제 1 부

AVR ATmega128
마이크로컨트롤러

제 1 장 개요

현재의 세상은 디지털 세상이라 한다. 일상생활에서 사용하고 있는 많은 기기들이 디지털방식으로 바뀌고 있다. 아날로그 TV는 성능 좋은 디지털 TV로 바뀌고, LP 레코드는 디지털 방식을 쓰는 CD로, VHS 비디오는 DVD, 등등. 그렇지만 많은 사람들이 디지털을 사용하면 기기들이 왜 좋은 성능을 갖는 지는 정확히 알지 못한다. 그냥 디지털을 쓰면 아날로그보다 좋은 줄 안다.

디지털의 핵심은 단순성이다. 디지털 시스템에선 신호선 하나는 단 두 개의 상태를 갖는다. 두 상태를 1과 0으로 표시하기도 하고 True/False 또는 HIGH/LOW로 표시한다. 얼마나 간단한 표시방법인가. 그러면 이 간단한 신호로 무엇을 할 수 있을 까. 이런 신호선 2개를 사용하여 보자

신호 1	신호 2
0	0
0	1
1	0
1	1

$2^2=4$가지를 표현할 수 있다. 아직도 표현 가능한 개수는 좀 작아 보인다. 신호선 여러 개 사용할 때 표현 가능한 개수는 2의 누승으로 표현된다. 신호선 3개를 사용하면 $2^3=8$가지, 10개를 사용하면 $2^{10}=1024$가지, 20개를 사용하면 $2^{20}=1024\times 1024$로 약 100만 가지를 표현할 수 있다. 신호선이 늘어남에 따라 표현 가능한 개수는 기하급수적으로 늘어난다. 가장 간단한 신호를 조합함으로써 어마어마한 양의 정보를 표현할 수 있다.

디지털에서는 이와 같이 여러 신호선을 사용하여 매우 복잡한 정보를 표현한다. 그러나 아무리 많은 신호선을 사용하더라도 표시할 수 있는 정보의 수는 유한하다. 이는 각 정보를 유한한 숫자로 표시할 수 있다는 것이다. 디지털의 사전적 의미는 숫자이다. 즉 디지털은 모든 정보를 숫자로 표현하여 전달하는 것이다. 만약 4가지 정보를 전달하려면 신호선 2개, 100만 가지 정보를 전달하려면 신호선 20개를 사용하면 된다.[1)]

---

1) 20개의 신호선이 필요하다고 해서 20개의 신호선을 다 쓸 필요는 없다. 예를 들어 1개의 신호선을 사용하면서 시간을 나눠서 처음에는 신호선 1의 정보, 다음은 신호선 2의 정보 이런 식으로 20번에 걸쳐서 보내면 20개의 신호선을 사용하는 것과 같은 효과를 낼 수 있다. 단 시간이 좀 더 걸릴 뿐이다.

이에 비해 아날로그신호는 신호선 하나로 무한한 수를 표시한다. 그럼 왜 무한한 수를 표시할 수 있는 아날로그 신호보다 유한한 수를 표시할 수 있는 디지털 신호가 왜 우월 할까?

아날로그신호에서 전달되는 신호는 무한히 많은 것 중에 하나이므로 받는 곳에서는 이를 100% 정확하게 판독하는 것은 불가능하다. 약간의 오차가 있을 수밖에 없다. 그러나 디지털에서는 1/0만 사용하는 신호선을 여러 개 사용하지만 각 신호선에서는 보내는 정보가 1/0으로만 표시되므로 이것을 받는 곳에서 잘못 판독할 확률은 거의 없다. VHS 비디오테입과 DVD를 비교해 보라. 아날로그 신호를 담고 있는 VHS 비디오테입은 복사를 여러 번 할수록 화질이 떨어지지만 디지털 신호를 담고 있는 DVD는 아무리 복사를 하여도 화질이 떨어지지 않는다.

마이크로프로세서는 이러한 디지털 신호를 고속으로 처리하는 두뇌역할을 하는 장치이다. 디지털 신호를 처리하는 거의 모든 기기에는 마이크로프로세서가 들어있다고 생각하면 된다. 어린이용 장난감, TV, 냉장고, 에어컨 등 가정용기기에서부터 산업체의 생산 장비, 자동차, 선박, 항공기, 미사일, 인공위성 등 최첨단 시스템에 이르기 까지 거의 모든 시스템에서 마이크로프로세서가 핵심적인 장치로서 사용되고 있다.

마이크로프로세서가 보유한 능력이란 매우 단순하다. 고속으로 수의 연산을 처리하는 것이다. 능력을 어떻게 사용할 것인 가는 사람이 부여한다. 같은 마이크로프로세서로 DVD를 만들 수도 있고 인공위성의 제어장치를 만들 수도 있는 것이다.

마이크로프로세서가 어떤 일을 하도록 하는 것을 “마이크로프로세서를 프로그램한다.”라고 한다. 사람에게 일을 시킬 때 언어를 사용하는 것처럼 마이크로프로세서에게 일을 시킬 때도 언어를 사용한다. 이러한 언어를 컴퓨터 언어라 한다.(마이크로프로세서는 컴퓨터를 구성하는 주요요소이므로 마이크로프로세서와 컴퓨터가 구별 없이 사용되기도 한다.)

마이크로프로세서는 엄청나게 많은 종류가 있지만 기본적인 개념은 거의 흡사하므로 하나의 마이크로프로세서를 배우면 다른 마이크로프로세서를 접하더라도 쉽게 응용할 수 있다. 여기서는 마이크로프로세서로 산업현장에서 많이 쓰고 있는 Atmel사의 ATmega128을 선택하였으며 프로그램 언어로 C-언어를 선택하여 마이크로프로세서의 프로그램을 배워보도록 한다.

제 2 장 미리 알아두어야 할 일반적 사항

2.1 이진수(Binary Number), 16진수(Hexadecimal Number)

우리가 일반적으로 사용하는 숫자는 십진법 숫자이다. 십진법은 수를 셀 때 0~9까지 세고 십이 되면 한자리를 올려서 표시하는 방법을 말한다. 이런 방법으로 0~9의 수를 나열하면 모든 수를 표시할 수 있다. 십진수는 십진법을 사용하여 표시한 수를 말한다. 십진법은 수를 표시하는 한 가지 방법일 뿐이다. 0~1까지 세고 2가 되면 한자리를 올리는 방법으로 수를 표시할 수도 있다. 2가 되면 한자리를 올리므로 이런 방법을 이진법이라 하고 이 방법으로 표시한 수를 이진수라 한다.

십진법이 사용된 것은 사람의 손가락이 10개이므로 사용하는 데 매우 익숙하기 때문이다. 마찬가지로 0,1 두 종류를 신호의 기본으로 하는 디지털 세계에서는 2진법을 사용하는 것이 자연스러운 선택이다. 여기서는 마이크로프로세서에서 많이 사용하는 2진수, 10진수 및 16진수에 대해 설명한다.

▸ 십진수(Decimal Number)

각 자리의 수자로 0 ~ 9까지 사용한다. 십진수로 표시된 수 123은 다음과 같은 의미를 갖는다.

$$123 = \underbrace{1\times10^{2}}_{\text{100자리}} + \underbrace{2\times10^{1}}_{\text{10자리}} + \underbrace{3\times10^{0}}_{\text{1자리}}$$

각 자리는 10의 누승(10^0, 10^1, 10^2, . . .)으로 표시된다.

