Operatory logiczne i bitowe

- **Konwencja oznaczeń liczb w prezentacji:**
  - **Liczba dziesiętna – bez sufiksu / prefixu:**
    - **1, 3, 13, 45**
  - **Liczba binarna – prefiks „0b”:**
    - **0b01100110, 0b101, 0b1**
  - **Liczba szesnastkowa – prefiks „0x” lub sufiks „h”:**
    - **0x43, 0x55, 0x66, 0xA2, 0xFF, 0x00, EDh, 0Fh**

Operatory logiczne i bitowe

- **Podstawowe operatory języka są powszechnie znane – ważne są jednak także pozostałe z nich, w programowaniu PC rzadko stosowane.**

- **Suma logiczna i iloczyn logiczny:**
  - **|| - suma logiczna,**
  - **&& - iloczyn logiczny,**
  - **! – negacja,**
- **Operatory bitowe:**
  - **<< - przesunięcie w lewo,**
  - **>> - przesunięcie w prawo,**
  - **& - bitowy iloczyn logiczny,**
  - **| - bitowa suma logiczna,**
  - **^ - bitowa różnica symetryczna (XOR),**
  - **~ - bitowa negacja.**

Różnica: Operatory logiczne nie zaglądają do wnętrza sprawdzanego słowa, operatory bitowe to robią. W efekcie ich działanie jest bardzo odmienne.

Operatory logiczne i bitowe

- **Odmienne działanie operatorów bitowych oraz logicznych:**
  - (!0b10001100) ó (!0x8C) => 0b00000000 ó (0x00) (FALSE)
  - (!0b00000000) ó (!0x00) => 0b00000001 ó (0x01) (TRUE)
  - (~0b10001100) ó (~0x8C) => 0b01110011 ó (0x73)
  - (~0b00000000) ó (~0x00) => 0b11111111 ó (0xFF)
  - 0b10100101 || 0b10001000 => 0b00000001 (0x01) (TRUE)
  - 0b10100101 | 0b10001000 => 0b10101101 (0xAD)
  - 0b10100101 && 0b10001000 => 0b00000001 (0x01) (TRUE)
  - 0b10100101 & 0b10001000 => 0b10000000 (0x80)

- Przy założeniu, że TRUE = 1, FALSE = 0;

Operatory logiczne i bitowe

```
  01010101 (== 0x55)
| 11110000 (== 0xf0)
-------------
= 11110101 (== 0xf5)
```

```
   01010101  (== 0x55)
&  11110000  (== 0xf0)
-------------
=  01010000  (== 0x50)
```

```
   01010101 (== 0x55)
^  11110000 (== 0xf0)
-------------
=  10100101 (== 0xa5)
```

źródło: http://www.freshsources.com/19930321.HTM

Przesunięcia << oraz >>

- Przy przesunięciu w lewo (<<):
  - Na LSB wprowadzane są zawsze zera:

    (1 << 3) => (0b00000001 << 3) => 0b00001000

Przesunięcia << oraz >>

- Przy przesunięciu w prawo (>>):
  - Na MSB wprowadzane są:
    - Zera – gdy liczba zadeklarowana jest jako unsigned:

      uint8_t data = 0x60;

      (data >> 2) => (0b01100000 >> 2) => 0b00011000

    - Powielany jest bit znaku (MSB) – gdy liczba zadeklarowana jest jako signed:

      int8_t data = 0x60;

      (data >> 2) => (0b01100000 >> 2) => 0b00011000

      int8_t data = 0x80;

      (data >> 2) => (0b10000000 >> 2) => 0b11100000

Manipulacje na pojedynczych bitach

```
rejestr |= (1 << numer_bitu); //ustawienie bitu
```

Przykład:

```
uint8_t data = 0xD4;
data |= (1 << 1);
```

wartość rejestru:	**0b11010100 (0xD4)**
wartość (1 << 1):	**0b00000010 (0x02)**
suma bitowa:	**0b11010110 (0xD6)**

Manipulacje na pojedynczych bitach

```
rejestr &= ~(1 << numer_bitu); //wyzerowanie bitu
```

Przykład:

```
uint8_t data = 0xD4;
data &= ~(1 << 2);
```

wartość rejestru:	**0b11010100 (0xD4)**
wartość (1 << 2):	**0b00000100 (0x04)**
wartość ~(1 << 2):	**0b11111011 (0xFB)**
suma bitowa:	**0b11010000 (0xD0)**

Manipulacje na pojedynczych bitach

```
rejestr ^= (1 << numer_bitu); //przełączenie bitu
```

Przykład:

```
uint8_t data = 0xD4;
data ^= (1 << 0);
```

wartość rejestru: `0b11010100 (0xD4)`
wartość (1 << 0): `0b00000001 (0x01)`
suma bitowa: `0b11010101 (0xD5)`

```
data ^= (1 << 0);
```

suma bitowa: `0b11010100 (0xD4)`

Konfiguracja rejestru

```
UCSRC = (1 << URSEL) | (3 << UCSZ0);
```

gdzie: **URSEL** = 7
UCSZ0 = 1

- Składniki logicznej sumy bitowej:
  - (1 << URSEL) => (1 << 7) => 0b10000000 (0x80)
  - (3 << UCSZ0) => (3 << 1) =>
    (0b00000011 << 1) => 0b00000110 (0x06)

- Suma bitowa:

  UCSRC = (0b10000000) | (0b00000110) => 0b10000110 (0x86)

Maskowanie bitów

- Wydzielenie 4 starszych bitów rejestru:
  - 0b01011010 & 0b11110000 => 0b01010000
- Podobnie – z wyrównaniem do prawej strony:
  - (0b01011010 >> 4) => 0b00000101
  - Dla bezpieczeństwa (patrz slajd 5 – gdyby liczba zadeklarowana była jako signed) dodatkowo można wykonać:

    (0b00000101 & 0b11110000) => 0b00000101