



2020.07.25
질문 정리

임베디드스쿨1기
Lv1과정
2020. 07. 25
권 영근

◉ 버스(Bus)

버스는 컴퓨터 내부의 구성 요소 사이에서 또는 컴퓨터들 사이에서 데이터를 전송하는 통신 시스템입니다. 초기에는 병렬 연결된 전선이었지만 최근에는 병렬 연결뿐만 아니라 직렬 연결된 전선이기도 합니다.

컴퓨터 시스템은 일반적으로 데이터를 처리하는 CPU와 프로그램 및 데이터를 저장하는 메모리, 외부 세계와 통신하는 페리페럴들인 입출력 모듈들로 구성되었습니다. 이 3가지 모든 장치들 사이에서 버스들은 데이터를 이동시킵니다. 이 버스들 중 CPU와 메모리를 연결하는 버스를 시스템 버스, 데이터 버스라고 부르기도 합니다.

어드레스 버스는 메모리 위치, 즉 실제 주소를 저장하는 버스입니다., 입출력 모듈에게

제어 버스는 CPU가 메모리, 입출력 모듈에게 데이터를 보낼 때나 자신의 상태를 알릴 때 사용되는 신호를 전송하는 버스입니다.

◉ 제어 장치

프로세서의 작동을 지시하는 CPU의 구성 요소입니다. 컴퓨터의 메모리, 산술 장치, 논리 장치, 입출력 모듈들에게 프로세서로 전송이 된 명령에 반응을 하는 방법을 알려줍니다. 그리고 타이밍 및 제어 신호를 제공을 하여 다른 장치들의 작동을 지시합니다. 대부분의 컴퓨터 자원들은 제어 장치에 의해 관리가 됩니다.

◉ 폰노이만 병목 현상이 일어나는 구체적인 원인

프로그램 메모리와 데이터 메모리가 물리적인 구분없이 하나의 버스를 통하여 CPU와 교류하기 때문입니다.

◉ 파이프라인이란?

직렬로 연결된 일련의 데이터 처리 원소들이며 여기에서 한 원소의 출력은 다음 원소의 입력입니다.

일례로 RISC 파이프라인은 동일한 설계에서 여러 명령을 겹치는 실행을 허용합니다.

◉ 하버드 구조에서는 한 클럭에 데이터와 명령어가 함께 들어옵니다. 하지만 폰노이만 구조에서는 그렇게 될 수 없습니다.

◉ 고비용인데 왜 하버드 구조인가?

데이터 메모리와 명령어 메모리가 분리되어 있어서 빠르게 그리고 안전하게 실행하기 때문입니다.

◉ 꼭 하버드 구조인가?

하버드 구조에도 단점이 있습니다. 데이터 메모리, 명령어 메모리 양쪽에 CPU가 연결되었기 때문에 회로가 복잡하며 이는 높은 가격으로 이어집니다. 가격을 낮추기 위해서 폰노이만 구조와 하버드 구조를 함께 사용하여 설계하기도 합니다.

◉ 캐시 메모리는 왜 쓸까?

폰노이만 구조 혹은 하버드 구조 상에서 CPU가 명령어 메모리와 데이터 메모리를 읽는 과정 중에 메모리의 용량, 물리적인 거리때문에 시간이 꽤 걸릴 수 있습니다. 그러한 시간을 줄이기 위해서 캐시 메모리를 사용합니다. 자주 사용하는 메모리 내용은 캐시 메모리에 저장이 됩니다.

◎ CPU 내부는 하버드 구조, 외부는 폰노이만 구조

하버드 구조는 일의 속도 및 효율을 위한 것이고 폰노이만 구조는 비용 절감을 위한 것입니다.

◉ 명령어와 데이터 구조가 나뉘어진다.

명령어와 데이터가 같은 메모리를 공유하는 폰노이만 구조와는 달리 하버드 구조에서는 명령어 메모리와 데이터 메모리가 분리되기 때문에 구조 역시 나뉘어집니다.

◉ 프로세서가 하는 일

Fetch(메모리에서 명령어를 불러들입니다.) → Decode(가져온 명령어를 해독하여 명령어 내의 데이터 정보와 연산 정보 추출 후, CPU 각 장치 내에 제어 신호를 보내서 연산, 처리를 준비합니다.) → Execute(명령어에서 추출한 데이터와 연산 정보를 이용하여 실제 연산을 합니다.). 이러한 사이클 이후 메모리에 결과를 저장 혹은 레지스터에 결과를 저장하게 될 것입니다.

◉ 파이프라인 구조가 무엇인가? 왜 쓰는 것일까?

파이프라인이란 명령어를 읽어 순차적으로 실행하는 프로세서에 적용되는 기술로서 한번에 하나의 명령어만 실행하는 것이 아니라 하나의 명령어가 실행되는 도중에 다른 명령어를 실행하는 식으로 동시에 여러 개의 명령어를 실행하는 기법입니다. 이 기법을 통하여 한 명령어의 특정 단계를 처리하는 동안, 다른 부분에서 다른 명령어의 다른 단계를 처리하여 속도가 향상이 됩니다.

◉ 파이프라인은 폰노이만 구조 및 하버드 구조에 모두 적용이 되는가?

네, 모두 적용이 됩니다.

◎ RISC, CISC는 무엇인가?

RISC(Reduced Instruction Set Computer)는 축소 명령어 집합 컴퓨터,
CISC(Complex Instruction Set Computer)는 복잡 명령어 집합 컴퓨터를 가리킵니다.

◎ Thread는 무엇인가?

어떤 프로그램(특히 프로세스) 내에서 실행되고 있는 흐름의 단위를 뜻합니다. 일반적으로

로그래밍은 1개의 스레드를 가지고 있지만 프로그램 환경에 따라 2개 이상의 스레드를 동시에 실행할 수 있습니다. 이러한 실행 방식을 ‘멀티스레드’ 라고 말합니다.

◉ 파이프라인 기법은 하버드 구조에만 쓰인다?

아닙니다. 폰노이만 구조에도 사용이 됩니다.

◉ 하버드 구조를 많이 사용하는 이유?

데이터 메모리와 프로그램 메모리가 분리되어 있어서 안전하고 빠르게 실행을 하기 때문입니다.

◎ MMU는 무엇인가?

메모리 관리 장치, 즉 MMU는 CPU가 메모리에 접근하는 것을 관리하는 하드웨어입니다. 가상 메모리 주소를 실제 메모리 주소로 변환하며 메모리 보호, 캐시 관리, 버스 중재 등의 역할을 합니다.



2020.07.25
수업 복습
operators

임베디드스쿨1기
Lv1과정
2020. 07. 25
권 영근

연산자

컴파일러에게 특정 수학 혹은 논리 기능을 수행하게 하는 기호입니다.

◉ 산술 연산자

Arithmetic Operators/Operation	Example
+ (Addition)	A+B
- (Subtraction)	A-B
* (multiplication)	A*B
/ (Division)	A/B
% (Modulus)	A%B

※ 위의 표에는 없어서 보충하겠습니다. ‘++’ 연산자와 ‘--’ 연산자들이 있습니다.

‘++a’ 는 해당 라인에서 즉시 a의 값을 1 증가시키며 ‘a++’ 는 다음 라인에서 a를 1 증가시킵니다. ‘--a’, ‘a--’ 들도 이와 비슷하게 작동을 합니다.

```
#include <stdio.h>

int main(void)
{
    int a = 21;
    int b = 10;
    int c;

    c = a + b;
    printf("Line 1 -Value of c is %d\n", c);

    c = a - b;
    printf("Line 2 -Value of c is %d\n", c);

    c = a * b;
    printf("Line 3 -Value of c is %d\n", c);

    c = a / b;
    printf("Line 4 -Value of c is %d\n", c);

    c = a % b;
    printf("Line 5 -Value of c is %d\n", c);

    return 0;
}
```

```
bolthakziy@bolthakziy-340XAA-350XAA-550XAA:~/C
bolthakziy@bolthakziy-340XAA-350XAA-550XAA:~/C
Line 1 -Value of c is 31
Line 2 -Value of c is 11
Line 3 -Value of c is 210
Line 4 -Value of c is 2
Line 5 -Value of c is 1
```

```
#include <stdio.h>

int main(void)
{
    int a = 21;
    int b = 10;
    int c;

    c = a++;
    printf("Line 1 -Value of c is %d\n", c);

    c = a--;
    printf("Line 2 -Value of c is %d\n", c);

    c = ++b;
    printf("Line 3 -Value of c is %d\n", c);

    c = --b;
    printf("Line 4 -Value of c is %d\n", c);

    return 0;
}
```

```
Line 1 -Value of c is 21
Line 2 -Value of c is 22
Line 3 -Value of c is 11
Line 4 -Value of c is 10
```