▸ 2진수(Binary Number)

각 자리의 수로 0,1만을 사용한다. 2진수로 표시된 1011_2는 다음과 같은 의미를 가지며, 십진수로 환산하면 11과 같다.

$$1011_2 = 1\times2^{3} + 0\times2^{2} + 1\times2^{1} + 1\times2^{0}$$
$$= 8 + 0 + 2 + 1 = 11 \text{ (십진수)}$$

위에서 아래첨자 “2”는 2진수를 나타낸다. 각 자리는 2의 누승(2^0, 2^1, 2^2, . . .)으로 표시된다.

▸ 16진수(Hexadecimal Number)

각 자리의 수로 0,1,. . . ,9,A,B,C,D,E,F를 갖는 수이다. 여기서 10이상의 수는 십진수로 나타낼 수 없어 A ~ F로 나타낸다. A ~ F는 십진수로 환산하여 10 ~ 15를 나타낸다. 16진수로 표시된 숫자 $AB38_{16}$은 다음과 같은 의미를 가지며 십진수로 43832이다.

$$AB38_{16} = 10\times16^3 + 11\times16^2 + 3\times16^1 + 8\times16^0$$
$$= 10\times4096 + 11\times256 + 3\times16 + 8\times1$$
$$= 43832\ (\text{십진수})$$

위에서 아래첨자 “16”은 16진수를 나타낸다. 각 자리는 16의 누승(16^0, 16^1 , 16^2 ,16^3 , . . .)으로 표시된다.

수를 어떤 진법을 사용하여 표시하여도 연산은 십진법과 같은 방법으로 수행한다.

예)

십진수	이진수	16진수
1 <== 올림	11 <== 올림	1 <== 올림
151	011_2	$A8_{16}$
+ 63	+ 110_2	+ $B18_{16}$
214	1001_2	$BC0_{16}$

예)

십진수	이진수	16진수
10 <== 빌림	2 <== 빌림	16 <== 빌림
71	10_2	$A8_{16}$
- 63	- 01_2	- 19_{16}
8	01_2	$8F_{16}$

※ 올림은 영문으로 Carry, 빌림은 Borrow라 한다.

2.2 데이터의 단위

앞에서도 언급한 바와 같이 디지털 세계에서는 하나의 신호선이 0 또는 1(False 또는 True)의 두 가지 경우만 표시하므로 신호를 표시하기 위해서는 2진수를 사용하는 것이 가장 편리하다. 따라서 데이터의 단위는 2진수를 사용하여 나타낸다.

비트(bit) : 이진수(Binary digits)의 약자로서 0 또는 1을 가진다.

니블(nibble) : 4비트를 니블이라 부른다. 니블로 2^4 = 16개의 수를 표시할 수 있으므로 1자리의 16진수를 표현할 수 있다.

바이트(byte) : 8비트를 바이트라 부른다. 이는 데이터의 크기를 표시할 때 가장 기본적인 단위이다. 1바이트로 2^8 = 256개의 수를 표시할 수 있다.

워드(word) : 2바이트를 워드라 부른다. 1워드로 2^{16} = 65536개의 수를 표시할 수 있다.

패러그래프(Paragraph) : 16바이트를 말한다.

페이지(Page) : 256바이트를 말한다.

Kbyte : 일반적으로 Kilo는 10^3 = 1000배를 뜻하나, 컴퓨터에서 Kilo는 2^{10}=1024 배를 가리킨다. 따라서 1Kbyte = 1024byte 이다.

Mbyte : 일반적으로 Mega는 $10^6 = 10^3 \times 10^3$ = 1000000배를 뜻하나, 컴퓨터에서는 $2^{20} = 2^{10} \times 2^{10}$배를 가리킨다. 1Mbyte=1024 Kbyte이다.

Gbyte : 일반적으로 Giga는 $10^9 = 10^3 \times 10^3 \times 10^3$ 배를 뜻하나, 컴퓨터에서는 $2^{30}=2^{10} \times 2^{10} \times 2^{10}$배를 가리킨다. 1Gbyte = 1024Mbyte 이다.

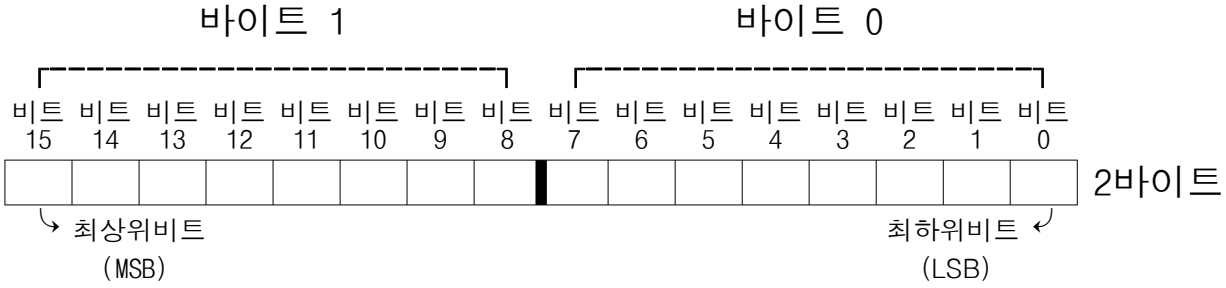
Kilo	: 10^3을 나타낸다.	접두어로 K를 쓴다.	예) Kg, KΩ, KHz
Mega	: 10^6을 나타낸다.	접두어로 M을 쓴다.	예) MΩ, MHz
Giga	: 10^9을 나타낸다.	접두어로 G을 쓴다.	예) GHz
milli	: 10^{-3}을 나타낸다.	접두어로 m를 쓴다.	예) msec, mm, mA
micro	: 10^{-6}을 나타낸다.	접두어로 μ를 쓴다.	예) μsec, μm, μA
nano	: 10^{-9}을 나타낸다.	접두어로 n를 쓴다.	예) nsec, nm, nF

2.3 데이터의 표시

컴퓨터에서 메모리의 기본단위는 바이트로 주어지므로 수를 표시할 때는 2의 누승바이트인 2^0=1바이트, 2^1=2바이트, 2^2=4바이트, 2^3=8바이트 등을 사용한다. 바이트를 구성하는 기본요소인 비트는 다음 그림과 같이 정의된다.

1바이트 수 :

2바이트 수 :



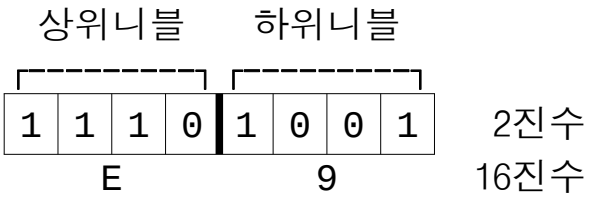
- 비트 또는 바이트를 셀 때는 우측 최하위자리를 0으로 하여 왼쪽으로 세어 나간다.
- 비트 0은 항상 최하위비트이며 이를 LSB(Least Significant Bit)로 표시한다.
- 1바이트(8비트) 수에서는 비트 7이 최상위비트이고 2바이트(16비트) 수에서는 비트 15가 최상위비트이다. 최상위비트는 MSB(Most Significant Bit)로 표시한다.
- 4바이트 수와 8바이트 수에도 같은 방법이 적용된다.

