C Programming part 3

Bitwise operators – defined for char, short, int, long

Very commonly used in embedded system programming!

Shift right	>>	Positions are filled with zeros	Shift
Shift left	<<		
Bitwise AND	&		Logical.
Bitwise OR	|		
Bitwise exclusive OR	^		
Bitwise negation	~		

Examples:

```
int a = 11; //Decimal
int b = 0x000B; //Hexa
```

Binary:

0000 0000 0000 1011

0 0 0 B

Hexadecimal:

0x000B

int represented on 16 bits

```
char c1 = 0xF3;  //Binary: 11110011
char c2 = 0x54;  //Binary: 01010101
char c3 = c1 & c2; //Binary: 01010001  which is 0x51

char c4, c5, c6, c7;
c4 = c2 | c3;  //Binary: 01010101  which is 0x55
c5 = ~c4;      //Binary: 10101010  which is 0xAA
c6 = c4 ^ c5;  //Binary: 11111111  which is 0xFF
c7 = c6 << 3;  //Result is: 11111000  which is 0xF8
c7 = c6 >> 2;  //Result is: 00111111  which is 0x3F
```

```
C7 = 0x10;    // 00010000 in binary and 16 in decimal.
C8 = C7>>1;   // 00001000 binary and 8 decimal
C8 = C7>>2;   // 00000100 binary and 4 decimal
```

Note: Shift to the right by 1 bit means division by 2, by 2 bits means division by 4 ($=2^2$), etc.

```
C8 = C7<<1;   // 00100000 binary and 32 decimal
```

Note: Shift to the left by 1 bit means multiplication by 2, by 2 bits means multiplication by 4, etc.

```
#define MASK_BIT_3 0x08    // 0000 1000
```

Bit 0 LSB, Bit 1, Bit 2, Bit 3

```
char a = 0x55;             // 0101 0 101
char b = a | MASK_BIT_3;   // 0101 1 101
```

sets this bit to '1'!

what if I want to set a particular bit to '0'?

```
#define MASK_BIT_3_TO_ZERO 0xF7   // 1111 0 111
char a = 0x0F;                    // 0000 1 111
char b = a & MASK_BIT_3_TO_ZERO;  // 0000 0 111
```

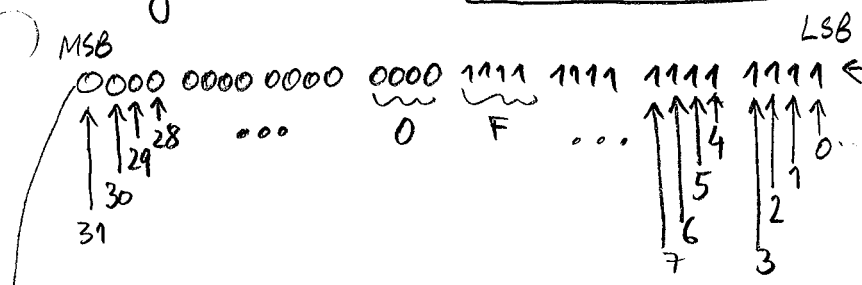
Bit 3

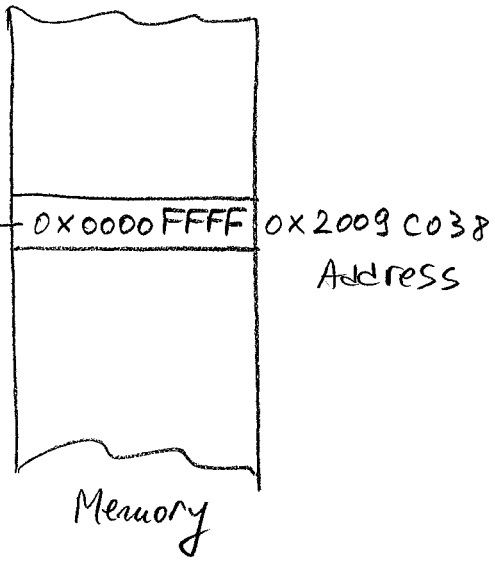
sets this bit to '0'!

