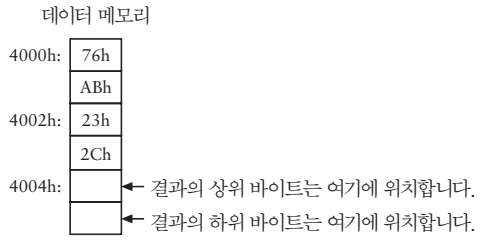
새로 변경된 자동화된 덧셈기의 각 명령어('중단' 명령을 제외하면)는 3바이트씩을 필요로 합니다.



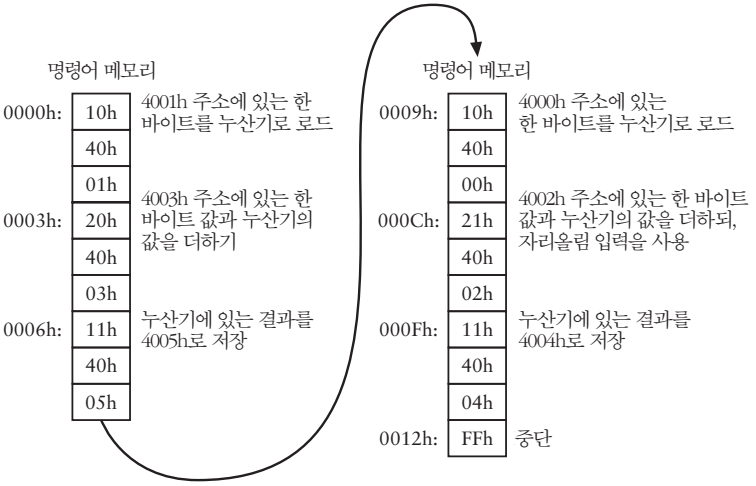
(중단을 제외한) 각각의 명령어 뒤에는 데이터 RAM에 대한 16비트 주소를 나타내는 2바이트가 따라옵니다. 여기서는 0000h, 0001h, 0002h라는 주소를 사용했지만, 어떤 것이든 될 수 있는 것이지요.

앞에서 16비트 숫자인 76ABh와 232Ch를 '더하기(Add)'와 '자리올림을 이용하여 더하기(Add with Carry)' 명령어로 더하는 예를 보여드렸습니다. 하지만, 이 숫자의 하위 바이트들은 0000h와 0001h에, 상위 바이트들은 0003h와 0004h에 각각 나뉘어 있으며, 덧셈의 결과는 0002h와 0005h에 저장되어 있으므로 처리하기에 아주 불편했습니다.

하지만, 위의 방법을 이용해서 자동화된 덧셈기를 바꾸면 위에서 사용했던 것보다 숫자들을 훨씬 더 편하고 합리적으로 메모리에 배치할 수 있습니다.



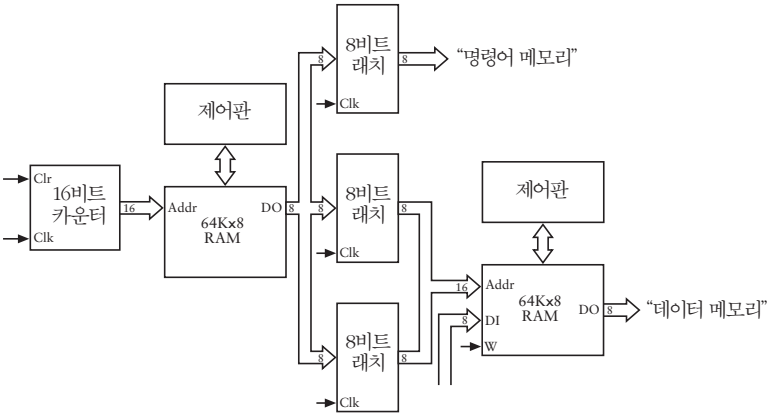
물론 이와 같이 6개의 저장 위치에 값들이 뭉쳐 있을 필요는 없습니다. 이 값들은 64KB 데이터 RAM의 어디에든 나뉘어서 위치할 수 있습니다. 이러한 메모리 위치에 존재하는 값들을 더하려면, 명령어 RAM에 있는 명령어들을 아래와 같이 설정하면 됩니다.



4001h와 4003h번지에 있는 아랫자리 바이트들은 먼저 더해지고, 그 결과는 4005h에 저장됩니다. (4000h와 4002h번지에 있는) 윗자리 바이트들은 자리올림 수를 이용하여 더해지며, 그 결과는 4004h번지에 저장됩니다. 여기에서 중단(halt) 명령어를 제거하고 명령어 메모리에 명령어를 더 추가하게 되는 경우, 뒤에서 수행하는 연산에서는 메모리 주소를 지정함으로써 간단하게 이전 연산에서 사용된 숫자들과 덧셈의 결과로 저장된 값을 사용할 수 있습니다.

이러한 설계를 구현하는 데 가장 중요한 것은 명령어 RAM의 출력이 세 개의 8비트 래치로 들어가야 한다는 점입니다. 3바이트 명령어의 각 바이트는 3개의 래치에 각각 저장됩니다. 첫 번째 래치에는 명령어 코드(instruction code)가 저장되며, 두 번째 래치에는 16비트 주소의 상위 부분, 세 번째 래치에는

주소의 하위 부분이 저장됩니다. 이 두 래치의 출력은 데이터 RAM의 16비트 주소로 사용됩니다.



메모리에서 명령어를 가지고 오는 과정은 명령어 패치(instruction fetch, 명령어 인출이라는 용어도 같이 사용됨)라 알려져 있습니다. 위의 설계에서 각 명령어는 3바이트 길이를 가지고 있으며, 메모리에서 한 번에 1바이트씩 명령어를 가지고 올 수 있으므로 명령어 패치를 위해서는 클럭 신호가 세 번 뛰어야 합니다. 따라서 하나의 명령어를 완전히 처리하기 위해서는 클럭 신호가 네 번 뛰어야 합니다. 이런 변화는 제어신호를 아주 복잡하게 만드는 것이지요.

이 장치는 명령어 코드에 대응하여 일련의 동작을 처리하므로, 명령어를 수행한다고 이야기할 수 있습니다. 하지만, 이 장치가 살아 있거나 한 것은 아니지요. 게다가 기계어(machine code)를 분석한다거나 어떤 것을 할지 판단하는 것도 아닙니다. 각각의 기계어 코드는 단지 독특한 방법으로 다양한 제어신호를 천이(遷移)시킴으로써 해당 장치가 다양한 일을 할 수 있도록 만드는 역할을 할 뿐입니다.