앞에서 말한 바와 같이 컴퓨터에서 데이터의 표시는 2진수로 표시하는 것이 바람직하나 숫자를 표시할 때 자리수가 매우 커져 사용하기에 불편하다. 4자리의 2진수는 다음과 같이 1자리의 16진수로 표시가 가능하므로 컴퓨터에서는 16진수를 많이 사용한다.

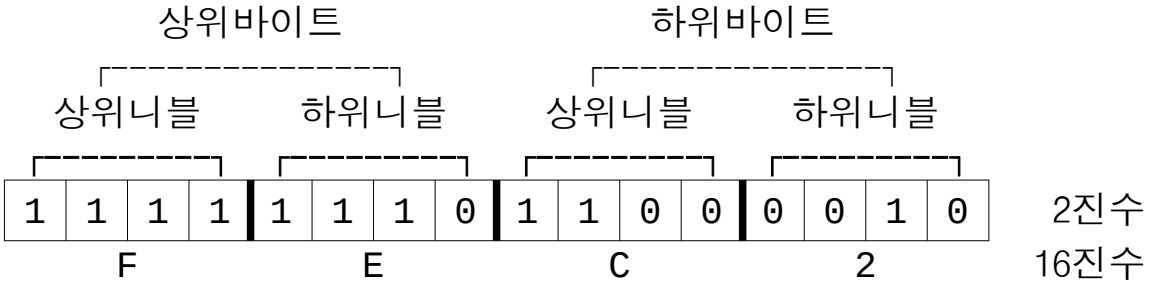
2진수		16진수	2진수		16진수
0000	==>	0	1000	==>	8
0001	==>	1	1001	==>	9
0010	==>	2	1010	==>	A
0011	==>	3	1011	==>	B
0100	==>	4	1100	==>	C
0101	==>	5	1101	==>	D
0110	==>	6	1110	==>	E
0111	==>	7	1111	==>	F

한 바이트로 표시된 수는 8자리의 2진수이므로 이를 16진수로 표시할 때는 니블 단위로 끊어서 표시하면 된다.

(예) 2진수 11101001_2를 16진수로 표시하면 다음과 같이 $E9_{16}$가 된다.



2진수 1111111011000010_2를 16진수로 표시하면 $FEC2_{16}$가 된다.



2.4 보수를 사용한 음수의 표시

컴퓨터에서 신호는 1과 0만을 사용하기 때문에 양수만을 표현한다. 음수를 표현하기 위한 다른 신호를 사용하지 않는다. (만약 음/양을 표시하려면 하나의 신호상태가 더 필요할 것이다.) 그러면 컴퓨터에서 음수를 어떻게 표시하는 지 살펴보자.

십진수에서 한자리 수의 십의 보수(Ten's Compliment)는 10에서 수를 뺀 수를 말한다. 3의 십의 보수는 10-3 = 7이다. 보수를 사용한 다음 계산을 보자.

```
      9          9
-     3      +   7      <== 3의 10의 보수
-------    -------
      6         16
                ↳ 2자리수의 올림을 무시하면 왼쪽계산과 동일
```

두 자리 십진수의 뺄셈을 고려해보자. 두 자리 십진수의 보수는 100에서 뺀 수이다.

```
     29         29
-     3      +  97      <== 03의 10의 보수
-------    -------
     26        126
                ↳ 3자리수의 올림을 무시하면 왼쪽계산과 동일
```

위와 같이 보수를 사용한 덧셈을 한 후 연산 자리수보다 한자리수 위의 올림을 무시하면 뺄셈의 결과와 같다. 이와 같이 보수를 사용하여 뺄셈을 수행할 수 있다. 보수 계산에서 주의하여야 할 것은 사용하는 자리수에 따라 보수가 다르다는 것이다. 위의 예와 같이 십진수 3에 대해 십의 보수는 사용하는 자리수에 따라

	십진수 3의 보수
1자리 계산 :	10-3 = 7
2자리 계산 :	100-3 = 97
3자리 계산 :	1000-3 = 997

. . .

와 같이 계산된다. 이는 모든 진법에 대해 적용된다.

앞 예의 한자리수 연산 9-3을 세자리 수를 사용한 연산을 고려하면 다음과 같다. 단 세자리 연산을 하였으므로 4자리의 올림을 무시한다.

```
      9            9
-     3      +   997      <== 3의 10의 보수
-------      -------
      6         1006
                ↳↳4자리수의 올림을 무시하면 왼쪽계산과 동일
```

즉 자리수에 따라 보수를 달리 사용하여야 함을 알 수 있다.

다음은 16진수의 뺄셈을 고려해보자. 16진수 $7A_{16}$의 16의 보수는 다음과 같이 계산된다.

16진수 $7A_{16}$의 보수

2자리 계산 : $100_{16} - 7A_{16} = 86_{16}$

3자리 계산 : $1000_{16} - 7A_{16} = F86_{16}$

4자리 계산 : $10000_{16} - 7A_{16} = FF86_{16}$

. . .

3자리의 16진수 뺄셈 $88_{16}-7A_{16}$을 하여보면

```
      88₁₆           88₁₆
-     7A₁₆     +   F86₁₆      <= 16의 보수
----------     ----------
      0E₁₆        100E₁₆
                  ↳↳4자리의 올림을 무시하면 왼쪽계산과 동일
```

이진수의 뺄셈을 고려해보자. 이진수의 2의 보수(Two's Compliment) 역시 십진법과 유사하게 계산된다. 8자리 연산을 위하여 2진수 00101100_2의 보수는

101100_2의 보수 : $100000000_2 - 101100_2 = 11010100_2$

이다. 하지만 2의 보수는 좀 더 쉽게 계산할 수 있다. 이진수에서 0은 1

로 1은 0으로 바꾼다. 이를 1의 보수(One's Compliment)라 한다. 1의 보수에 1을 더하면 2의 보수를 얻는다.

00101100_2 ==> 11010011_2 : 1의 보수
+ 1

11010100_2 : 2의 보수

※ 101100_2의 1의 보수를 취할 때 8자리를 모두 채운 00101100_2에 대해 1의 보수를 취하여야 한다.

8자리 이진수의 뺄셈과 보수를 사용한 덧셈을 고려하여 보자.

11001001_2 11001001_2
- 00101100_2 + 11010100_2 <= 2의 보수

10011101_2 110011101_2
↳9자리수의 올림을 무시하면 왼쪽과 동일

9자리의 올림을 무시하면 보수를 사용한 덧셈과 뺄셈의 결과가 같다.

이상으로 보수를 사용한 덧셈으로 뺄셈을 수행할 수 있음을 알 수 있다. 컴퓨터에서는 보수를 사용하여 음수를 표현한다. 단 앞에서 언급한 바와 같이 보수는 사용하는 자리 수에 따라 달라짐을 유의하여야 한다. 컴퓨터에서 -3의 표현을 보자.