Example: Recall from Lab #1:

```
FIO1SET  EQU 0x2009C038
LDR  R1, =FIO1SET
LDR  R3, [R1]
ORR  R3, #0x10000000
STR  R3, [R1]
```

Register R3 after `LDR R3,[R1]`





Register R3 after `ORR R3, #0x1000 0000`

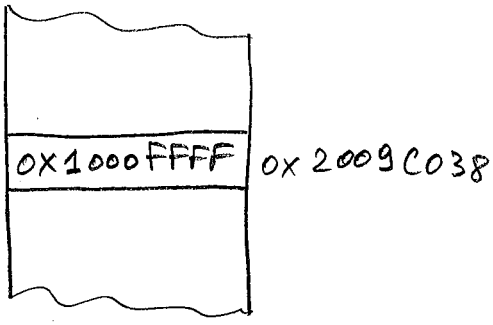
0001 0000 0000

this is basically a mask we'll use to set bit index 28 to '1' (this will drive pin P1.28 HIGH)

Bitwise OR! with mask

0001 0000 0000 0000 1111 1111 1111 1111

Memory contents after `STR R3, [R1]`



Functions - a 1st encounter

Function declaration: (usually done inside a header file .h)

```
int my_adder(int a, int b);
```

- `int` — function returns an integer
- `my_adder` — function name or identifier
- `int a, int b` — two parameters or arguments

Function description or definition: (usually done inside a source file .c)

```
int my_adder (int a, int b)
{
    int temp c = 0;
    c = a+b;
    return c;
}
```

Function call:

```
...
int one = 3;
int two = 7;
int res = 0;
res = my_adder (one, two);  //result is 10 ☺
```

Pointers - a 1st encounter

- A pointer is a variable which takes addresses as values.
- We declare a pointer like any other variable but with the difference that we use *.

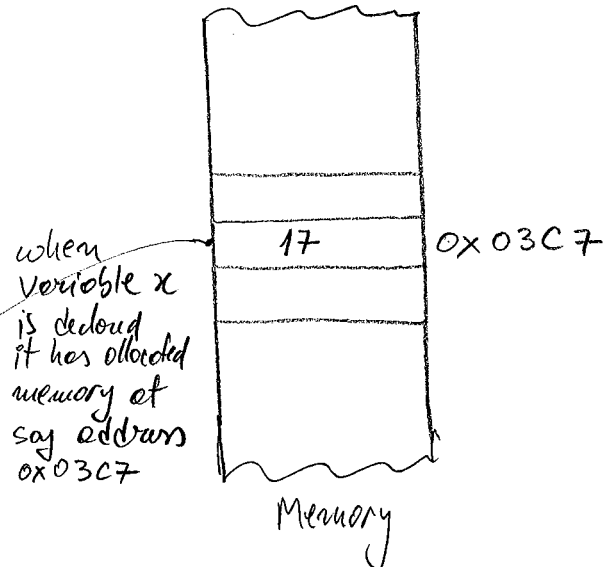
asterisk sign

Example:

```
int *p;
```

name of variable pointer

establishes that p contains addresses of memory locations where we have stored data of type int.




```
int x = 17;
p = &x;  //give p the address of x ! p stores or has value 0x03C7
```

```
int y = 3, z = 4;
y = x + 2 ;   // y gets value 19
y = *p + 2 ;  // y gets value 19
```

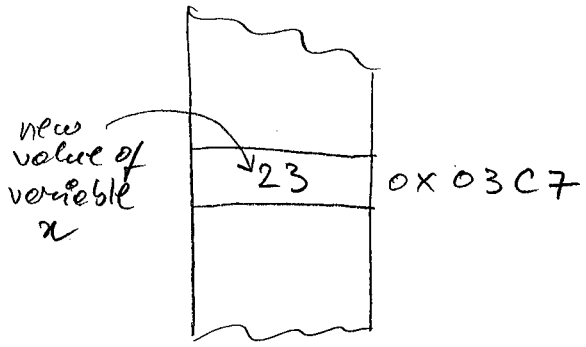
} identical assignments!

↑ Dereferencing operator ≡ provides the value stored in memory location whose address is the value of variable P!

```
p = &x;
```

↑ Referencing or address operator ≡ gives/assigns to pointer variable p a value which is the address of memory location where variable x is stored!

```
*p = 23; // changes the contents of variable x, that is
         // the contents of memory location where x is
         // stored!
```




```
z = *p + x ; // z becomes 46
```

Question

What does this do?

```
z = *&x ;  // z gets assigned value 23!
```

$*\&x \equiv x$

↑ address of variable x

contents of mem. location with this address; that is value of x!