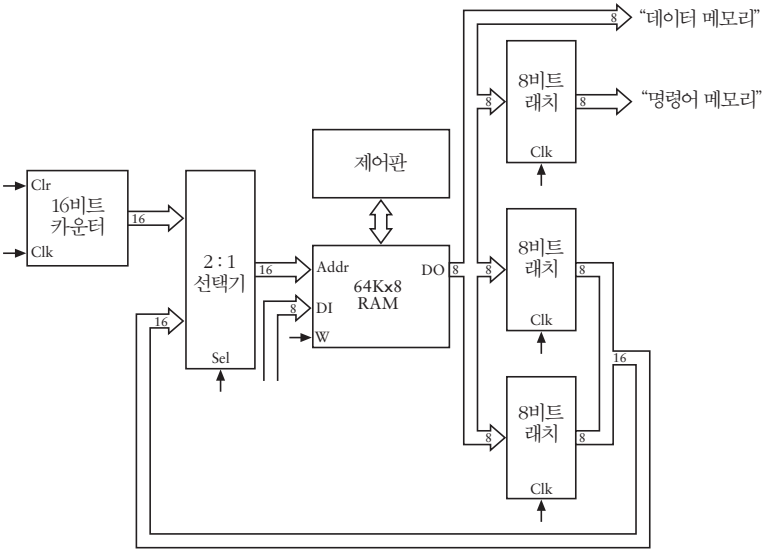
장치를 좀 더 다방면에 사용할 수 있게 되었으나 속도는 많이 느려졌습니다. 주파수가 같은 오실레이터가 사용된다면, 새로 만든 덧셈기는 앞에

서 처음 만들었던 자동화된 덧셈기보다 숫자를 더하는 속도가 1/4 밖에 되지 않습니다. 이러한 결과는 TANSTAAFL('탄스타플'이라 발음하죠)이란 공학적 원리를 보여주는데, 이는 '공짜 점심은 없다(There Ain't No Such Thing As A Free Lunch)'는 격언의 줄임말입니다. 이 말은 장치에서 어떤 부분을 좋게 만들면 다른 부분에는 악영향을 미치게 된다는 것입니다.

이러한 장치를 릴레이로 만든다면 회로에서 가장 큰 부분은 당연히 두 개의 64KB RAM이 될 것입니다. 게다가 훨씬 오래 전이었다면 이러한 구성요소에 소모되는 자원을 절약하기 위하여 일단 필요한 메모리는 1KB 정도라고 결정했을지도 모르겠습니다. 만일 0000h에서 03FFh까지의 주소 영역에 모든 데이터를 저장할 수 있는 것이 확실하다면 64KB보다 작은 메모리를 사용해도 괜찮을 것 같습니다.

여전히 두 개의 RAM을 사용해야 한다는 사실에 치를 떨 필요도 없습니다. 사실 전혀 걱정할 필요가 없습니다. 처음에는 자동화된 덧셈기의 구조를 최대한 간단하고 명확하게 하기 위해 명령어와 데이터라는 두 개의 RAM이 필요하다고 이야기했습니다. 하지만, 각 명령어를 위하여 3바이트를 사용하기로 결정하였으므로(두 번째와 세 번째 바이트는 데이터의 위치를 가리키는 주소를 나타냅니다), 이제 더 이상 RAM을 두 개로 분리하여 사용할 필요가 없습니다. 명령어와 데이터 모두 같은 RAM에 저장할 수 있기 때문이지요.

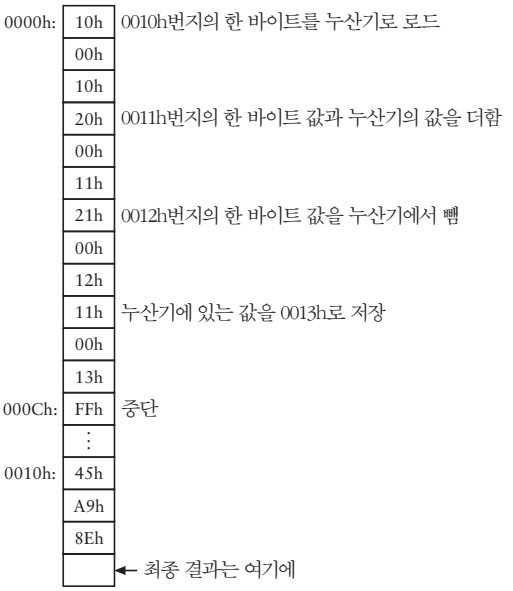
이를 위하여 2 : 1 선택기를 사용하여 RAM에 입력되는 주소를 선택할 수 있어야 합니다. 보통 때 주소는 앞에서와 마찬가지로 16비트 카운터의 출력이 됩니다. 명령어 코드와 명령어에 따라오는 2바이트 주소를 저장하기 위하여 RAM의 데이터 출력은 여전히 세 개의 8비트 래치에 연결됩니다. 명령어에 따라오는 16비트 주소는 2 : 1 선택기의 두 번째 입력이기도 합니다. 따라서 주소가 래치에 저장된 후에는 해당 주소가 2 : 1 선택기의 선택에 따라 RAM 주소 입력으로 전달됩니다.



지금까지 아주 많은 진전을 이루었습니다. 이제 명령어와 데이터를 하나의 RAM에 저장할 수 있게 되었습니다. 예를 들어, 다음 쪽의 그림은 두 개의 8비트 수를 더한 후에 빼기 위해서는 어떻게 해야 하는지 보여줍니다.

일반적으로 명령어는 0000h에서부터 시작하는데, 이는 리셋 이후에서부터 카운터가 바로 RAM에 접근하기 시작하기 때문입니다.[4] 마지막의 중단 명령은 000Ch번지에 저장되어 있습니다. 저장해야 하는 세 개의 숫자와 연산의 결과는 RAM의 어디에든 저장될 수 있지만, (물론, 명령어가 들어있는 RAM의 앞부분 13바이트는 사용하면 안되겠지요) 여기서는 0010h에서부터 데이터가 저장되도록 하겠습니다.

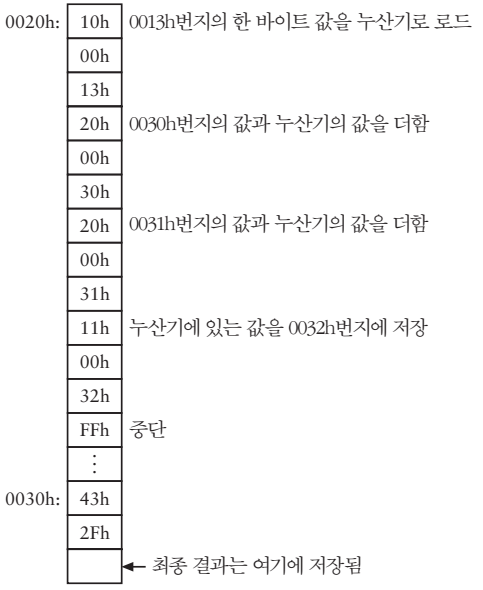
4 (옮긴이) 리셋을 시키면 카운터에 존재하는 래치와 같은 저장장치들의 값이 0으로 변경된다는 것을 기억하시지요?