1바이트 수일 때 : 3을 1바이트 2진수로 표시하면 00000011_2
2의 보수 : 11111101_2 ==> -3을 1바이트 수로 표시
16진수로 표시하면 FD_{16}

2바이트 수일 때 : 3을 2바이트 2진수로 표시하면 0000000000000011_2
2의 보수 : 1111111111111101_2 ==> -3을 2바이트로 표시
16진수로 표시하면 $FFFD_{16}$

같은 음수라도 사용 바이트의 수에 따라 완전히 다름을 알 수 있다.

1 바이트로 부호 있는 수를 표현하여 보자. -1은 보수로 표현되므로 1 바이트 이진수로 표시하면 11111111_2이다.(이는 부호 없는 수일 때 십진수 255를 표시한다.) 따라서 부호가 있을 때는 십진수로 -128 ~ 127까지 표시할 수 있다.

음수 : $10000000_2(-128)$ ~ $11111111_2(-1)$
양수 : $00000000_2(0)$ ~ $01111111_2(127)$

2 바이트로 부호 있는 수를 표현하면

음수 : $1000000000000000_2(-32768)$ ~ $1111111111111111_2(-1)$
양수 : $0000000000000000_2(0)$ ~ $0111111111111111_2(32767)$

☞ 음수는 최상위비트가 1이고 양수는 0임을 알 수 있다. 즉 최상위비트를 보면 음수/양수를 판별할 수 있다. 이런 이유로 최상위비트를 **부호비트(Sign Bit)**라고 한다 : 1바이트 수일 때는 비트 7이, 2바이트 수일 때는 비트 15가 부호비트이다.

2.5 오버플로와 언더플로

두 개의 부호 없는 1바이트수의 덧셈을 고려해보자. 200 + 58 = 258, 그러나 258은 1바이트로 표시할 수 없는 수이다.

258 = 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 $_2$
↳1바이트를 벗어남

연산 200+58의 결과에서 1바이트를 벗어나는 비트는 무시되어 200+58 = 2인 잘못된 연산 결과를 얻는다. 이와 같이 덧셈의 결과가 수를 표현하기 위해 사용한 바이트로 표시할 수 없어 결과가 틀리게 나오는 것을 오버플로(Overflow)라 한다.

두 개의 부호 없는 1바이트수의 뺄셈을 고려해보자. 1 - 2 = -1, 그러나 -1은 부호 없는 1바이트 수로 표시할 수 없는 수이다. -1은 보수로 표시되므로

-1 = | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |$_2$

부호 없는 1바이트 수에서 11111111_2는 255이므로 결과적으로 1-2 = 255인 잘못된 연산이 수행된다. 이와 같이 뺄셈의 결과가 수를 표현하기 위해 사용한 바이트로 표시할 수 없어 결과가 틀리게 나오는 것을 언더플로(Underflow)라 한다.

오버플로와 언더플로가 발생하는 수는 사용하는 바이트 그리고 부호 있음과 없음에 따라 서로 다르다.

오버플로 발생시점

부호 없는 1바이트 수 덧셈 결과가 255(FF_{16})보다 클 때
부호 있는 1바이트 수 덧셈 결과가 127($7F_{16}$)보다 클 때
부호 없는 2바이트 수 덧셈 결과가 65535($FFFF_{16}$)보다 클 때
부호 있는 2바이트 수 덧셈 결과가 32767($7FFF_{16}$)보다 클 때

언더플로 발생시점

부호 없는 1바이트 수 뺄셈 결과가 0(00_{16})보다 작을 때
부호 있는 1바이트 수 뺄셈 결과가 -128(80_{16})보다 작을 때
부호 없는 2바이트 수 뺄셈 결과가 0(0000_{16})보다 작을 때
부호 있는 2바이트 수 뺄셈 결과가 -32768(8000_{16})보다 작을 때

2.6 마이크로컴퓨터의 구조

일반적 마이크로컴퓨터의 구조는 그림 2.1과 같이 3대요소로 나뉜다.

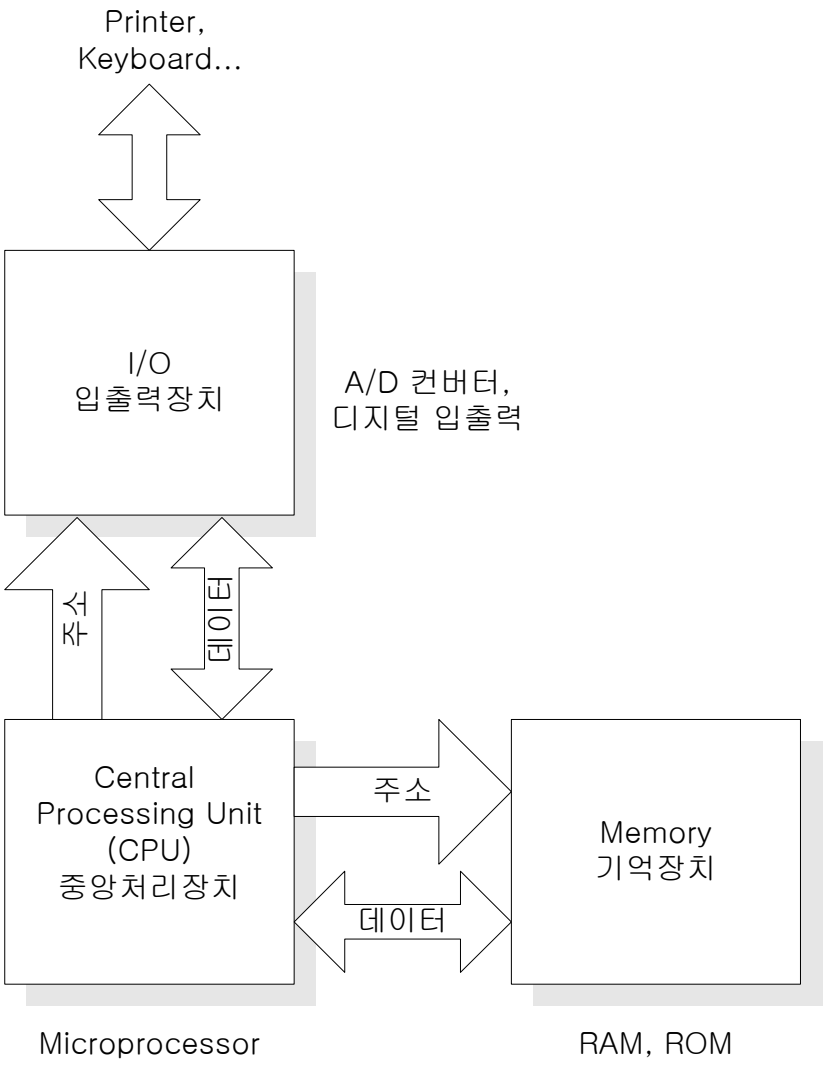


그림 2.1 일반적 마이크로컴퓨터의 구조

■ **중앙처리장치(Central Processing Unit, CPU)** : 마이크로프로세서라고도 하며, 연산(사칙연산, 비교 등)을 수행하는 장치로서 내부에는 연산에 필요한 제한된 개수의 레지스터(일종의 메모리)를 가지고 있다. 일반적인 마이크로프로세서(Pentium Processor 등)는 많은 양의 데이터를 저장할 메모리를 가지고 있지 않을 뿐 더러, 입출력을 위한 장치(디지털 입출력장치, 타이머 등)를 가지고 있지 않다. 따라서 마이크로컴퓨터를 구성하기 위해서는 마이크로프로세서 외부에 메모리와 입출력장치를 장착하여야 한다.