◉ 관계 연산자

Operators	Example/Description
>	$x > y$ (x is greater than y)
<	$x < y$ (x is less than y)
>=	$x \geq y$ (x is greater than or equal to y)
<=	$x \leq y$ (x is less than or equal to y)
==	$x == y$ (x is equal to y)
!=	$x != y$ (x is not equal to y)

```

#include <stdio.h>

int main(void)
{
    int a = 21;
    int b = 10;

    if(a == b)
    {
        printf("Line 1 - a is equal to b\n");
    }
    else
    {
        printf("Line 1 - a is not equal to b\n");
    }

    if(a < b)
    {
        printf("Line 2 - a is less than b\n");
    }
    else
    {
        printf("Line 2 - a is not less than b\n");
    }

    if(a > b)
    {
        printf("Line 3 - a is greater than b\n");
    }
    else
    {
        printf("Line 3 - a is not greater than b\n");
    }

    a = 5;
    b = 20;

    if(a <= b)
    {
        printf("Line 4 - a is either less than or equal to b\n");
    }

    if(b >= a)
    {
        printf("Line 5 - b is either greater than or equal to a\n");
    }

    return 0;
}

```

```

Line 1 - a is not equal to b
Line 2 - a is not less than b
Line 3 - a is greater than b
Line 4 - a is either less than or equal to b
Line 5 - b is either greater than or equal to a

```

◉ 논리 연산자

Operators	Example/Description
&& (logical AND)	$(x > 5) \&\& (y < 5)$ It returns true when both conditions are true
(logical OR)	$(x \geq 10) (y \geq 10)$ It returns true when at-least one of the condition is true
! (logical NOT)	$!((x > 5) \&\& (y < 5))$ It reverses the state of the operand “ $((x > 5) \&\& (y < 5))$ ” If “ $((x > 5) \&\& (y < 5))$ ” is true, logical NOT operator makes it false

```

#include <stdio.h>

int main(void)
{
    int a = 21;
    int b = 10;
    int c;

    if(a && b)
    {
        printf("Line 1 - Condition is true.\n");
    }

    if(a || b)
    {
        printf("Line 2 - Condition is true.\n");
    }

    a = 0;
    b = 10;

    if(a && b)
    {
        printf("Line 3 - Condition is true.\n");
    }
    else
    {
        printf("Line 3 - Condition is false.\n");
    }

    if(!(a && b))
    {
        printf("Line 4 - Condition is true.\n");
    }

    return 0;
}

```

```


Line 1 - Condition is true.
Line 2 - Condition is true.
Line 3 - Condition is false.
Line 4 - Condition is true.

```

◉ 비트 연산자

비트 연산을 컴파일러에게 수행하게 하는 기호입니다.

◆ NOT

Negation	
NOT	
Definition	\bar{x}
Truth table	(10)
Logic gate	
Normal forms	
Disjunctive	\bar{x}
Conjunctive	\bar{x}
Zhegalkin polynomial	$1 \oplus x$

P	$\neg P$
True	False
False	True

1. 0을 1로, 그리고 1을 0으로 만드는 연산자입니다. 그렇기 때문에 어느 2진수 숫자에 NOT 연산자를 취하면 그 2진수 숫자의 '1의 보수' 형태의 숫자가 나타납니다.

예를 들어서 0111(2)에 NOT을 취하게 되면 1000(2)가 됩니다.

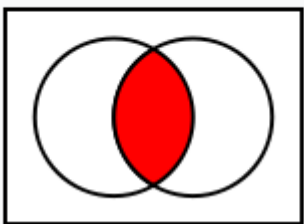
2. 1의 내용을 활용하여 알아보면, 어떤 수의 비트 보수는 그 수의 2의 보수에서 1을 뺀 수와 같습니다. 즉 ' $\text{NOT}(x) = -x - 1$ ' 입니다.

◆ AND

입력부의 모든 값이 True일 때, 출력값이 True가 되는 비트 연산입니다.


Logical conjunction

AND



Definition xy

Truth table (0001)

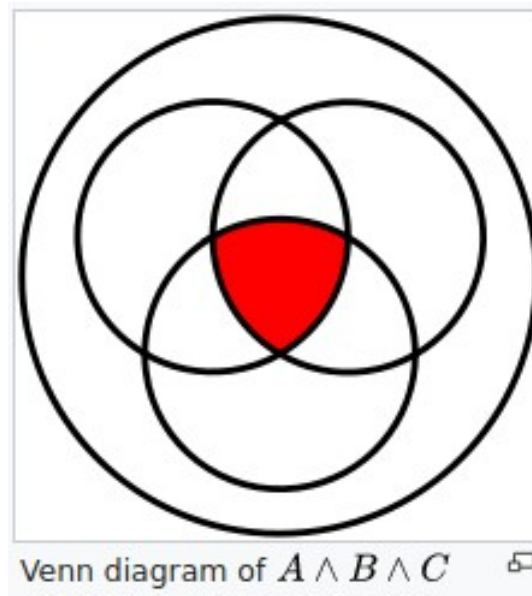
Logic gate 

Normal forms

Disjunctive xy

Conjunctive xy

Zhegalkin polynomial xy



A	B	$A \wedge B$
T	T	T
T	F	F
F	T	F
F	F	F

1. 최대한 적은 메모리를 사용하기 위해서 비트 마스크 기능에 사용이 됩니다. 예를 들어 어느 레지스터의 'flag' 비트를 0으로 만드는데(clear) 사용될 수 있습니다.

(ex)

```
#define RegisterBits (*(unsigned int*)0x3C)
// RegisterBits => |Bit7|Bit6|flag|Bit4|Bit3|Bit2|Bit1|Bit0|
```

```
RegisterBit = 0xFF;
Delay(1000);
RegisterBit &= 0xDF; // 0xDF = 0b11011111
// RegisterBits => |1|1|0|1|1|1|1|1|
```

2. 자연수의 홀수, 짝수 판별에도 사용이 됩니다.

(ex)

```
int a = 6, b = 3; // 6 = 0b00000110, 3 = 0b00000011
int mask = 1; // 1 = 0b00000001
```

```
a &= mask;
b &= mask;
```

// a = 0, b = 1

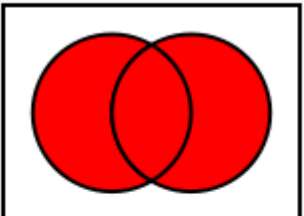
// If the number is even, the result will be '0' but if not, it'll be '1'.

◆ OR

모든 입력부의 값들이 False일 때, 출력값이 False이 되며 그 이외의 경우, 출력값은 True입니다.


Logical disjunction

OR



Definition $x + y$

Truth table (0111)

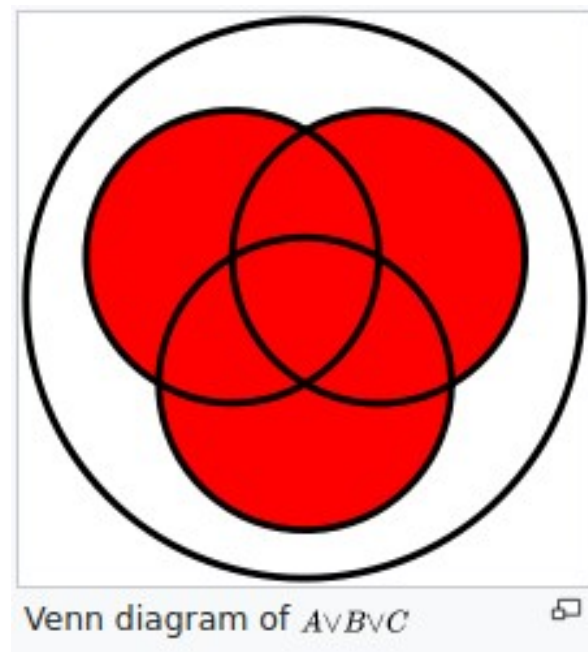
Logic gate 

Normal forms

Disjunctive $x + y$

Conjunctive $x + y$

Zhegalkin polynomial $x \oplus y \oplus xy$



A	B	$A \vee B$
T	T	T
T	F	T
F	T	T
F	F	F

역시 비트 마스크 기능에서 사용될 수 있는데, 어느 레지스터의 특정 비트를 1로 설정할 때 (set), 사용이 됩니다.