연산의 결과에 두 값을 더 더해야 하는 방법을 찾아내야 한다고 가정해 보겠습니다. 일단 모든 명령어들을 완전히 바꾸는 방법이 있겠지만, 그렇게 하고 싶은 마음은 별로 없으실 것입니다. (기억하시죠? 인간은 귀찮은 걸 싫어합니다.) 이미 만들어둔 명령들 뒤에 새로운 명령어들을 추가해서 이런 작업을 수행할 수 있다면 좋겠지요. 처음에 해야 할 일은 000Ch번지에 있는 중단 명령을 로드 명령으로 바꾸는 것입니다. 그 뒤로 두 개의 더하기 명령과 하나의 저장 명령, 그리고 새로 중단 명령어 하나를 사용해야 합니다. 하지만, 0010h번지에 이미 데이터가 들어있다는 문제가 있습니다. 따라서 이 데이터를 다른 곳으로 옮기고 메모리 접근을 포함하고 있는 명령어들의 주소를 바꾸는 작업을 해야 하겠지요.

'흠, 계속 이런 작업이 필요하다면 명령어와 데이터를 하나의 RAM에 같이 넣는 것은 그다지 좋은 생각이 아니었는걸.'이라고 생각하실지 모르겠습니다. 하지만, 이런 문제는 빠르건 느리건 발생할 문제였습니다. 이제 이 문제

를 해결해 보겠습니다. 이를 위하여 새로운 작업을 위한 명령어를 0020h번지에 입력하고, 새로운 데이터들은 0030h번지에서부터 입력될 수 있도록 프로그램을 변경하는 방식을 사용하겠습니다.



처음 로드 명령어가 참조하는 메모리 위치인 0013h는 첫 번째 연산 결과가 저장되어 있는 위치라는 점에 주목하세요.

위와 같이 변경함으로써 일부 명령어들은 0000h에서부터 시작하고, 데이터 중 일부는 0010h에서부터 시작하며, 또 다른 명령어들은 0020h에서부터 시작하고, 나머지 데이터는 0030h번지부터 위치합니다. 이제 자동화된 덧셈기가 0000h에서 시작해서 모든 명령어를 잘 수행할 수 있도록 만들기만 하면 될 것 같습니다.

수행을 중단하지 않기 위하여 000Ch에 있는 중단 명령어를 없애야 한다는 것은 자명하지만, 실제로는 명령어를 없애는 것이 아니라 다른 명령으로

대체한다는 것을 의미합니다.

그것만으로 충분할까요? 문제는 중단 명령을 어떤 다른 것으로 바꾸더라도 명령어 바이트로 인식되리라는 것이지요. 또한, 그 뒤의 모든 3바이트 단위 위치에 있는 000Fh, 0012h, 0015h, 0018h, 001Bh, 001Eh번지에 있는 바이트도 역시 명령어 바이트로 인식됩니다. 이들 중에 한 바이트라도 11h라면 어떤 일이 벌어질까요? 명령어로 해석되는 경우에 저장 명령이 되겠지요. 또한, 이 명령어 뒤에 오는 2바이트의 값이 0023h 값이 되어 이 주소를 참조하게 되면 어떤 일이 벌어질까요? 이 명령은 누산기에 있는 값을 해당 주소(0023h)로 저장하도록 만들지만, 이 번지에는 이미 뭔가 중요한 것들이 들어 있는 상태이지요. 또한, 이런 일이 벌어지지 않았다 하더라도 자동화된 덧셈기는 메모리 001Eh번지에 접근한 이후에 실제로 명령어가 존재하는 0020h가 아닌 0021h에서 다음 명령어에 해당하는 몇 바이트들을 가지고 오게 됩니다.

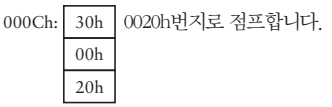
이로써 000Ch번지에 있는 중단 명령어를 없애는 것만으로는 충분치 않다는 것을 잘 알게 되었습니다. 더 좋게 만들려면 어떻게 해야 할까요?

새로운 명령어인 분기(分岐, Jump) 명령으로 교체하는 방법을 고려해 볼 수 있습니다. 이제 명령어 목록에 새로운 명령을 추가해 봅시다.

명령	부호
로드	10h
저장	11h
더하기	20h
빼기	21h
자리올림을 이용하여 더하기	22h
빌림을 이용하여 빼기	23h
분기	30h
중단	FFh

일반적으로 이 자동화된 덧셈기는 RAM에 대하여 순차적으로 번지를 생성하여 접근합니다. 분기 명령어는 장치에서 이러한 접근 패턴을 변경할 수 있도록 만들어 줍니다. 이 명령은 RAM의 다른 특정 지점에서부터 주소 지정이 이루어지도록 만들어줍니다. 이런 형태의 명령을 종종 Branch(분기) 명령, 혹은 '다른 장소로 이동하라'는 의미에서 Goto 명령이라 부르기도 합니다.

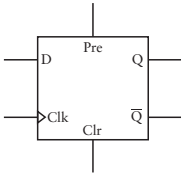
앞의 예에서 000Ch번지의 중단 명령을 분기 명령으로 바꿀 수 있습니다.



바이트 값 30h는 분기 명령을 나타내는 명령어 코드입니다. 이 명령어 코드 뒤에 따라오는 16비트 주소는 자동화된 덧셈기가 그 다음에 읽어야 하는 명령의 주소를 나타냅니다.

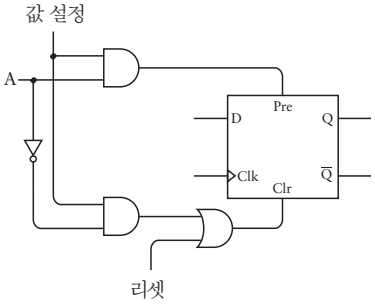
따라서 앞의 예에서 자동화된 덧셈기는 평소와 다름없이 0000h번지에서 시작해서 로드, 더하기, 빼기, 저장 명령을 한 번씩 수행합니다. 그 이후에 분기 명령을 수행하여 0020h번지의 로드 명령어부터 다시 시작해서 두 개의 더하기 명령과 한 개의 저장 명령을 수행한 이후에 마지막으로 중단 명령을 수행하게 됩니다.

분기 명령은 명령어 주소를 지정하는 16비트 카운터에 영향을 줍니다. 자동화된 덧셈기는 분기 명령을 만나는 경우 명령어 주소를 지정하는 프로그램 카운터의 주소를 분기 명령어 뒤에 있는 16비트 값으로 변경하여 출력할 수 있어야 합니다. 이러한 동작은 16비트 카운터를 만들 때 사용한 엣지 트리거 D 플립플롭에 있는 프리셋(Preset)과 지우기(Clear) 입력을 이용하여 구현할 수 있습니다.



일반적인 경우에 프리셋과 지우기 입력에는 0 값을 주며, 만일 프리셋이 1이 되면 Q의 값이 1로 변하고, 지우기가 1이 되면 Q의 값이 0으로 변한다는 것을 기억하실 것입니다.