■ **메모리(Memory)** : 프로그램과 데이터를 저장하는 장치이다. 프로그램은 CPU에 내리는 명령의 집합으로 메모리에 저장된다. CPU는 메모리에 있는 명령어를 읽고, 이에 따라 작업을 수행한다. 프로그램 수행에 필요한 데이터 역시 메모리에 저장된다. 메모리는 크게 두 가지로 분류된다.

RAM(Random Access Memory) : 읽기/쓰기가 가능한 메모리로서 전원이 꺼지면 내용이 사라진다.

ROM(Read Only Memory) : 읽기만 가능한 메모리로서 전원이 꺼지더라도 내용이 보존된다.

PC에서는 대부분이 RAM이며, ROM은 아주 작은 부분을 차지한다. ROM은 전원이 꺼지더라도 내용이 남아 있으므로, 전원을 켰을 때 처음 하는 작업은 ROM에 저장한다. PC의 BIOS는 전원을 켰을 때 하드디스크에 들어있는 Windows를 메모리에 옮기는 작업을 하는 프로그램으로 ROM에 들어있다.(이와 같은 작업을 부팅이라 한다.)

■ **입출력장치(Input/Output Device)** : 외부장치로부터 입력을 받거나, 외부장치에 데이터를 출력하는 장치이다. 예로서 A/D 컨버터, 디지털 입출력장치 등이 있다.

마이크로프로세서를 말할 때 흔히 8비트 또는 16비트 마이크로프로세서라는 용어를 쓴다. 이 때 8비트 마이크로프로세서라 함은 CPU에서 연산을 처리할 때 기본적인 단위가 1바이트(8비트)임을 나타낸다. 8비트 마이크로프로세서의 기본연산단위는 1바이트이므로 2바이트 연산을 위해서는 2번의 계산이 필요하게 된다. 그러나 16비트 마이크로프로세서의 기본연산단위는 2바이트이므로 1번의 계산으로 2바이트연산을 수행할 수 있어 8비트 마이크로프로세서보다는 빠른 처리속도를 보인다. 참고로 현재 PC에서 사용되고 있는 Pentium은 64비트(8바이트) 마이크로프로세서이다.

마이크로프로세서의 또 하나의 사양으로 CPU 클록 속도가 있다. CPU의 모든 작동을 클록에 연동되어 수행하므로 클록 속도가 빠르면 당연히 수행속도가 빨라진다. 그러나 CPU마다 장착할 수 있는 최대 클록속도는 제한된다. 참고로 **ATmega128**은 최대 16MHz 클록을 사용하고 현재 PC의 Pentium CPU에는 수 GHz 클록을 사용한다.

2.7 메모리

메모리는 데이터를 읽고 쓸 수 있느냐 읽기만 할 수 있느냐에 따라 읽고 쓰기 메모리(일반적으로 RAM이라고 하나 정확한 표현은 Read Write Memory가 맞다)와 읽기전용 메모리(ROM: Read Only Memory)로 나뉜다. 그림 2.2는 기억장치로 사용되고 있는 메모리의 종류를 보여준다.

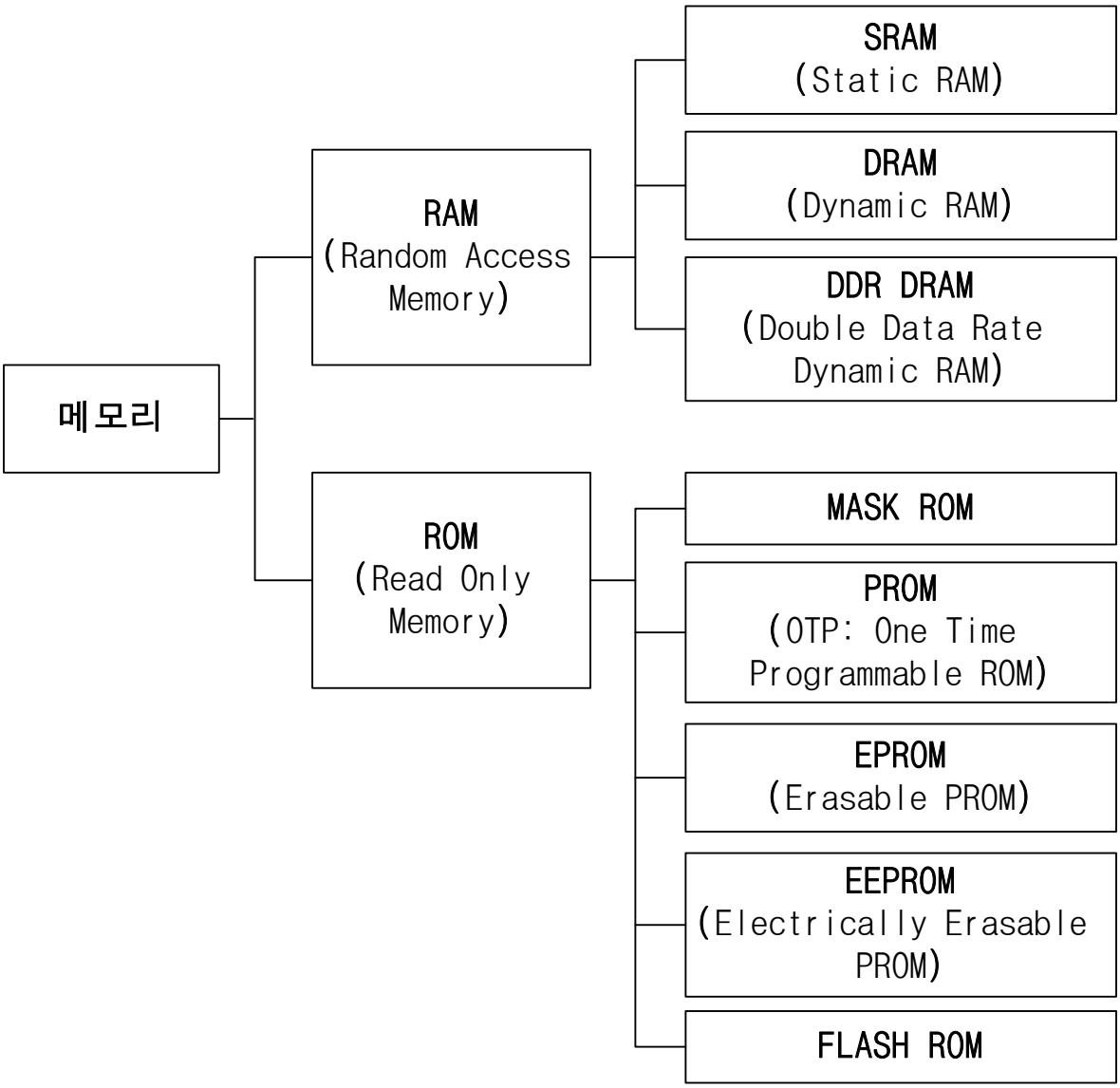


그림 2.3 메모리의 종류

(1) RAM(Random Access Memory)

RAM은 전원이 꺼지면 기억이 소멸되는 메모리이므로 휘발성메모리(Volatile Memory)라고도 한다.