(ex)

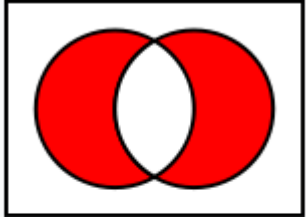
```
#define RegisterBits (*(unsigned int*)0x3C)
// RegisterBits => |Bit7|Bit6|flag|Bit4|Bit3|Bit2|Bit1|Bit0|
```

```
RegisterBit = 0x00;
Delay(1000);
RegisterBit &= 0x80; // 0x80 = 0b10000000
// RegisterBits => |1|0|0|0|0|0|0|0|
```


◆ XOR

입력값들이 서로 다를 때에만 출력값이 True가 되게 하는 비트 연산입니다.

Exclusive or
XOR

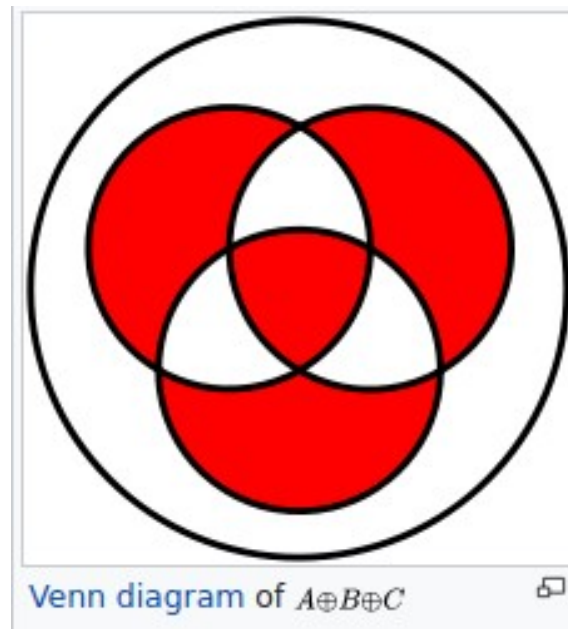


Truth table (0110)
Logic gate



Normal forms

Disjunctive $\bar{x} \cdot y + x \cdot \bar{y}$
Conjunctive $(\bar{x} + \bar{y}) \cdot (x + y)$
Zhegalkin polynomial $x \oplus y$



Input		Output
A	B	
0	0	0
0	1	1
1	0	1
1	1	0

1. 어느 레지스터의 특정 비트를 Toggle($0 \rightarrow 1$, $1 \rightarrow 0$)하는데 사용이 됩니다.

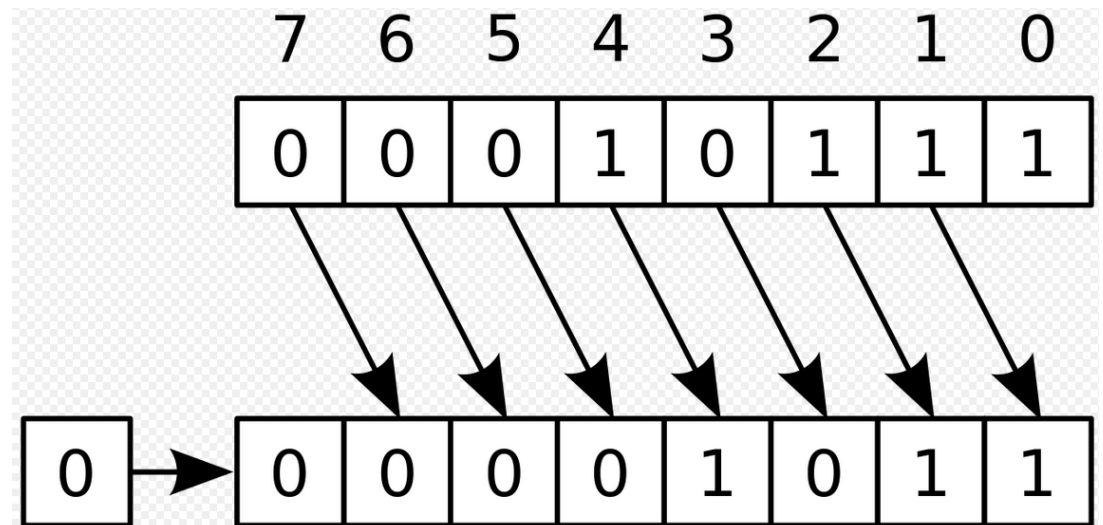
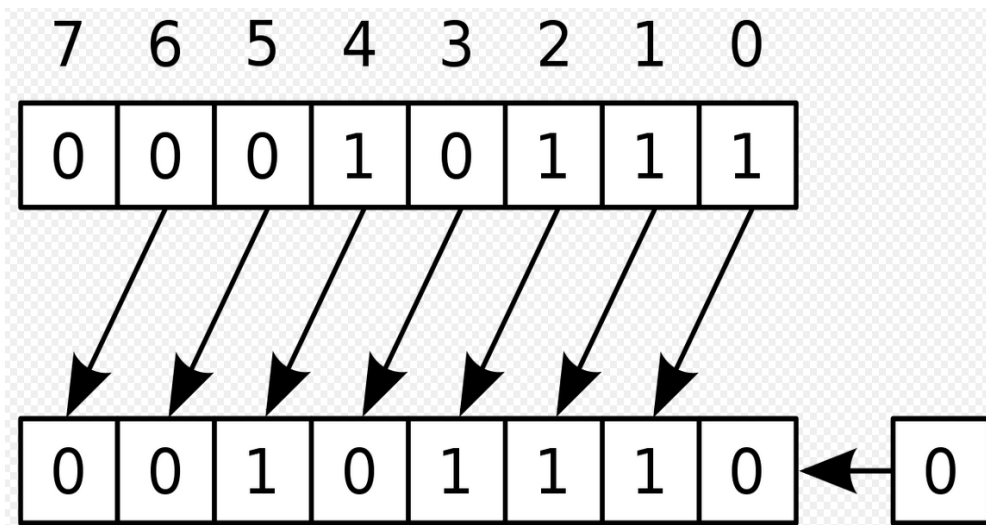
(ex)
`#define RegisterBits (*(unsigned int*)0x3C)`
`// RegisterBits => |Bit7|Bit6|flag|Bit4|Bit3|Bit2|Bit1|Bit0|`

`RegisterBit = 0x0A; // 0x0A = 0b00001010`
`Delay(1000);`

```
RegisterBit ^= 0xFF; // 0xFF = 0b11111111
// RegisterBit => |1|1|1|1|0|1|0|1|
```

2. 어셈블리 프로그래머와 최적화 컴파일러에서 때때로 어느 레지스터의 특정 비트를 0으로 클리어하기 위해서 XOR 연산을 사용합니다. 왜냐하면 직접 0값을 레지스터에 저장하는 것에 비해서 적은 클럭 사이클과 메모리가 필요하기 때문입니다.

◆ Left Shift & Right Shift



비트들을 왼쪽 혹은 오른쪽으로 이동시키는 연산입니다.

1. Left Shift 연산에서 n칸을 이동할 시, 기존의 수에서 $2^{**}n$ 배 곱해진 수가 나타납니다 (단, 오버플로우가 발생하지 않는다는 조건 하에서입니다.).

(ex)

```
int a = 23; // 23 = 0b00010111  
int b, c;
```

```
b = a << 1; // b = 46 = 0b00101110  
c = a << 2; // b = 92 = 0b01011100
```

2. Right Shift 연산에서 n칸을 이동할 시, 기존의 수에 $2^{**}n$ 을 나눈 수가 나타납니다.

(ex)

```
int a = 23; // 23 = 0b00010111  
int b;
```

```
b = a >> 1; // b = 11 = 0b00001011
```

3. Arithmetic Shift라는 연산도 있습니다. 일반적인 Left Shift 및 Right Shift의 기능과 거의 비슷합니다. 다만 Right Arithmetic Shift를 할 때, 입력값의 부호 상태는 유지된 채, 비트들이 오른쪽으로 이동합니다. 이러한 점이 다를 뿐입니다.