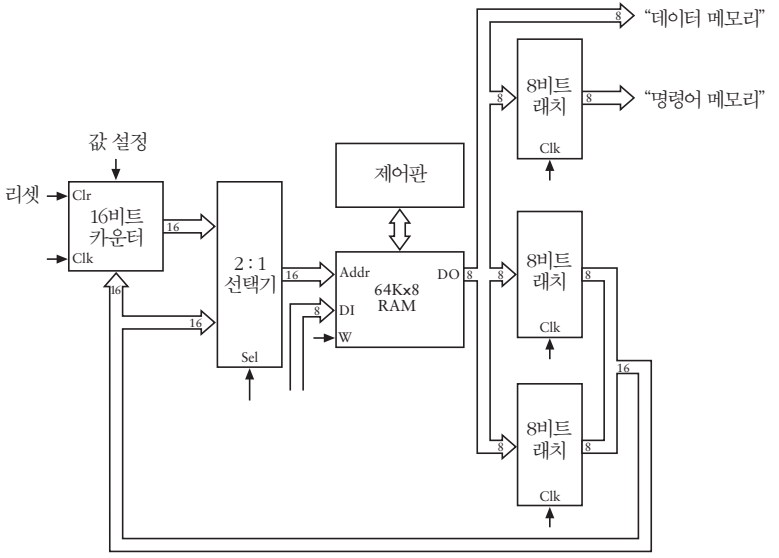
하나의 플립플롭에 주소를 나타내는 A라는 값을 로드하려면 다음 그림과 같이 회로를 연결하면 됩니다.



보통의 경우 값 설정(Set It) 신호는 0이 되며, 플립플롭의 프리셋 입력도 0이 됩니다. 또한, 리셋 입력이 1인 경우가 아니라면 지우기 입력도 0을 주게 됩니다. 이 신호를 이용하여 플립플롭은 값 설정 신호의 입력과 관계없이 값을 0으로 만들어 줄 수 있습니다. 값 설정 신호가 1인 경우에 A가 1이라면 프리셋 입력은 1이 되며, 지우기 입력은 0이 되므로 Q와 A의 값이 같아지게 됩니다. 마찬가지로 A의 값이 0인 경우에 프리셋 입력은 0이 되며, 지우기 입력은 1이 되어서 A와 Q의 값을 같게 만들어 줄 수 있지요.

16비트 카운트의 각 비트에 대하여 위와 같은 회로 구성이 필요합니다. 카운터에 특정한 값이 로드된 이후에는 해당 값에서부터 계속하여 숫자를 세게 됩니다.

이 부분을 제외하면 별로 바꿀 부분이 없습니다. 단지 분기 명령의 목적 주소로 사용되는 RAM에서 나온 16비트 주소 값을 래치하고, 이를 2 : 1 선택기에 입력하여 데이터 접근시 RAM의 주소로 사용될 수 있도록 만들고, 동시에 이를 16비트 카운터의 입력으로 연결하여 값 설정 신호에 따라 카운터로 로드될 수 있도록 만들면 됩니다.

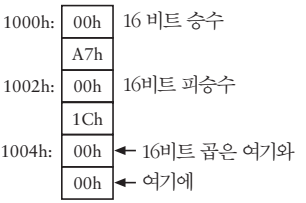


당연한 이야기지만, 명령어 코드가 30h인 경우에는 명령어 코드 뒤에 따라오는 주소가 래치된 이후에 값 설정 신호를 1로 설정해야만 합니다.

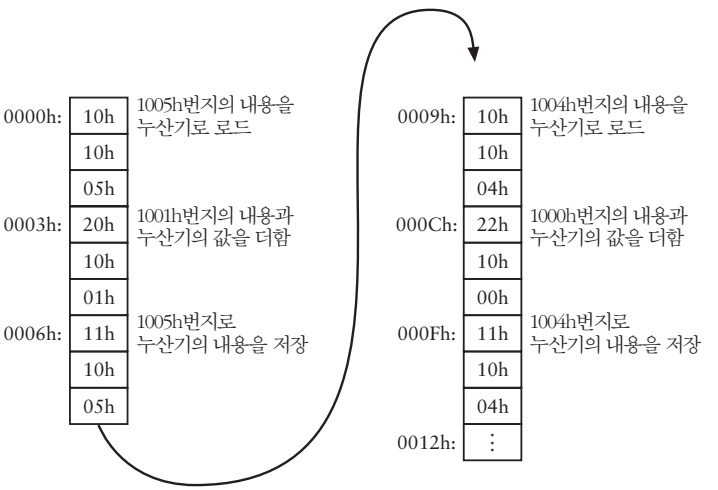
분기(Jump) 명령은 여러모로 유용합니다. 하지만, 항상 다른 주소로 이동하는 것보다는 특정한 조건에서만 어떤 주소로 이동할 수 있는 편이 훨씬 더 유용하게 사용될 수 있습니다. 이러한 명령어를 조건 분기(conditional jump)라 부르며, 이러한 형태의 명령이 얼마나 유용한지 확인할 수 있는 가장 좋은 방법은 '우리가 만든 자동화된 덧셈기에서 두 개의 8비트 수를 어떻게 곱할 수 있을까?'라는 질문에 대하여 답을 해보는 것입니다. 예를 들면, 어

떻게 하면 A7h와 1Ch의 곱을 얻어낼 수 있을까요?

쉬워 보이지 않나요? 두 8비트 수를 곱하면 16비트 결과가 나옵니다. 편의를 위하여 곱셈에 사용되는 세 개의 수 모두 크기가 16비트라고 가정해 봅시다. 일단 처음에 할 일은 곱할 수와 곱셈의 결과를 어디에 넣을 것인지 정하는 것이겠지요.



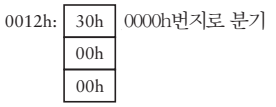
A7h와 1Ch(십진수로 28이죠)의 곱과 A7h를 28번 더하는 것이 같다는 사실은 누구나 알고 있습니다. 따라서 1004h와 1005h번지에 있는 16비트 값은 실질적으로 합의 누계가 되는 것이지요. 다음은 이 지점에 A7h를 한 번 더하는 코드입니다.



6개의 명령어를 수행하고 나면 1004h와 1005h에 있는 16비트 값은 A7 곱하기 1과 같은 값이 됩니다. 따라서 A7h 곱하기 1Ch를 수행하기 위해서는 이 6개의 명령어를 27번 더 반복해야 합니다. 또는 0012h에 중단 명령을 넣고, 리셋 버튼을 28번 눌러도 같은 결과를 얻을 수 있습니다.

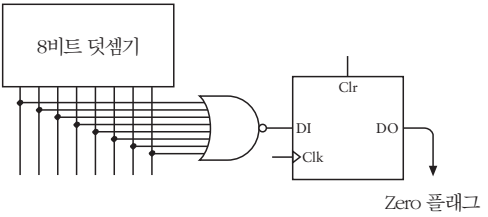
물론, 위의 두 방법 모두 아주 좋아 보이지는 않습니다. 두 방법 모두 어떤 일(많은 명령어를 입력해야 한다든지, 리셋 버튼을 눌러야 한다든지)을 곱하는 수만큼 사용자가 반복해야만 하는 것이지요. 16비트 수에 어떤 수를 곱해야 할 때마다 이런 일을 반복하는 것을 원치는 않을 것 같습니다.

그럼 0012h에 분기 명령어가 존재한다면 어떻게 될까요? 이 명령어는 카운터를 0000h로 돌아가게 만듭니다.