SRAM(Static RAM)

전원이 투입되어 있는 동안 기억을 유지하는 메모리이다. 사용이 간단한 장점은 있지만 각각의 기억 비트(기억장소의 최소 단위로 cell이라 한다)를 구성하는데 여러 개의 반도체 소자가 사용되어 한 메모리 칩 당 구성할 수 있는 기억용량이 적다는 단점이 있다. 많은 용량의 메모리가 필요하지 않는 제어시스템에서는 주로 SRAM을 사용한다.

DRAM(Dynamic RAM)

이것은 메모리의 한 셀을 구성하는데 단 한 개의 반도체 소자를 사용하기 때문에 같은 면적의 칩에 많은 용량의 기억장소를 구성할 수 있어서 주로 대용량의 메모리 시스템을 구축하는데 사용되지만 전원이 투입되어 있어도 일정시간(보통 4 msec)이 경과 되면 저절로 기억내용이 소멸되기 때문에 주기적으로 기억내용을 다시 충전(refresh라 한다)시켜야 하는 단점이 있다. 이 리프레시를 위해 메모리 칩 외부에 리프레시를 위한 별도의 복잡한 회로가 필요하여 간단한 시스템에서는 사용하기 힘들고 주로 대용량의 메모리가 필요한 컴퓨터 시스템에서 사용된다. PC의 주기억장치는 거의 전부 이 DRAM을 사용하고 있다.

(2) ROM(Read Only Memory)

전원이 꺼져도 기억이 소멸되지 않는 메모리이다. 비휘발성메모리(Non-volatile Memory)라고도 한다.

마스크(mask) ROM

ROM 제조 시 프로그램 코드에 맞게 내부 회로를 고정시켜 제작하는 것으로서 읽기만 가능하고 더 이상의 지우기와 쓰기는 불가능하다. 메모리 제조회사에서 대단위로 판박이처럼 만들기 때문에 가격이 저렴하지만 소량의 주문은 받지 않아 소규모 생산에는 적합하지 않다.

PROM(Programmable ROM) : OTP(One Time Programmable)

내부 메모리 셀들에 전기적으로 녹일 수 있는 퓨즈를 달아 두고 외부에서 프로그래밍을 통해 퓨즈를 녹일 수 있으며 해당 비트의 퓨즈가 녹아떨어지면 1, 그대로 있으면 0이 기억되는 식으로 단 한 번의 프로그래밍이 가능하다. 일단 프로그램이 써 지면 ROM 외부에서의 어떤 작업으로도 내부에 있는 녹은 퓨즈를 다시 붙일 수가 없기 때문에 다시 지우고 쓰기가 불가능하다. 이런 특성 때문에 OTP라고도 하며 가격이 싸서 마스크 ROM을 제작할 수 없는 소량의 제품 개발 시 주로 사용된다.

EPROM(Erasable PROM)

ROM 모듈에 투명한 창이 달려 있으며 이 창에 자외선을 쪼이면 내부 메모리 셀들의 기억이 지워지고 초기 상태로 돌아가서 새로운 쓰기가 가능하다. 이런 지우기/쓰기가 수십만 번 반복적으로 가능하여 과거에 주로 사용되었던 형태이다.

EEPROM(Electrically Erasable Programmable ROM)

위의 EPROM은 기존의 기억을 지우기 위해 기판에 삽입된 ROM을 빼내서 전용의 자외선소거기에 넣어 기억을 지운 후 다시 전용의 쓰기 장치(ROM Writer라 한다)에 넣어 새로운 프로그램을 써 넣어야 하는 불편한 점이 있었다. 반면 EEPROM은 ROM이 기판에 삽입된 상태로 전기적인 신호에 의해 기억을 지울 수 있고 새로운 내용을 써 넣을 수 있어서 아주 편리하다.

FLASH 메모리

EEPROM의 한 종류로서 요즘 가장 널리 사용되는 ROM 메모리이다. EEPROM과 마찬가지로 회로에 삽입된 채로 지우고 쓸 수가 있고 특히 메모리 전체 내용을 한 순간에 지우고 다시 쓸 수 있어서 종래의 EEPROM보다 속도가 훨씬 빠르다. 종래의 EEPROM은 한 순간 한 바이트 씩 지우거나 쓸 수 있었다.

☞ ROM은 한 번 프로그램하면 거의 변경하는 일이 없다. 이런 의미로 흔히 하드웨어와 소프트웨어의 중간 개념인 펌웨어(firmware)라 한다.

2.8 CPU와 메모리의 데이터 교환

■ **주소(address)** : 메모리에 데이터를 쓰려면 저장할 장소의 위치를 지정하여야한다. 마찬가지로 메모리의 데이터를 읽어 오려면 읽어 올 데이터가 있는 위치를 지정하여야 한다. 이러한 메모리의 위치를 주소라 한다. 메모리의 기본 단위는 바이트이므로 바이트별로 주소가 지정된다. 메모리의 주소는 숫자로 표시되며 2진수로 표시된 수가 그림 2.1의 주소버스를 통해 메모리에 전달된다.(버스는 다수의 선을 말하며 주소를 지정하기 위해 다수의 선이 나가므로 주소버스라 부른다.) 이 때 주소버스의 크기, 즉, 비트수가 CPU에서 지정할 수 있는 메모리의 주소의 최대크기를 정한다.

예) 주소버스가 16bit일 때 지정 가능한 주소는 0000_{16}번지부터 $FFFF_{16}$번지로서 메모리 크기로는 2^{16} byte = $2^6 \times 2^{10}$ byte = 64 Kbyte이다. 따라서 장착할 수 있는 메모리의 최대크기는 64Kbyte이다.

예) 주소버스가 32bit이면 장착 가능한 메모리 크기는

$$2^{32} \text{ byte} = 2^2 \times 2^{10} \times 2^{10} \times 2^{10} \text{ byte} = 4 \text{ Gbyte}$$

이다.

■ **주소 디코딩** : CPU와 메모리 또는 입출력장치간의 데이터교환은 그림 2.1에서 데이터버스를 통하여 이루어진다. 이 데이터버스는 모든 메모리와 입출력장치가 공유하므로, CPU가 지정한 장치만이 데이터버스를 사용하도록 교통정리가 필요하다. 이는 주소를 사용하여 이루어지며, CPU가 주소를 지정하면 해당주소를 가진 장치만 데이터버스를 사용하도록 회로를 구성한다. 이를 주소 디코더(Address Decoder)이라 한다.

■ **데이터교환** : 메모리에서 데이터를 읽어 들일 때는 CPU는 주소버스를 통해 메모리의 위치를 지정하게 되고, 주소 디코더에 의해 해당 메모리는 데이터버스에 데이터를 보낸다. CPU는 데이터버스의 내용을 읽게 된다. 메모리에 데이터를 쓸 때는 CPU는 메모리의 위치를 주소버스로 전달하고, 데이터 버스에 데이터를 보낸다. 해당 메모리는 데이터버스의 데이터를 받아 저장한다.