(ex)

```
mov eax, -23 ; -23 = 0b10010111
```

```
sar eax, 1 ; eax 안에는 0b11001011, 즉 -11이 들어있습니다.
```

위의 사례를 통해서 Arithmetic Shift 연산 때문에 음수들 사이에서 곱셈 및 나눗셈이 가능하다는 것을 유추할 수 있습니다. 만일 -23을 1만큼 Right Shift를 하였다면, -11대신 0b01001011, 즉 75라는 값을 얻게 될 것입니다. 음수를 2로 나눴는데 음수 대신 양수가 나오는 것도 모잘라 나뉘어진 수가 기존의 수보다 그 절대량이 커지는 일이 벌어진 것입니다.

◆ 지정 연산자

Assignment Operators

Operator	Example	Equivalent Expression
=	$m = 10$	$m = 10$
+=	$m += 10$	$m = m + 10$
-=	$m -= 10$	$m = m - 10$
*=	$m *= 10$	$m = m * 10$
/=	$m /=$	$m = m / 10$
%=	$m \% = 10$	$m = m \% 10$
<<=	$a <<= b$	$a = a << b$
>>=	$a >>= b$	$a = a >> b$
>>>=	$a >>>= b$	$a = a >>> b$
&=	$a \&= b$	$a = a \& b$
^=	$a \wedge= b$	$a = a \wedge b$
=	$a = b$	$a = a b$

소스 코드 ' $x = x + y$ '의 실행 속도보다 ' $x += y$ '의 실행 속도가 더 빠릅니다.

```

#include <stdio.h>

int main(void)
{
    int a = 21;
    int c;

    c = a;
    printf("Line 1 - = Operator Example, Value of c = %d\n", c);

    c += a;
    printf("Line 2 - += Operator Example, Value of c = %d\n", c);

    c -= a;
    printf("Line 3 - -= Operator Example, Value of c = %d\n", c);

    c *= a;
    printf("Line 4 - *= Operator Example, Value of c = %d\n", c);

    c /= a;
    printf("Line 5 - /= Operator Example, Value of c = %d\n", c);

    c = 200;
    c %= a;
    printf("Line 6 - %= Operator Example, Value of c = %d\n", c);

    c <<= 2;
    printf("Line 7 - <<= Operator Example, Value of c = %d\n", c);

    c >>= 2;
    printf("Line 8 - >>= Operator Example, Value of c = %d\n", c);

    c &= 2;
    printf("Line 9 - &= Operator Example, Value of c = %d\n", c);

    c ^= 2;
    printf("Line 10 - ^= Operator Example, Value of c = %d\n", c);

    c |= 2;
    printf("Line 11 - |= Operator Example, Value of c = %d\n", c);

    return 0;
}

```

```

Line 1 - = Operator Example, Value of c = 21
Line 2 - += Operator Example, Value of c = 42
Line 3 - -= Operator Example, Value of c = 21
Line 4 - *= Operator Example, Value of c = 441
Line 5 - /= Operator Example, Value of c = 21
Line 6 - %= Operator Example, Value of c = 11
Line 7 - <<= Operator Example, Value of c = 44
Line 8 - >>= Operator Example, Value of c = 11
Line 9 - &= Operator Example, Value of c = 2
Line 10 - ^= Operator Example, Value of c = 0
Line 11 - |= Operator Example, Value of c = 2

```



HW1

임베디드스쿨1기

Lv1과정

2020. 07. 25

권 영근

HW1

```
#include <stdio.h>

int main(void)
{
    unsigned char A, B, C, ans1, ans2, ans3, ans4;

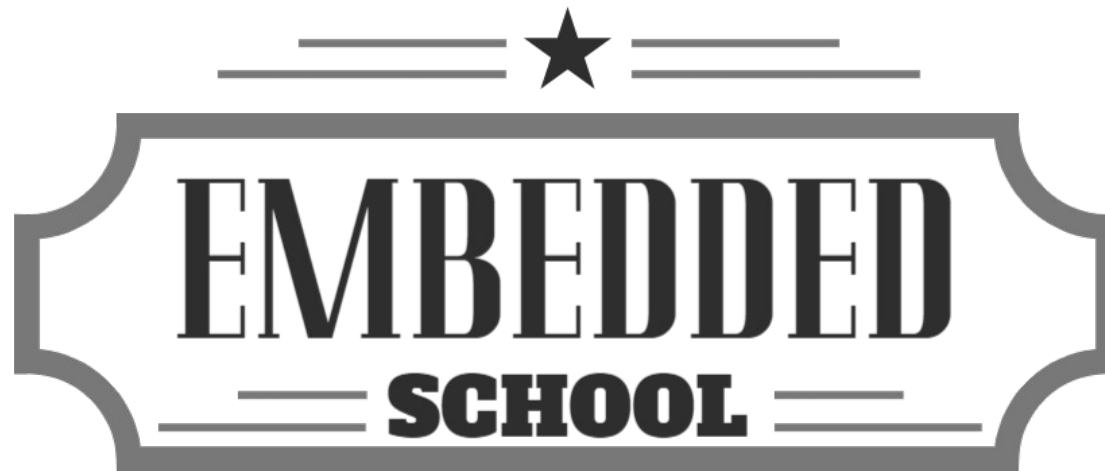
    scanf("%d", &A);
    scanf("%d", &B);
    scanf("%d", &C);

    ans1 = (A + B) % C;
    ans2 = ((A % C) + (B % C)) % C;
    ans3 = (A * B) % C;
    ans4 = ((A % C) * (B % C)) % C;

    printf("%d\n%d\n%d\n%d\n", ans1, ans2, ans3, ans4);

    return 0;
}
```

```
bolthakziy@bolthakziy-340XAA-350XAA-550XAA:~/C$ ./main
5
8
4
1
1
0
0
```



2020.07.25
수업 복습
Decision Making

임베디드스쿨1기
Lv1과정
2020. 07. 25
권 영근

의사 결정

◉ 특정 조건에 따라 문장의 실행 순서를 결정하거나 특정 조건이 충족될 때까지 문자열 반복하는 행위입니다.

◆ if statement

‘만일 조건문이 사실이라면 A를 행하고 그렇지 않으면 A를 행하지 않는다.’ 지침을 수행합니다.

1. 단순 If문

```
#include <stdio.h>

int main(void)
{
    int a = 10;

    if(a < 20)
    {
        printf("a is less than 20\n");
    }

    printf("Vlaue of a is : %d\n", a);

    return 0;
}
```

```
bolthakziy@bolthakziy-340XAA-350XAA-550XAA:~/C$ ./main
a is less than 20
Vlaue of a is : 10
```

2. 중첩된 If문

If문 안에 If문이 더 들어있는 구조입니다. 첫 번째 If문의 조건이 참이어야 내부의 If문 조건을 마주할 수 있습니다.

```
#include <stdio.h>

int main(void)
{
    int a = 100, b = 200;

    if(a == 100)
    {
        if(b == 200)
        {
            printf("Value of a is 100 and value of b is 200.\n");
        }
    }

    printf("Exact value of a is : %d\n", a);
    printf("Exact value of b is : %d\n", b);

    return 0;
}
```

```
Value of a is 100 and value of b is 200.
Exact value of a is : 100
Exact value of b is : 200
```

◆ If ~else statement

if문의 조건이 거짓을 경우 그 다음의 조건문을 만나 그 조건문이 참인지 거짓인지에 따라 실행을 달리 합니다.

1. if & else 구문

```
#include <stdio.h>

int main(void)
{
    int a = 100;

    if(a < 20)
    {
        printf("a is less than 20.\n");
    }
    else
    {
        printf("a is not less than 20.\n");
    }

    printf("Value of a is : %d.\n", a);

    return 0;
}
```

```
a is not less than 20.
Value of a is : 100.
```


2. if ~ else if ~ else 구문

```
#include <stdio.h>

int main(void)
{
    int a = 100;

    if(a == 10)
    {
        printf("Value of a is 10.\n");
    }
    else if(a == 20)
    {
        printf("Value of a is 20.\n");
    }
    else if(a == 30)
    {
        printf("Value of a is 30.\n");
    }
    else
    {
        printf("None of the value is matching.\n");
    }

    printf("Exact value of a is %d.\n", a);

    return 0;
}
```

None of the value is matching.
Exact value of a is 100.

※ if문에서 'else' 키워드에는 조건문이 옆에 붙지 않는다는 것을 알 수 있습니다.