이 방법은 일종의 트릭입니다. 처음에는 1004h와 1005h에 있는 16비트 값이 A7h 곱하기 1이 됩니다. 그 이후에 분기 명령으로 인하여 프로그램의 흐름은 맨 처음으로 이동하게 되지요. 두 번째로 모든 명령어가 수행되면 16비트 결과 값은 A7h 곱하기 2가 됩니다. 결과적으로 언젠가 A7h 곱하기 1Ch도 되겠지만, 거기서 정지하지는 않습니다. 계속해서 돌고 돌고 또 도는 거죠.

우리가 원하는 것은 원하는 만큼만 반복시킬 수 있는 분기 명령어입니다. 이것이 바로 조건 분기 명령어인 것이지요. 게다가 구현도 어렵지 않습니다. 구현을 위해서는 우선 1비트 래치 한 개를 자리올림 래치와 같이 만들어 두어야 합니다. 이 래치는 8비트 덧셈기의 출력 값이 0일 때 1의 값을 지니게 되므로 Zero 래치라 부르도록 하겠습니다.



8비트 NOR 게이트의 출력은 모든 입력이 0일 때만 1의 값을 지니게 됩니다. Zero 래치의 클럭 입력은 자리올림 래치의 클럭 입력과 마찬가지로 더하기, 빼기, 자리올림을 이용하여 더하기, 빌림을 이용하여 빼기 명령이 수행될 때만 동작하게 됩니다. 래치된 값은 Zero 플래그라 부릅니다. 이 플래그의 동작은 생각하시는 것과 약간 다를 수 있으니 주의가 필요합니다. 덧셈기 출력의 모든 비트가 0값을 가지고 있다면 Zero 플래그가 1이 되며(0이 아니지요), 덧셈기의 출력 값이 0이 아닌 경우라면 Zero 플래그가 0이 됩니다.

자리올림 래치와 Zero 래치를 사용하면 분기와 관련된 명령어를 4배로 늘릴 수 있습니다.

명령	부호
로드	10h
저장	11h
더하기	20h
빼기	21h
자리올림을 이용하여 더하기	22h
빌림을 이용하여 빼기	23h
분기	30h
Zero면 분기	31h
자리올림이면 분기	32h
Zero가 아니면 분기	33h
자리올림이 아니면 분기	34h
중단	FFh

예를 들어, 'Zero가 아니면 분기(Jump If Not Zero)' 명령어는 Zero 래치의 출력 결과가 0일 때 지정된 주소로 분기하는 동작을 수행합니다. 이는 달리 이야기하면 직전에 수행되었던 더하기, 빼기, 자리올림을 이용하여 더하기, 빌림을 이용하여 빼기 명령어의 결과가 0일 때는 분기하지 않는다는 말이지요. 이러한 명령을 처리할 수 있도록 만들기 위하여 변경해야 하는 것은 일반적인 분기(Jump) 명령을 위하여 구현된 부분에 제어신호를 추가하는 정도입니다. 예를 들어 'Zero가 아니면 분기' 명령의 구현은 Zero 플래그가 0인 경우에만 16비트 카운터의 '값 설정' 신호가 변경될 수 있도록 만들기만 하면 되는 것이지요.

이제 위에서 보여드린 두 숫자를 곱하는 명령어 코드 중 0012h번지에서 시작되는 부분을 다음과 같이 바꾸면 될 것 같습니다.

0012h:	10h	1003h번지에서 한 바이트를 누산기로 로드
	10h	
	03h	
0015h:	20h	001Eh번지의 한 바이트를 누산기의 값과 더함
	00h	
	1Eh	
0018h:	11h	누산기의 값을 1003h에 저장
	10h	
	03h	
001Bh:	33h	zero 플래그의 값이 1이 아니라면 0000h번지로 분기
	00h	
	00h	
001Eh:	FFh	중단

이미 살펴보았지만, 첫 번째 처리 과정에서 1004h번지와 1005h번지에 저장되는 16비트 값은 A7h 곱하기 1의 값입니다. 위의 프로그램에서는 1003h번지에 있는 한 바이트 값을 누산기로 로드하도록 되어 있는데, 이 값이 1Ch입니다. 이 값은 001Eh번지에 있는 한 바이트 값과 더해지게 됩니다. 기억력

이 좋으신 분들은 001Eh에는 중단 명령어 코드(FFh)값이 들어 있다는 것을 기억하시겠지요. 하지만, 이 값도 데이터로써 로드되는 경우에는 명령어로써 해석되는 것이 아니라 하나의 유효한 값으로 처리됩니다.[5] FFh와 1Ch를 더하는 것은 1Ch에서 1을 빼는 것과 같은 결과가 되므로 값은 1Bh가 됩니다. 이 값은 0이 아니므로 Zero 플래그의 값은 0이 됩니다. 1Bh 값은 다시 1003h에 저장됩니다. 그 뒤의 명령은 'Zero가 아니면 분기' 명령인데, Zero 플래그의 값이 1이 아니므로 분기가 일어나게 됩니다. 따라서 그 다음 명령어는 0000h에서 가지고 오게 되지요.

저장(Store) 명령어의 수행은 Zero 플래그에 아무런 영향도 주지 못한다는 점을 기억하십시오. Zero 플래그는 연산에 관련된 더하기, 빼기, 자리올림을 이용하여 더하기, 빌림을 이용하여 빼기 명령어들의 결과에만 영향을 받기 때문에, 이들 명령어가 마지막으로 수행되었을 때 저장된 결과가 유지됩니다.

모든 명령어가 두 번째로 수행되면 1004h번지와 1005h번지에 저장되는 16비트 값은 A7h 곱하기 2의 값이 됩니다. 1Bh에는 다시 FFh 값이 더해져서 1Ah가 되며, 이는 0이 아니므로 다시 프로그램의 처음으로 돌아가게 되지요.

28번 수행된 이후에 1004h번지와 1005h번지에 저장되어 있는 16비트 값은 A7h 곱하기 1Ch 값이 됩니다. 1003h 주소에 있는 값은 1이 되지요. 이 값에 FFh를 더하면 그 결과 값은 0이 되므로, 드디어 Zero 플래그의 값이 1이 됩니다. 따라서 'Zero가 아니면 분기' 명령을 통하여 0000h로 다시 돌아가지 않고 그 다음 명령어인 중단을 수행하게 됩니다. 드디어 해냈습니다.

이제 진정으로 컴퓨터라고 부를 수 있을 만한 하드웨어를 만들어 냈다고 자신 있게 말할 수 있을 것 같습니다. 좀 더 정확히 말하자면, 우리가 만든

5 (옮긴이) 이 책에서는 예제를 간단하게 설명하기 위하여 이런 형태를 사용했지만, 대부분의 경우에 데이터 접근에서 이런 식으로 프로그램 코드 부분을 접근하는 것은 권장하지 않습니다. 프로그램이 변경됨에 따라 데이터 접근에 대한 부분도 같이 변경되어야 하는 경우가 발생하기 때문이지요. (해당 주소의 값이 중단이 아닌 다른 명령어로 바꿀 때를 생각해 보세요.)