2.9 CPU와 입출력장치간의 데이터 교환

■ I/O mapped I/O : CPU와 입출력장치간의 데이터 교환 역시 메모리와의 데이터 교환과 마찬가지로 주소버스와 데이터버스를 사용하여 이루어진다. 여기서 사용되는 주소버스 및 데이터버스는 메모리 역시 사용하게 되므로 현재 CPU가 수행하는 데이터교환 대상이 입출력장치인지 메모리인지를 구분하기 위해 또 하나의 신호선을 사용한다. 이와 같이 입출력장치의 주소와 메모리의 주소를 구분하여 사용하는 입출력장치를 I/O mapped I/O라고 한다.

■ Memory mapped I/O : 메모리와 입출력장치는 기능은 서로 다르지만 CPU의 입장에서 보면 주어진 주소에 데이터를 쓰기/읽기를 한다는 면에서는 동등하다. 따라서 입출력장치를 하나의 메모리로 간주할 수 있다. 이와 같이 메모리와 입출력장치의 주소를 구분하지 않고 하나의 주소영역으로 사용하며, 메모리 주소범위의 일정부분에 입출력장치의 주소를 할당하여 사용하는 것을 Memory mapped I/O라고 한다.

2.10 메모리 엑세스의 구조

메모리 엑세스를 방법에는 기본적인 두 가지 구조(Architecture)가 있다.

■ Von Neumann Architecture

미국의 수학자 John Von Neumann이 제안한 구조로 그림 2.5와 같이 프로그램과 데이터를 하나의 메모리에 저장한다. 프로그램과 데이터는 같은 데이터 버스와 주소버스를 사용한다. PC는 이 구조를 사용한다.

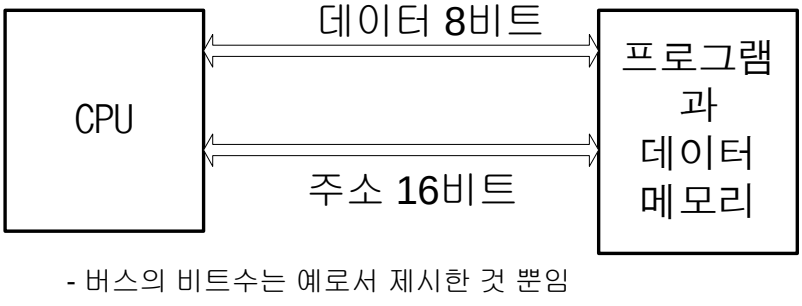


그림 2.4 Von Neumann Architecture

■ Harvard Architecture

그림 2.6과 같이 프로그램과 데이터를 각각 물리적으로 다른 메모리에 저장한다. 프로그램과 데이터는 서로 다른 데이터 버스와 주소버스를 사용하므로 프로그램과 데이터에 대해 다른 비트수를 사용할 수 있다. 또한 프로그램과 데이터를 동시에 읽어 들일 수 있다. 제어용 마이크로프로세서에서는 이 구조를 많이 쓰고 있다. 여기서 다루는 ATmega128 역시 이 구조를 사용한다.

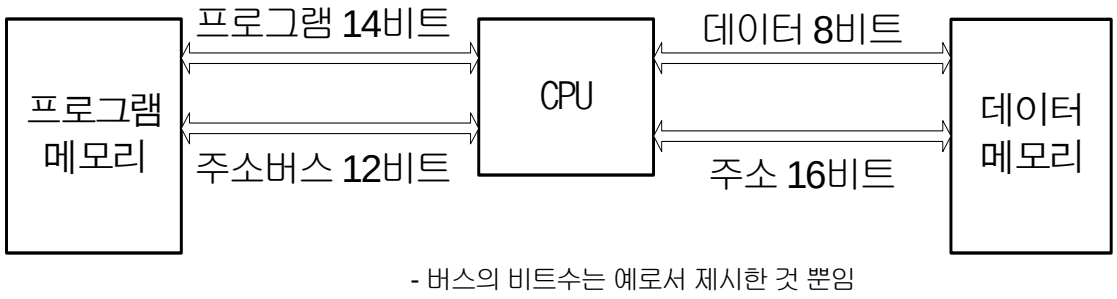


그림 2.5 Harvard Architecture

2.11 프로그램의 수행 및 프로그램 언어

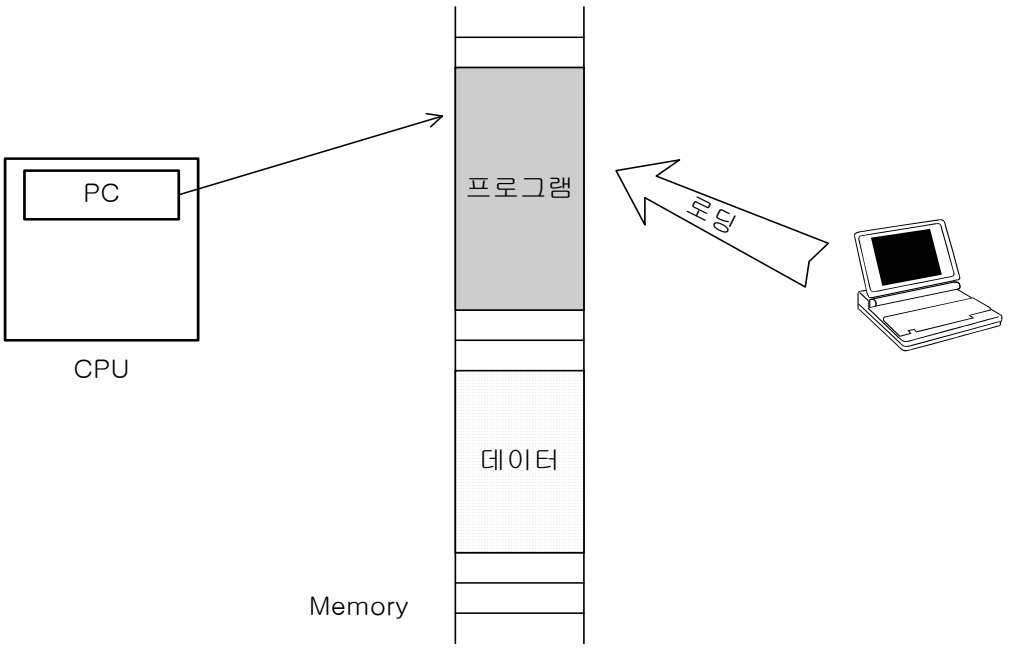


그림 2.7 프로그램의 로딩 및 수행

■ **프로그램의 수행**: 프로그램은 CPU가 할 명령어들의 집합이며 프로그램과 프로그램 수행에 필요한 데이터는 그림 2.7과 같이 메모리에 저장된다. CPU 내에는 Program Counter(**PC**)라는 레지스터가 있어 수행하여야 할 명령어가 위치한 메모리의 가리킨다. CPU는 이 **PC**가 가리키는 명령어를 읽어서 수행한다. 명령어의 수행이 끝나면 **PC**는 자동적으로 다음 명령어가 있는 메모리를 가리킨다. 따라서 프로그램을 수행하기 위해서는 **PC**를 프로그램의 초기위치를 가리키도록 하면 된다.