◆ switch ~ case statement

1. 단일 switch 구문

어떤 특정 입력값을 받아들입니다. 'case' 라는 키워드를 사용하여 여러 가지 경우의 조건들을 나눈 후, 입력값과 일치하는 조건이 있을 시, 그 조건에 해당하는 문장을 실행합니다.

```
#include <stdio.h>

int main(void)
{
    char grade = 'B';

    switch(grade)
    {
        case 'A' :
            printf("Excellent!\n");
            break;
        case 'B' :
        case 'C' :
            printf("Well done!\n");
            break;
        case 'D' :
            printf("You passed!\n");
            break;
        case 'F' :
            printf("You failed...\n");
            break;
        default :
            printf("Invalid grade.\n");
            break;
    }

    printf("Your grade is %c.\n", grade);

    return 0;
}
```

```
Well done!
Your grade is B.
```

Switch 구문의 'default' 키워드는 if 구문의 'else' 키워드와 비슷합니다. 가장 마지막으로 남는 조건을 가지는 동시에 옆에 조건(문)과 관련된 것이 없기 때문입니다.

2. 중첩된 switch 구문

```
#include <stdio.h>

int main(void)
{
    int a = 100, b = 200;

    switch(a)
    {
        case 100 :
            printf("This is a part of outer switch.\n");

            switch(b)
            {
                case 200 :
                    printf("This is a part of inner switch.\n");
                    break;
                default :
                    break;
            }
            break;
        default :
            break;
    }

    printf("Exact value of a is %d\n", a);
    printf("Exact value of b is %d\n", b);

    return 0;
}
```

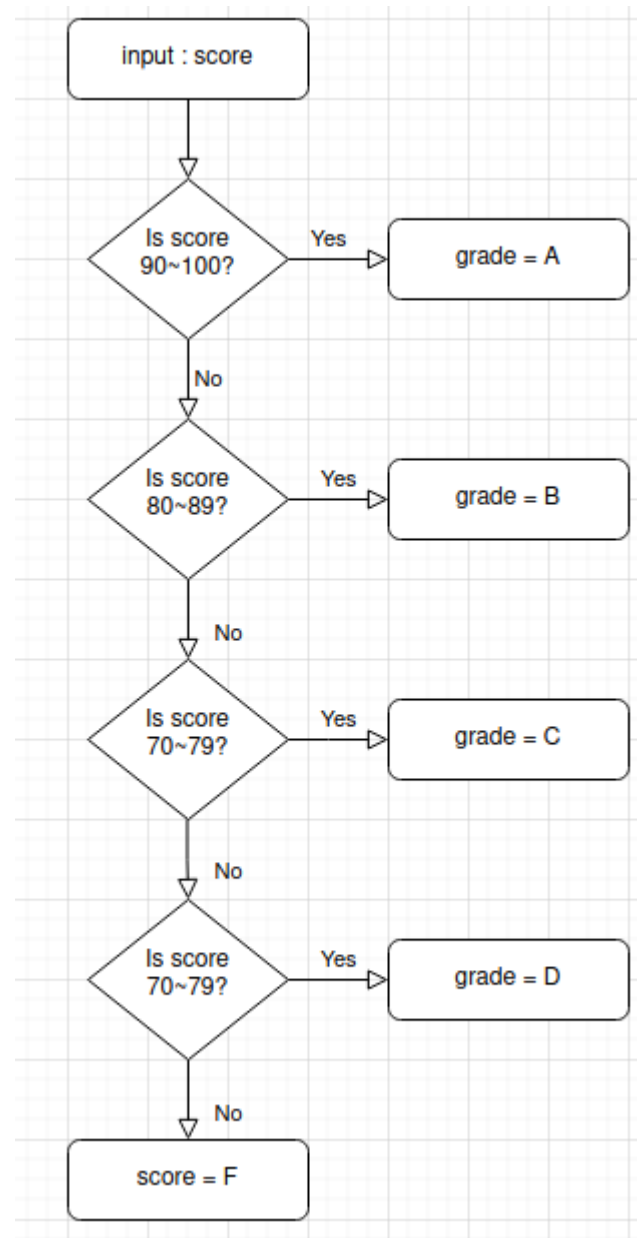
```
bolthakziy@bolthakziy-340XAA-350XAA-550XAA:~/C$ gcc -o
bolthakziy@bolthakziy-340XAA-350XAA-550XAA:~/C$ ./main
This is a part of outer switch.
This is a part of inner switch.
Exact value of a is 100
Exact value of b is 200
```



2020.07.25
HW1~HW2

임베디드스쿨1기
Lv1과정
2020. 07. 25
권 영근

HW1



```
#include <stdio.h>

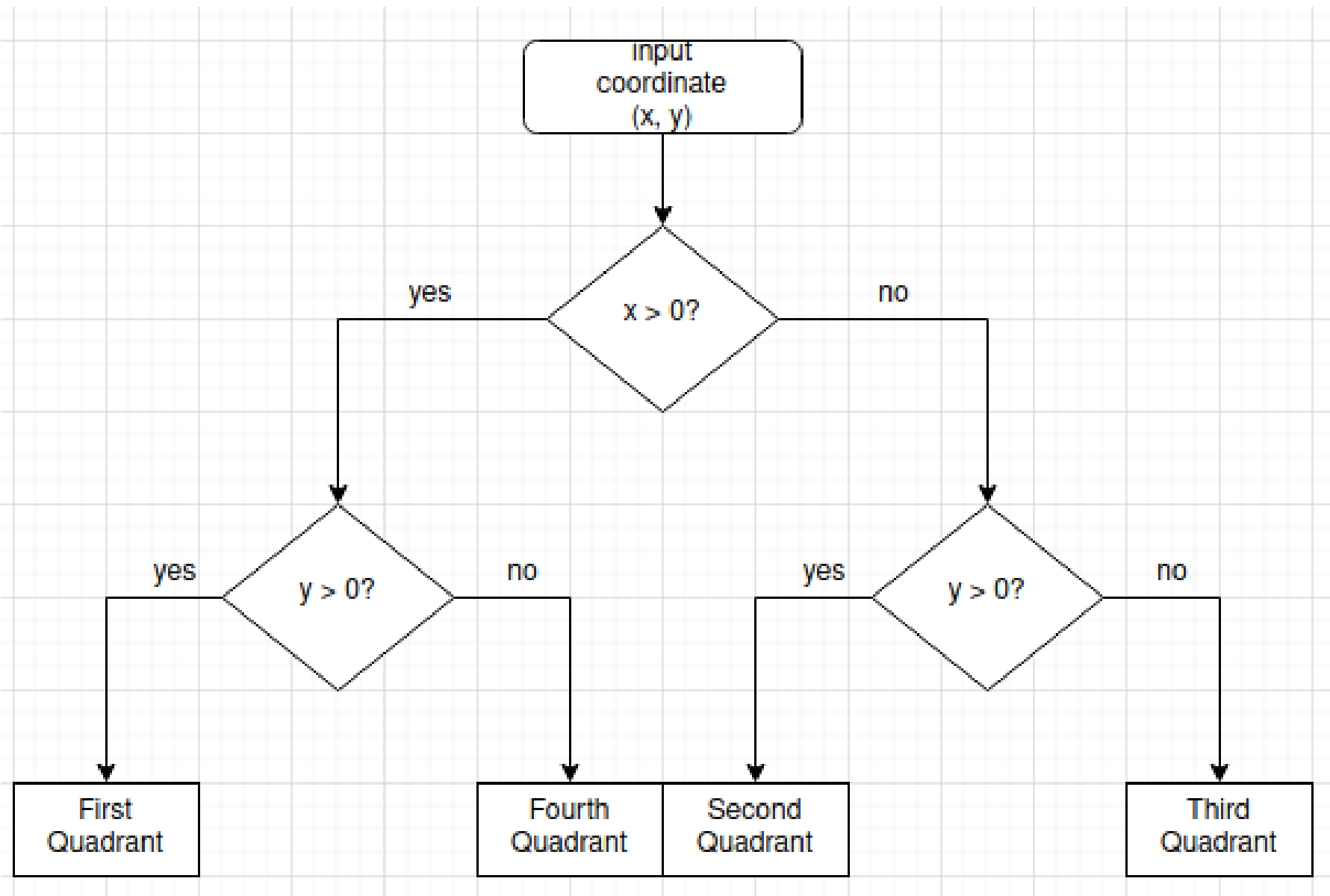
int main(void)
{
    unsigned char score, grade;
    scanf("%hhd", &score);

    if((score <= 100) && (score >= 90))
    {
        grade = 'A';
    }
    else if((score <= 89) && (score >= 80))
    {
        grade = 'B';
    }
    else if((score <= 79) && (score >= 70))
    {
        grade = 'C';
    }
    else if((score <= 69) && (score >= 60))
    {
        grade = 'D';
    }
    else
    {
        grade = 'F';
    }

    printf("%c\n", grade);

    return 0;
}
```

```
bolthakziy@bolthakziy-340XAA-350XAA-550XAA:~/C$ ./main
100
A
bolthakziy@bolthakziy-340XAA-350XAA-550XAA:~/C$ ./main
65
D
bolthakziy@bolthakziy-340XAA-350XAA-550XAA:~/C$ ./main
88
B
bolthakziy@bolthakziy-340XAA-350XAA-550XAA:~/C$ ./main
71
C
bolthakziy@bolthakziy-340XAA-350XAA-550XAA:~/C$ ./main
43
F
```