것이 아주 기초적인 컴퓨터라고 할 수 있지만 컴퓨터인 것만은 확실하죠. 이러한 차이를 만드는 것은 바로 조건 분기의 존재 여부입니다. 계산기와 컴퓨터를 구분하는 가장 중요한 요소는 바로 조건에 따른 제어가 가능한 반복 혹은 루프(loop)라고 부르는 기능을 포함하고 있느냐는 부분이지요. 앞에서 조건 분기를 이용하여 어떻게 두 수를 곱할 수 있는지 알아보았습니다. 이와 비슷한 방법으로 두 수를 나눌 수도 있고, 8비트 수로 제한되지 않을 수도 있습니다. 16비트, 24비트, 32비트, 혹은 그보다 큰 수를 더하고, 빼고, 곱하고 나눌 수도 있지요. 마음만 먹으면 제곱근, 로그, 삼각 함수들도 계산할 수 있습니다.

이제 컴퓨터를 하나 만들었으니, 뭔가 컴퓨터를 다루는 것과 같은 느낌을 주는 용어를 써야 할 것 같습니다.

우리가 만든 컴퓨터는 불연속적인 숫자(discrete number)[6]를 다루고 있으므로 디지털 컴퓨터로 분류될 수 있습니다. 예전에는 아날로그 컴퓨터도 있었지만 현재는 거의 사라졌지요. (디지털 데이터는 값이 각각 나뉘어져 있는 불연속적인 데이터를 다룹니다. 아날로그 정보는 그 값이 연속적이며 모든 영역에 값이 존재할 수 있지요.)[7]

디지털 컴퓨터는 프로세서, 메모리, 적어도 하나의 입력장치와 출력장치, 이렇게 네 가지 주요 구성요소로 이루어집니다.[8] 여기서 만든 컴퓨터에서는

6 (옮긴이) 이산 수(離散 數)라는 표현도 많이 사용합니다. 이산 수학이란 말을 들어보신 분들도 많을 것입니다.

7 (옮긴이) 실제로 자연계에 존재하는 정보는 대부분 아날로그 정보입니다. 이 정보는 값이 연속적이라 내포하고 있는 데이터가 매우 크다는 장점이 있지만, 다루기가 까다롭고 잡음에 약하다는 단점이 있었습니다. 이 연속적인 값을 일정 단계로 샘플링해서 불연속적인 데이터로 가공하여 사용하는 경우가 많지요. 이런 과정을 양자화(Quantization)라 하고 양자화를 거친 데이터는 디지털 컴퓨터에서 처리가 가능해지는 것입니다. 물론, 디지털 데이터는 역양자화(Inverse Quantization) 혹은 보간(interpolation)을 거쳐서 중간에 사라진 데이터를 적절히 채워 넣어서 아날로그 데이터로 다시 변환할 수 있습니다.

8 (옮긴이) 입력장치와 출력장치는 묶어서 프로세서, 메모리, 입/출력장치의 세 가지 구성 요소를 가진다고 정의하는 책도 많습니다.

64KB RAM을 메모리로 사용했고, 입력장치와 출력장치로는 RAM 제어판의 스위치와 전구를 각각 사용했습니다. 이 스위치와 전구들을 이용해서 메모리에 수를 입력하고, 연산의 결과를 확인할 수 있지요.

그 이외의 것은 모두 프로세서입니다. 프로세서는 중앙처리장치(a central processing unit) 짧게는 CPU라 부릅니다.[9] 일반적으로 프로세서를 컴퓨터의 두뇌라고 부르는 경우가 많지만, 여기서 우리가 만든 프로세서는 두뇌라고 하기에는 거리가 있으므로 이 용어를 사용하지는 않겠습니다. (마이크로프로세서라는 용어가 요즘 널리 사용되고 있는데, 18장에서 좀 더 설명 드리겠지만 마이크로프로세서는 기술의 진보에 의하여 프로세서가 좀 더 작아진 것이라 생각하면 되겠습니다. 이 장에서 우리가 릴레이를 이용해서 만든 것을 가지고 아주 작은 것을 나타내는 마이크로(micro)라는 수식어를 붙이기는 좀 무리가 있지요.)

우리가 만든 프로세서는 8비트 프로세서라 부를 수 있습니다. 누산기가 8비트 데이터 폭을 가지고 있으며 대부분의 데이터 패스가 모두 8비트 데이터 폭을 가지고 있지요. 16비트 데이터 패스를 가지고 있는 부분은 RAM에 접근하기 위한 주소 생성 부분입니다. 만일, 이 부분을 8비트로 제한하는 경우 65,536바이트가 아닌 256바이트만 접근할 수 있는데, 이때 사용할 수 있는 데이터의 수가 너무 적어지는 단점이 있습니다.

일반적으로 프로세서는 다양한 구성요소를 가지고 있습니다. 앞에서 정의한 누산기(accumulator)는 프로세서 내부에서 숫자를 저장하는 역할을 수행하지요. 우리가 만든 컴퓨터에서 8비트 인버터와 8비트 덧셈기를 통칭해서 수치 연산 및 논리 연산 유닛(Arithmetic Logic Unit) 혹은 짧게 ALU라 부를 수 있습니다. 우리가 만든 ALU는 덧셈과 뺄셈 연산만을 수행할 수 있었지

9 (옮긴이) 이 책에서는 프로세서와 CPU를 같은 의미로 쓰고 있으나, 좀 더 엄격하게 말하면, 프로세서는 데이터를 처리할 수 있는 장치를 통칭하는 말로, CPU를 프로세서라는 분류의 일부라 보시는 것이 맞지요. 이 책에서 프로세서라 칭하는 것은 CPU를 의미한다고 보시면 되겠습니다.

요. 나중에 살펴보겠지만 좀 더 복잡한 컴퓨터의 ALU는 AND, OR, XOR와 같은 논리 연산을 포함하고 있습니다. 명령어 메모리 접근을 위하여 사용했던 16비트 카운터는 일반적으로 프로그램 카운터(Program Counter)라고 부릅니다.

우리가 만든 컴퓨터는 릴레이, 전선, 스위치, 전등으로 만들어져 있습니다. 모두 하드웨어지요. 반대로 메모리에 입력되어 있는 명령어들과 숫자(데이터)들은 소프트웨어라 부릅니다. 이 부분들은 하드웨어보다 훨씬 쉽게 변경할 수 있으므로 '부드럽다(soft)'고 이야기합니다.

컴퓨터를 이야기할 때 소프트웨어라는 용어는 대부분 컴퓨터 프로그램 혹은 짧게 프로그램이라는 용어와 거의 같은 뜻으로 사용됩니다. 소프트웨어를 작성하는 작업은 보통 컴퓨터 프로그래밍이라 알려져 있습니다. 앞에서 컴퓨터를 이용하여 두 수를 곱할 수 있도록 일련의 명령을 결정한 일이 바로 컴퓨터 프로그래밍입니다.

일반적으로 컴퓨터 프로그램은 코드(명령어들 자체를 의미합니다)와 데이터(코드에 의하여 조작되는 숫자들을 의미하지요)로 구분될 수 있습니다. 물론, 중단 명령어(FFh)가 숫자 -1로도 사용되었던 것과 같이 이 구분이 명확하지는 않을 수도 있습니다.