■ **프로그램의 로딩 및 수행** :

- PC와 같은 일반적인 컴퓨터에서는 여러 종류의 프로그램을 필요에 따라 수행하여야 하므로, 모든 프로그램을 미리 메모리에 저장할 수가 없다. 따라서 필요에 따라서 프로그램을 메모리에 로딩을 하고 수행시켜야 한다. 보통의 프로그램은 하드디스크에 저장되어 있다. 윈도우즈에서는 해당 아이콘 더블클릭하면 해당 프로그램을 하드디스크에서 읽어들여 메모리에 로딩을 하고 Program Counter 레지스터를 프로그램의 초기위치로 이동시켜 프로그램을 수행한다. 이러한 일을 수행하는 윈도우즈 같은 프로그램을 운영체제(Operating system)라고 한다.
- 제어용 마이크로컴퓨터인 경우 하나의 전용 프로그램이 수행된다. 따라서 프로그램을 메모리에 미리 저장해 놓는다. 이 때 전원이 꺼지더라도 프로그램이 없어지지 않도록 메모리는 ROM을 사용한다.
- 그러나 프로그램을 개발할 때는 프로그램이 계속 변경되므로 이 방법을 사용할 수 없다. 이 경우 프로그램 개발은 개인용 컴퓨터에서 하고 이를 마이크로컴퓨터의 메모리에 다운로드한 다음 수행한다. 그러나 ROM으로 FLASH 메모리를 사용하기 전에는 ROM에 프로그램을 저장하는 것은 ROM Writer를 사용하여야 하는 매우 번거로운 작업이었다. 그래서 개발 중에는 프로그램을 RAM에 저장하고 개발이 끝나면 ROM Writer로 완성된 프로그램을 ROM에 굽는 방법을 사용하였다. 그러나 FLASH 메모리를 ROM으로 사용하면서 PC에서 손쉽게 프로그램을 FLASH에 다운로드를 할 수 있게 되었다. **ATmega128**은 In System Programming(ISP)기술을 사용하여 프로그램을 **ATmega128**의 FLASH 메모리에 직접 프로그램을 다운로드 하고 즉시 프로그램을 수행할 수 있다.

■ 프로그램 언어

마이크로프로세서에게 작업을 수행시키려면 컴퓨터 언어를 사용하여 작업이 수행되도록 프로그램을 작성하여야 한다.

- **기계어(Machine Language)** : CPU에 대한 명령어는 숫자로 표기되다. 이렇게 숫자로 표기된 명령어를 기계어라 하며 메모리에 프로그램은 기계어형태로 저장된다. 마이크로프로세서마다 고유한 기계어를 가지고 있다.
- **어셈블리 언어(Assembly Language)** : 숫자로 표시된 기계어를 보고 명령을 이해하는 것은 거의 불가능하므로 마이크로프로세서 제작사에서는 각 기계어 명령에 1:1 대응하여 명령의 의미를 내포하고 있는 어셈블리 언어를 제공한다. 예를 들어 CPU내의 A레지스터의 내용을 B레지스터에 이동하는 명령이 기계어로 038H(16진수)라고 하여 보자. 038H로부터는 명령의 내용을 이해하기 힘드나, 이를 다음과 같이 표현하면 의미가 명확해 진다.

  기계어 : 038H
  어셈블리 : MOV A, B

  어셈블리 명령어는 기계어에 1:1 대응하므로 이 역시 마이크로프로세서마다 고유하다.
- **어셈블러(Assembler)** : CPU가 이해할 수 있는 명령어는 기계어이므로 어셈블리 언어로 작성된 프로그램은 기계어로 번역되어야 한다. 어셈블러는 어셈블리 언어로 작성된 프로그램을 기계어로 번역하는 소프트웨어를 말한다.
- **고급언어(High Level Language)** : 어셈블러 명령어가 수행명령의 의미를 내포하고 있다고는 하나, 이를 이용해서 프로그램을 하는 것 역시 매우 번거로운 일뿐 아니라, 마이크로프로세서마다 고유한 어셈블리 언어를 가지고 있으므로 마이크로프로세서의 종류가 바뀌면 어셈블리 프로그램은 재 작성하여야 한다. 이에 비해 고급언어는 수학적인 연산이나 논리를 사용자가 이해하기 쉽도록 또한 사용 마이크로프로세서와는 무관하도록 정의하여 프로그램 작성을 좀 더 용이하도록 한다. C-언어, FORTRAN 등이 대표적인 고급언어이다.

▸ **컴파일러(Compiler)** : 고급언어로 작성된 프로그램을 마이크로프로세서가 수행 가능한 기계어로 번역하는 소프트웨어를 컴파일러라 한다. (C-컴파일러는 C-언어를 기계어로 번역하는 소프트웨어이다.) 고급 언어는 마이크로프로세서에 무관하도록 정의되었으므로 컴파일러만 있으면 프로그램의 수정 없이 다른 마이크로프로세서에도 사용할 수 있다. (경우에 따라서 약간의 수정이 필요)

☞ 다음 절부터는 2진수와 16진수의 표현을 번거로운 아래첨자를 사용하지 않고 C-언어나 어셈블리 언어에서 사용하는 법을 채용하여 다음 두 가지 방법을 혼용해서 사용할 것이다.

어셈블리에서 사용하는 방법

`11110000B` - 뒤의 B는 Binary Number(2진수)를 나타냄.

`0ABH` - 앞의 0은 숫자임을 나타내고, 뒤의 H는 Hexadecimal Number(16진수)를 나타냄.

C-언어에서 사용하는 방법

`0b10010011` - 앞의 '0b'는 이진수임을 나타낸다.

`0xAB` - 앞의 '0x'는 이 수가 16진수임을 나타낸다.

연습문제

1. 다음 수를 십진수로 표시하라.
   (1) 11001000_2 (2) 10_{16} (3) $1A0_{16}$ (4) 213_8
2. 다음수를 16진수로 표시하라.
   (1) 11001000_2 (2) 111011000111_2 (3) 102
3. 다음 연산을 수행하라.(16진수 연산 결과는 16진수로, 2진수 연산결과는 2진수로 표시할 것)
   (1) $AB_{16} + 48_{16}$ (2) $1AB_{16} - 3A_{16}$ (3) $11001001_2 + 01000110_2$
   (4) $11001000_2 - 01000110_2$

4. 24비트의 주소버스를 사용할 때 사용가능한 주소의 범위를 16진수로 표시하라. 이 때 장착할 수 있는 메모리의 크기는 몇 바이트인가?
5. -11을 표현하려고 한다.
   (1) 이를 2바이트로 나타낼 때 2진수와 16진수로 표시하라.
   (2) 이를 4바이트로 나타낼 때 2진수와 16진수로 표시하라.
6. 1바이트 부호 없는 두 값의 연산을 하였을 때 결과 값이 1바이트로 표현된다고 하자. 다음 연산의 결과 값은 얼마인가.(모두 16진수로 표시할 것)
   (1) $AB_{16} + 55_{16}$ (2) $AB_{16} + B8_{16}$ (3) $AB_{16} - B8_{16}$
7. 16진수 CB_{16}로 주어진 데이터가 있다고 하자.
(1) 데이터가 부호있는 1바이트 수라고 하면 이 데이터가 나타내는 십진수는 무엇인가.
(2) 데이터가 부호없는 1바이트 수라고 하면 이 데이터가 나타내는 십진수는 무엇인가.
(3) 데이터가 부호있는 2바이트 수라고 하면 이 데이터가 나타내는 십진수는 무엇인가.
(4) 데이터가 부호없는 2바이트 수라고 하면 이 데이터가 나타내는 십진수는 무엇인가.