HW1

```
#include <stdio.h>

int main(void)
{
    unsigned char quadrant;
    int pointX, pointY;
    scanf("%d", &pointX);
    scanf("%d", &pointY);

    if((pointX > 0) && (pointY > 0))
    {
        quadrant = 1;
    }
    else if((pointX > 0) && (pointY < 0))
    {
        quadrant = 4;
    }
    else if((pointX < 0) && (pointY > 0))
    {
        quadrant = 2;
    }
    else if((pointX < 0) && (pointY < 0))
    {
        quadrant = 3;
    }
    else
    {
    }

    printf("%hhd\n", quadrant);

    return 0;
}
```

```
bolthakziy@bolthakziy-340XAA-350XAA-550XAA:~/C$ ./main
12
5
1
bolthakziy@bolthakziy-340XAA-350XAA-550XAA:~/C$ ./main
9
-13
4
bolthakziy@bolthakziy-340XAA-350XAA-550XAA:~/C$ ./main
-4
3
2
bolthakziy@bolthakziy-340XAA-350XAA-550XAA:~/C$ ./main
-9
-7
3
```




2020.07.25
수업 복습
Loop Control

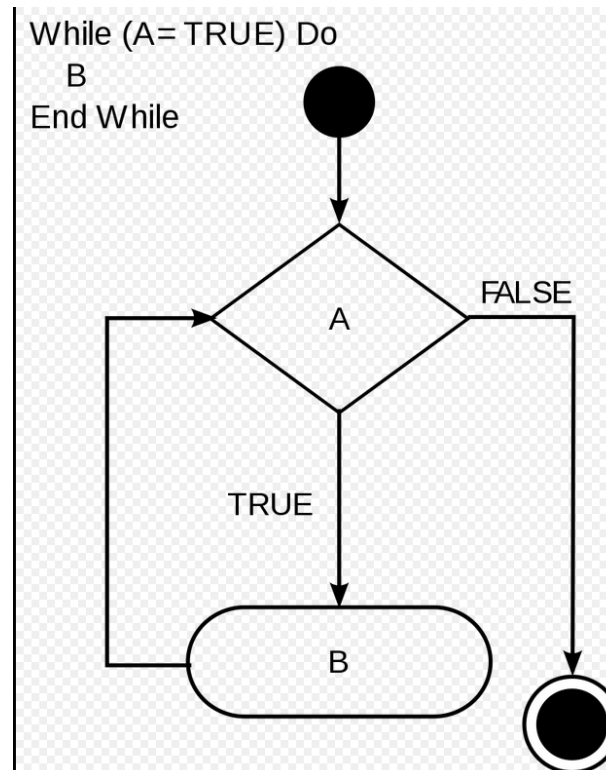
임베디드스쿨1기
Lv1과정
2020. 07. 25
권 영근

Loop Control

◉ 특정 조건에 도달할 때까지 반복을 하는 것입니다.

◆ while Loop

주어진 조건문에 따라 소스 코드를 반복적으로 실행할 수 있는 제어문입니다. 반복되는 if 문이라고 생각할 수도 있습니다. 조건을 항상 True로 설정을 하면 무한 루프를 만들 수 있습니다. 이러한 무한 루프는 전원이 인가되면 항상 특정 기능을 반복해야 하는 펌웨어 시스템에 주로 사용이 됩니다.



```
#include <stdio.h>

int main(void)
{
    int a = 10;

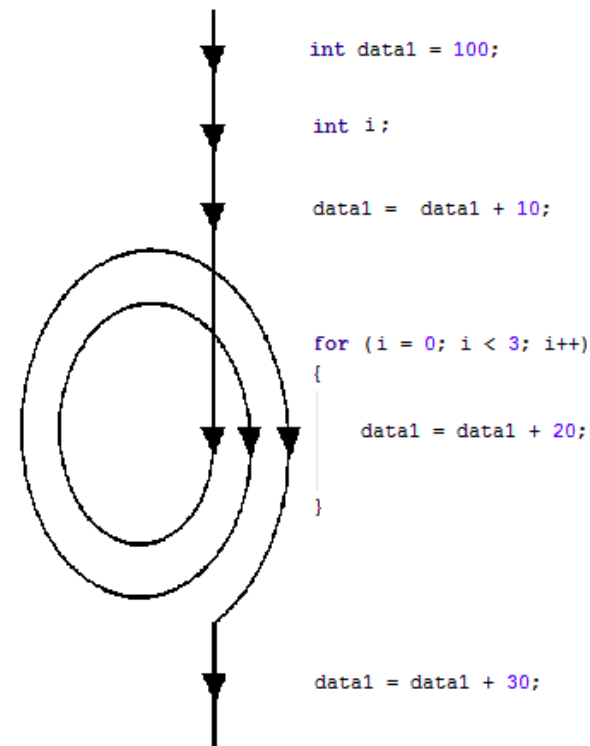
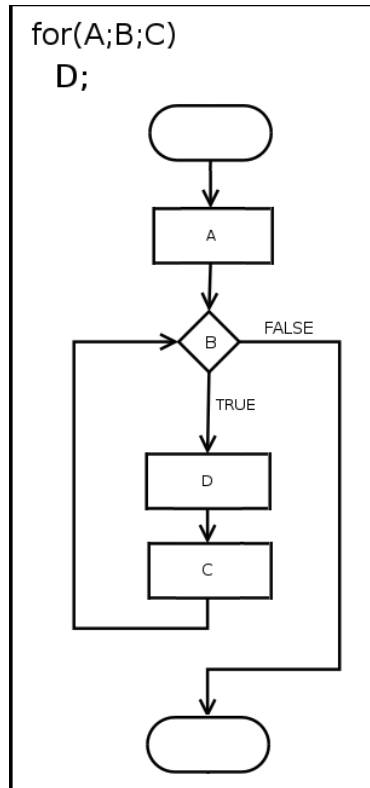
    while(a < 20)
    {
        printf("value of a : %d\n", a);
        a++;
    }

    return 0;
}
```

```
value of a : 10
value of a : 11
value of a : 12
value of a : 13
value of a : 14
value of a : 15
value of a : 16
value of a : 17
value of a : 18
value of a : 19
```

◆ for Loop

반복을 지정하기 위한 제어 흐름입니다. 반복을 지정하는 헤더와 매 반복마다 한 번씩 실행되는 소스 코드들이 있는 본체로 구성되어 있습니다. 헤더에 '루프 카운터'라는 변수가 있는데, 이것은 반복을 제어하는 변수로서 지정된 순서대로 정수 값의 범위를 취합니다. 일례로 0에서 시작을 하여 9까지 1씩 증가하면서 10번 동안 본체의 소스 코드를 수행합니다.



```
#include <stdio.h>

int main(void)
{
    int a;

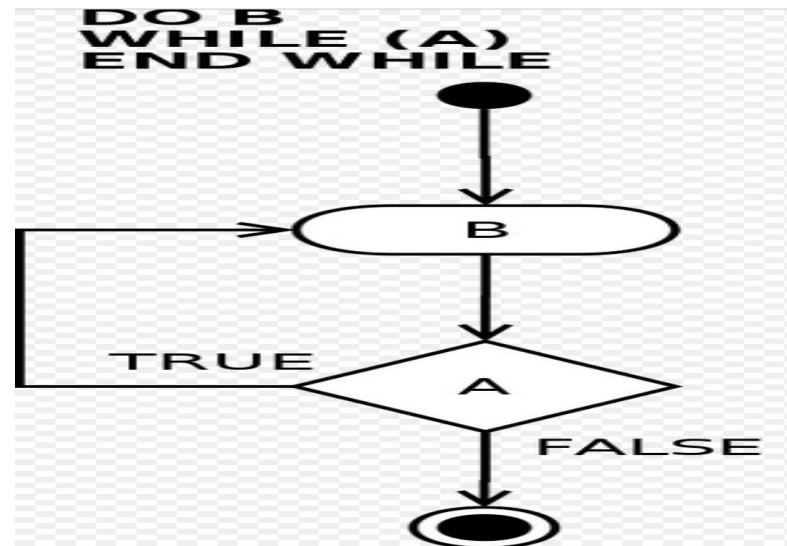
    for(a = 10; a < 20; a = a + 1)
    {
        printf("Value of a : %d\n", a);
    }

    return 0;
}
```

```
Value of a : 10
Value of a : 11
Value of a : 12
Value of a : 13
Value of a : 14
Value of a : 15
Value of a : 16
Value of a : 17
Value of a : 18
Value of a : 19
```