"휴가 동안 내내 코딩만 했네." 혹은 "코딩을 좀 하려고 아침 7시에 일어났어." 라고 종종 얘기하듯이 컴퓨터 프로그래밍은 코드 작성 혹은 코딩이라고 부르기도 합니다. 컴퓨터 프로그래머는 종종 코더라고 불리기도 합니다. 하지만 많은 프로그래머가 이 용어를 싫어하며, 그 대신 소프트웨어 엔지니어라는 용어를 더 선호하지요.

로드(Load)와 저장(Store)를 의미하는 10h, 11h 같이 프로세서의 동작에 대응하는 명령어의 동작 코드를 머신 코드(machine code) 혹은 기계어(machine language)라 이야기합니다. 언어(language)라는 용어가 사용된 것은 이것이 사

람이 읽고 쓰는 것과 유사하게 이를 기계가 이해하고 대응할 수 있다는 점 때문입니다.

앞에서 우리가 만든 컴퓨터로 전달되는 명령어를 이야기할 때 자리올림을 이용하여 더하기와 같이 약간 긴 용어들을 사용했습니다. 하지만 기계어는 니모닉(mnemonics)이라 불리는 대문자로 이루어진 짧은 약어를 할당하여 사용하는 것이 일반적입니다.

보통 니모닉에는 두 글자나 세 글자가 사용됩니다. 우리가 만든 컴퓨터에서 인식할 수 있는 기계어에 대응하는 각각의 니모닉은 다음과 같습니다.

명령	부호	니모닉
로드	10h	LOD
저장	11h	STO
더하기	20h	ADD
빼기	21h	SUB
자리올림을 이용하여 더하기	22h	ADC
빌림을 이용하여 빼기	23h	SBB
분기	30h	JMP
Zero면 분기	31h	JZ
자리올림이면 분기	32h	JC
Zero가 아니면 분기	33h	JNZ
자리올림이 아니면 분기	34h	JNC
중단	FFh	HLT

이러한 니모닉은 다른 축약어들과 같이 사용할 때 아주 유용합니다. 예를 들어, '1003h에 있는 바이트 값을 누산기로 로드'라고 길게 쓰는 것보다 다음과 같이 쓰는 것이 더 간결하지요.

```
LOD A, [1003h]
```

니모닉 오른쪽에 있는 A나 [1003h]와 같은 값은 인수(argument)라 부르고, 해당 로드 명령어가 어떤 방식으로 동작할 것인지 알려주는 역할을 합니다.

왼쪽에 있는 인수는 명령어 수행의 결과가 저장되는 목적지(destination; 여기서 A는 누산기를 의미합니다)를 나타내며, 오른쪽의 경우 명령어 수행과정에서 처리될 값(소스(source))을 나타내게 됩니다. 대괄호([ ])는 누산기로 1003h 값이 로드되는 것이 아니라 메모리 1003h번지에 있는 값이 누산기로 로드된다는 것을 의미합니다.

이와 비슷하게 '001Eh번지에 있는 한 바이트를 누산기의 값과 더하라'는 명령은 다음과 같이 짧게 쓸 수 있습니다.

```
ADD A,[001Eh]
```

'누산기의 내용을 1003h 번지로 저장하라' 는 명령은 다음과 같이 쓸 수 있습니다.

```
STO [1003h],A
```

위의 저장 명령에서도 목적지는 왼쪽에 있으며, 소스는 오른쪽에 위치합니다. 누산기의 내용은 메모리 1003h번지에 저장되어야 합니다. '만일 Zero 플래그 값이 1이 아니라면 0000h로 분기하라' 는 긴 명령은 다음과 같이 짧게 적을 수 있습니다.

```
JNZ 0000h
```

이 명령에서는 0000h 주소에 저장된 값이 사용되는 것이 아니라 0000h번지로 분기하는 것이므로 대괄호가 사용되지 않습니다.

명령어들은 순차적으로 나열되는 것이 일반적이므로 짧게 다음과 같이 나타내면 상자를 그릴 필요도 없고 읽기도 편할 것 같습니다. 어떤 명령어들이 특정한 주소에 저장되어 있다는 것을 나타내기 위해서는 아래와 같이 16진수 주소를 쓰고 콜론 기호(:)를 붙여주면 됩니다.

```
0000h:      LOD A,[1005h]
```

또한, 특정 주소에 있는 데이터의 경우 다음과 같이 표현할 수 있습니다.

```
1000h:    00h, A7h
1002h:    00h, 1Ch
1004h:    00h, 00h
```

쉼표로 구분되어 있는 2바이트 데이터에서 왼쪽 부분은 지정된 주소에 저장되어 있는 데이터 값을 나타내며, 오른쪽에 있는 값은 그 다음 주소에 저장되어 있는 값을 의미합니다.

따라서 위의 세 줄은 다음과 같이 적을 수 있습니다.

```
1000h:    00h, A7h, 00h, 1Ch, 00h, 00h
```

앞에서 기술했던 곱셈 프로그램은 다음과 같이 표현할 수 있습니다.

```
0000h:    LOD A,[1005h]
          ADD A,[1001h]
          STO [1005h],A

          LOD A,[1004h]
          ADC A,[1000h]
          STO [1004h],A

          LOD A,[1003h]
          ADD A,[001Eh]
          STO [1003h],A

          JNZ 0000h

001Eh:    HLT

1000h:    00h, A7h
1002h:    00h, 1Ch
1004h:    00h, 00h
```

물론 빈 줄과 공백을 적절하게 사용하면 프로그램 전체를 한눈에 읽기 편하게 만들어 줄 수 있습니다.

주소는 경우에 따라 바뀔 수 있으므로 코드를 작성할 때는 실제의 숫자 주소를 사용하지 않는 것이 좋습니다. 예를 들어, 2000h번지에서 20005h번지에 숫자들을 저장하기로 결정했다면, 프로그램에서 많은 부분을 다시 작

성해야 할 것입니다. 따라서 숫자를 직접 적는 대신 메모리 위치를 참조할 수 있는 레이블을 이용하는 것이 더 좋겠지요. 이러한 레이블은 아래와 같이 간단한 단어 혹은 거의 단어와 비슷한 문자열을 사용합니다.

```
BEGIN:    LOD A,[RESULT + 1]
          ADD A,[NUM1 + 1]
          STO [RESULT + 1],A

          LOD A,[RESULT]
          ADC A,[NUM1]
          STO [RESULT],A

          LOD A,[NUM2 + 1]
          ADD A,[NEG1]
          STO [NUM2 + 1],A

  JNZ BEGIN

NEG1:     HLT
NUM1:     00h, A7h
NUM2:     00h, 1Ch
RESULT:   00h, 00h
```

위의 코드에서 NUM1, NUM2, RESULT와 같은 레이블들은 모두 2바이트가 저장되어 있는 메모리 주소를 나타냅니다. 이 문장에서 'NUM1+1', 'NUM2+1', 'RESULT+1'이라는 레이블은 그 레이블이 있는 주소 다음 바이트의 주소를 나타냅니다. HLT 명령어에 NEG1(negative one, 음수 1)이라는 레이블이 붙어 있다는 점도 주목하시기 바랍니다.

마지막으로 어떤 의도로 해당 문장을 적었는지 잊을 가능성이 있으므로, 실제 문장 옆에 세미콜론 기호(;)를 쓰고 나서 짧게 주석을 적을 수 있습니다.

```
BEGIN:    LOD A,[RESULT+1]
          ADD A,[NUM1+1]          ; 하위 바이트 더하기
          STO [RESULT+1],A

          LOD A,[RESULT]
          ADC A,[NUM1]            ; 상위 바이트 더하기
```

```
                STO [RESULT],A

                LOD A,[NUM2+1]
                ADD A,[NEG1]            ; 두번째 숫자로 감소시킴
                STO [NUM2+1],A

                JNZ BEGIN

NEG1:           HLT

NUM1:           00h, A7h
NUM2:           00h, 1Ch
RESULT:         00h, 00h
```

여기서 보여 드린 소스코드가 바로 프로그래밍 언어의 한 종류인 어셈블리어(assembly language)라는 컴퓨터 언어입니다. 이 언어는 16진수 숫자로만 구성된 기계어를 이용하는 방식과 여러 명령어를 사람이 사용하는 자연어인 영어로 장황하게 풀어쓰는 방식의 중간 정도의 방식으로, 명령을 사용할 때는 메모리 주소를 레이블로 표현하는 방식을 같이 사용합니다. 어셈블리어와 기계어를 혼동하시는 경우가 많은데, 사실 두 언어는 같은 것을 서로 다르게 표현한 차이가 있는 것이라 생각하시면 됩니다. 어셈블리어로 쓰인 모든 문장은 특정한 바이트의 기계어에 대응합니다.

이 장에서 만든 컴퓨터를 위한 프로그램을 작성하는 경우 프로그램을 일단 어셈블리어의 형태로 종이에 적어두어야 합니다. 적어둔 프로그램이 어느 정도 동작할 것 같고, 이를 테스트 해볼 준비가 되었다면 이제 손으로 작성한 어셈블리어를 기계어로 바꿀 때가 된 것이지요. 이와 같이 어셈블리어로 만들어진 프로그램을 이에 대응하는 기계어로 바꾸는 과정을 어셈블(assemble)이라 합니다. 이 시점에서 스위치를 이용하여 기계어 프로그램을 RAM에 입력하고, 프로그램을 구동(run)시킬 준비가 된 것입니다. 프로그램을 구동시킨다는 것은 입력한 명령어들을 순서대로 수행하게 만드는 것을 의미합니다.

컴퓨터 프로그래밍에 대한 개념을 배울 때, 버그(bug)에 대한 개념도 같이 배워두는 것이 좋습니다. 코드를 작성할 때, 특히 여기서와 같이 기계어로 코드를 작성할 때는 실수가 발생하기 쉽습니다. 데이터의 값을 잘못 입력하는 것도 별로 좋지 않지만, 명령어 코드를 잘못 입력한다면 어떤 일이 생길까요? 예를 들어, 실수로 로드 명령을 의미하는 10h 대신에 저장 명령을 의미하는 11h를 입력했다면, 원하는 숫자가 로드되지 않을 뿐만 아니라 누산기에 있는 임의의 값을 해당 주소에 덮어쓰게 됩니다.

몇몇 버그는 예상할 수 없는 결과를 초래합니다. 분기 명령어에서 점프할 곳에 유효한 명령이 존재하지 않는 경우나 저장 명령에 의해서 명령어 위치에 어떤 값이 저장되어 버리는 경우를 가정해 보세요. 도대체 어떤 일이 벌어질지 알 수 없지요.

우리가 만든 곱셈 프로그램에도 버그가 있습니다. 이 프로그램을 두 번 수행하는 경우, 두 번째 수행 과정에서는 A7h 곱하기 256이 수행되며, 그 결과는 이전에 연산된 결과에 더해지게 됩니다. 이미 프로그램이 한 번 수행된 후이기 때문에 1003h번지의 값은 0이 되어 있습니다. 두 번째 수행될 때는 이 값에 FFh가 더해지며, 그 결과가 0이 되지 않으므로 결과가 0이 될 때까지 계속하여 프로그램이 수행됩니다.

우리가 만든 컴퓨터에서 곱셈이 가능하다는 사실을 확인한 것과 같이 나눗셈도 역시 해낼 수 있습니다. 또한, 제곱근, 로그, 삼각 함수와 같은 원함수(primitive function)도 처리할 수 있을 것이라는 확신도 있습니다. 이러한 처리를 수행하기 위한 장치에서 필요로 하는 것은 적절한 프로그램을 수행할 수 있는 덧셈, 뺄셈, 조건 분기가 가능한 명령어와 하드웨어입니다. 이런 조건만 갖추어진다면 프로그래머들은 '소프트웨어를 이용해서 모든 것을 할 수 있다'고 말할 수 있을 것입니다.

물론, 이런 작업을 위한 소프트웨어는 상당히 복잡하겠지요. 많은 책에서

프로그래머가 어떤 문제들을 해결하는 데 사용하는 알고리즘들을 설명하고 있습니다. 물론 아직 이런 것까지 설명할 단계는 아닌 것 같습니다. 이제 겨우 숫자에 대해서 생각해 보았을 뿐이고 분수를 어떻게 표현할지에 대한 부분 등은 살펴보지도 않았으니까요. 이러한 부분은 23장에서 살펴보도록 하겠습니다.

이러한 장치를 구성하는 하드웨어들은 백 년도 넘은 과거에서부터 존재했었다고 앞에서 몇 번 이야기한 적이 있습니다. 하지만, 그 당시에 이 장에서 설명했던 형태의 컴퓨터가 만들어진 것은 아닙니다. 위의 컴퓨터를 만들 때 사용한 많은 개념이 1930년대 중반에 처음으로 릴레이를 이용한 컴퓨터가 구성되었을 때까지만 해도 존재하지 않았으며, 1945년경에 들어서야 서서히 이해되기 시작했습니다. 그때까지만 해도 컴퓨터를 만들려고 하는 사람들은 내부적으로 이진수가 아닌 십진수를 이용하려고 했습니다. 특히 초기의 컴퓨터에서 메모리는 엄청나게 크고 비쌌습니다. 지금도 마찬가지지만 64KB RAM을 만들기 위해서 커다란 릴레이 5백만 개를 사용한다는 것은 좀 어처구니가 없는 것이지요.

이제 우리가 지금까지 해온 일들과 연산과 계산 관련 장치들에 대한 역사에 대하여 되돌아볼 시간이 된 것 같습니다. 아마도 컴퓨터를 만들기 위해서 수많은 릴레이를 이용할 필요가 없다는 것은 아셨을 것입니다. 12장에서 이야기했던 것처럼 결과적으로 릴레이는 진공관이나 트랜지스터 같은 전자 장치로 대체되었습니다. 또한 우리가 설계한 프로세서나 메모리와 동일한 것을 누군가가 이미 손안에 들어올 수 있는 크기로 만들었다는 것을 알 수 있을 것입니다.