◆ do while Loop

실행할 소스 코드 블록을 한 번은 실행한 후, 블록 끝에 있는 조건에 따라서 블록 안의 소스 코드를 반복하거나 종지를 하는 제어 흐름 명령문입니다. 이 때문에 종종 테스트 후의 Loop라고 불리기도 합니다. 소스 코드를 실행하기 전에 조건을 점검하는 while 구문과는 다르게 do while 구문은 종료 조건의 구문이며, 이는 곧 소스 코드를 항상 먼저 실행한 후에 조건문을 확인하는 구문을 뜻합니다. while문이 조건의 True를 선행 조건으로 소스 코드를 반복적으로 실행한다면 do while 문은 조건의 False를 해제 조건으로 소스 코드를 반복적으로 실행합니다. 또한 while 문처럼 무한 루프로 만들 수 있는데, 'break' 키워드를 활용하여 어떤 특정 조건 하에서 do while 무한 루프를 탈출할 수 있습니다.



```
#include <stdio.h>
```

```
int main(void)
```

```
{
```

```
    int a = 10 ;
```

```
    do
```

```
    {
```

```
        printf("Value of a ; %d\n", a);
```

```
        a = a + 1;
```

```
    }while(a < 20);
```

```
    return 0;
```

```
}
```

```
Value of a ; 10
```

```
Value of a ; 11
```

```
Value of a ; 12
```

```
Value of a ; 13
```

```
Value of a ; 14
```

```
Value of a ; 15
```

```
Value of a ; 16
```

```
Value of a ; 17
```

```
Value of a ; 18
```

```
Value of a ; 19
```

◆ break

진행되고 있는 루프가 어느 특정 조건이 되었을 때, 해당 루프를 중지하고 루프 구문 바로 다음의 명령문으로 제어를 전송하게 도와주는 keyword입니다.

```
#include <stdio.h>

int main(void)
{
    int a = 10 ;

    while(a < 20)
    {
        printf("Value of a : %d\n", a);
        a++;

        if(a > 15)
        {
            break;
        }
    }

    return 0;
}
```

```
Value of a : 10
Value of a : 11
Value of a : 12
Value of a : 13
Value of a : 14
Value of a : 15
```


◆ continue

때때로 프로그래머들은 루프 구문을 실행하는 과정에서 어느 특정한 때만은 루프 구문 내의 건너 뛰고 그 루프 구문의 반복 작업을 계속적으로 하고자 하는 바람을 가지고 있습니다. 이러한 바람을 실현시키는 keyword가 'continue' 입니다.

```
#include <stdio.h>

int main(void)
{
    int a = 10 ;

    do
    {
        if(a == 15)
        {
            a = a + 1;
            continue;
        }

        printf("Value of a : %d\n", a);
        a++;
    }while(a < 20);

    return 0;
}
```

```
Value of a : 10
Value of a : 11
Value of a : 12
Value of a : 13
Value of a : 14
Value of a : 16
Value of a : 17
Value of a : 18
Value of a : 19
```

◆ goto statement

프로그램의 어느 부분에서 다른 부분으로 건너뛸 때, 사용되는 명령어입니다. 과도한 goto 구문을 사용하면 읽고 유지하기 힘든 스파게티 코드가 된다면서 많은 컴퓨터 과학자들이 goto문의 과도한 사용을 경계하였습니다. 대표적인 컴퓨터 과학자가 에츨허르 데이크스트라입니다.

물론 실력이 매우 뛰어난 프로그래머가 goto 구문을 사용한다면 관찬을 것입니다.

```
#include <stdio.h>

int main(void)
{
    int a = 10 ;

    LOOP : do
    {
        if(a == 15)
        {
            a = a + 1;
            goto LOOP;
        }

        printf("Value of a : %d\n", a);
        a++;
    }while(a < 20);

    return 0;
}
```

```
Value of a : 10
Value of a : 11
Value of a : 12
Value of a : 13
Value of a : 14
Value of a : 16
Value of a : 17
Value of a : 18
Value of a : 19
```



HW1~HW3

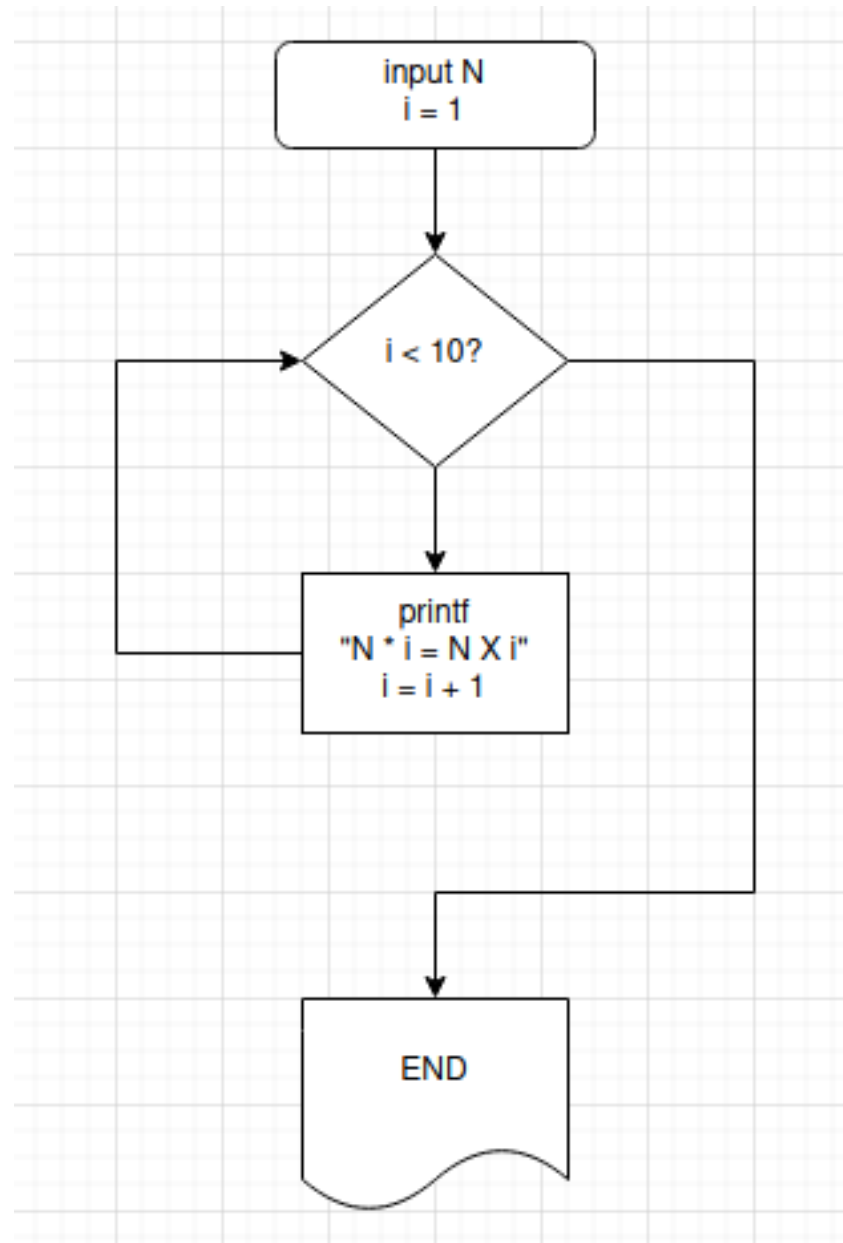
임베디드스쿨1기

Lv1과정

2020. 07. 25

권 영근

HW1



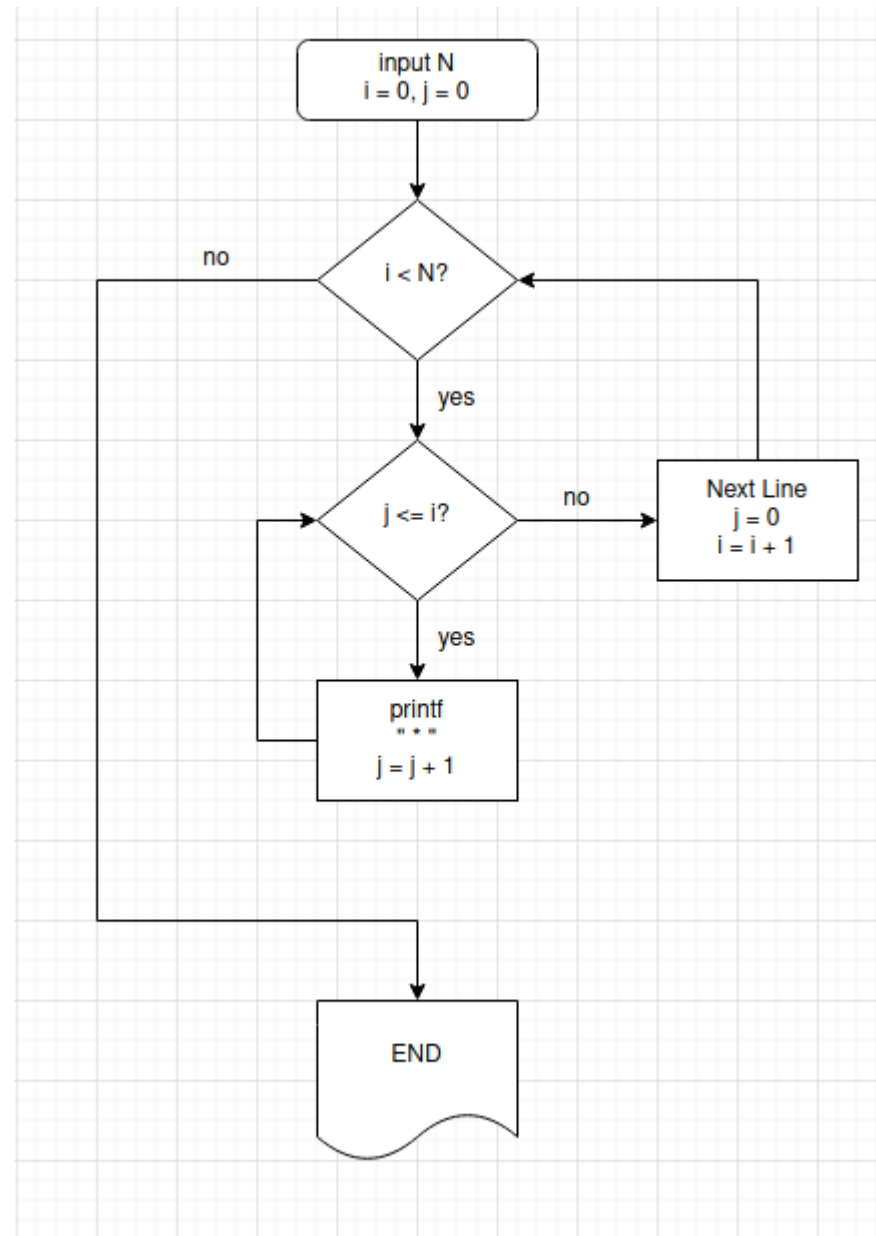
```
#include <stdio.h>

int main(void)
{
    unsigned char N, i = 1;
    scanf("%hhd", &N);

    while(i < 10)
    {
        printf("%hhd * %hhd = %hhd\n", N, i, N * i);
        i++;
    }

    return 0;
}
```

```
bolthakziy@
2
2 * 1 = 2
2 * 2 = 4
2 * 3 = 6
2 * 4 = 8
2 * 5 = 10
2 * 6 = 12
2 * 7 = 14
2 * 8 = 16
2 * 9 = 18
```



```
#include <stdio.h>
```

```
int main(void)
```

```
{
```

```
    unsigned char N, i, j;
```

```
    scanf("%hhd", &N);
```

```
    for(i = 0; i < N; i++)
```

```
    {
```

```
        for(j = 0; j <= i; j++)
```

```
        {
```

```
            printf("*");
```

```
        }
```

```
        printf("\n");
```

```
    }
```

```
    return 0;
```

```
}
```

5

*

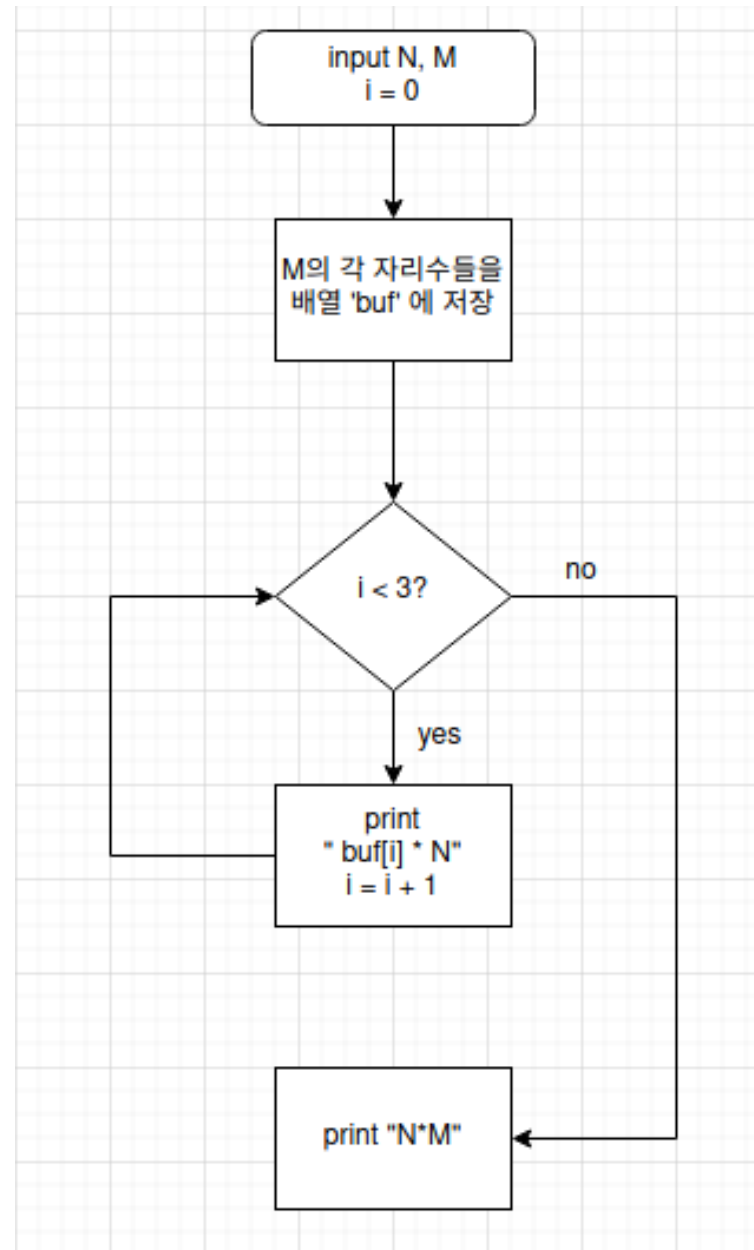
* *

* * *

* * * *

* * * * *

HW3




```

#include <stdio.h>

int main(void)
{
    unsigned int N, M;
    unsigned char i;
    unsigned int buf[3] = {0};
    scanf("%d", &N);
    scanf("%d", &M);

    buf[2] = M / 100;
    buf[1] = (M % 100) / 10;
    buf[0] = (M % 100) % 10;

    for(i = 0; i < 3; i++)
    {
        printf("%d\n", buf[i] * N);
    }
    printf("%d\n", N * M);

    return 0;
}

```

```

472
385
2360
3776
1416
181720

```

참고 자료

1. 위키피디아
2. <https://www.supercoders.in/2019/03/>
3. <https://fresh2refresh.com/c-programming/c-operators-expressions/>
4. draw.io