Piotr Heinzelmann

①

a) $1111\ 1111_{(2)} =$

(16) 0xFFh

(8) 0377

(10) 255 - od razu widać :)

---

b) $1100\ 0011_{(2)} =$

011.000.011
3 0 3 $(8) = 0303_{(8)}$

0xC3h (16) =

$(10) = 0xC3_{(16)} = \text{"C"} \cdot 16 + 3 = 12 \cdot 16 + 3 =$
$= 12\cancel{0} + 36 + 36 + 3 = 195_{(10)}$

---

c) $0001.0001_{(2)}$

(8) → 00.010.001 = $021_{(8)}$

(16) → 0x11h = $0x11h_{(16)}$

(10) → 0x11 → $\text{"1"} \cdot 16 + \text{"1"} = 16 + 1 = 17_{(10)}$

---

d) $0011.1100_{(2)}$

(8) 00.111.100 = $074_{(8)}$

(16) 0011.1100 = 0x3Ch

(10) 0x3Ch → $(10) = \text{"3"} \cdot 16 + \text{"C"} = 48 + \text{"C"} = 48 + 12 = 60_{(10)}$

---

algorytm konwersji 3 cyfrowego hex na cztery oct.
(ja wiem który oct.) ABC - Hex defg - oct.

$ABC = xxxx.xxxx.xxxx_{(2)} =$

$= defg\ \ xxx.xxx.xxx.xxx_{(8)}$

16:

| | |
|---|---|
| 0000 | 0 |
| 0001 | 1 |
| ..10 | 2 |
| ..11 | 3 |
| .100 | 4 |
| 101 | 5 |
| 110 | 6 |
| 111 | 7 |
| 1000 | 8 |
| 1001 | 9 |
| 1010 | A |
| 1011 | B |
| 1100 | C |
| 11.1 | D |
| 111. | E |
| 1111 | F |

8

| | |
|---|---|
| ... | 0 |
| ..1 | 1 |
| .1. | 2 |
| .11 | 3 |
| 1.. | 4 |
| 1.1 | 5 |
| 11. | 6 |
| 111 | 7 |

Bx Bx+1 Bx+2



liczba "16" BCD = 567
liczba "8" efgh = ?
+9 +10 +11 +12

1) Wczytaj najmłodszą "16" D do rejestru AH
MOV AH, [BX+2]   A: ....|xxxx|....|....
AH | AL

2) Przesuń AX o 3 bity w prawo
(chyba że wolisz dzielić przez 8)
MOV CL, 03;   A: ....|...x|xxx.|....
SHR AX, CL;

3) Przesuń AL o 5 bitów w prawo
MOV CL, 05;   A: ....|...x|....|.xxx   h = 7
SHR AL, CL
AH | AL

4) Zapisz najmłodszą liczbę wyniku
MOV [BX+12], AL;

5) Przesuń AX o 1 bit w prawo
SHR AX, 1   A: ....|....|x...|..xx

6) Załaduj kolejną liczbę do AH
MOV AH, [BX+1]   A: ....|xxxx|x...|..xx

7) Przesuń AX o 2 bity w prawo
SHR AX, 1; SHR AX, 1;   A: ....|..xx|xxx.|....
AH | AL

8) Przesuń AL o 5 bitów w prawo
SHR AL, CL   A: ....|..xx|....|.xxx

9) zapisz wynik do drugiego znaku "8"
g = AL

MOV [BX+4], AL        A: ....|..xx|....|.xxx

10) przesuń A o 2 bity w prawo
SHR AX,1; SHR AX,1;

A: ....|....|xx..|...x

11) wczytaj 3 znak "16" do AH
MOV AH,[BX]

A: ....|xxxx|xx..|...x

12) przesuń AX o 1 bit w prawo
SHR AX,1        A: ....|xxx|xxx.|....

13) przesuń AL w prawo o 5 bitów
SHR AL,CL        A: ....|xxx|.....|.xxx

14) zapisz wynik AL w liczbie f
MOV [BX+10],AL

15) zapisz liczbę e
MOV [BX+9],AH

---

rozwiązanie w C
char B; char C; char D;
char d; char e; char f; char g;
int sum= (((B*16)+C)*16)+D; // *16 to to samo co <<4 lub SHR
g=sum%8;
sum=sum>>3;  // sum>>3 to to samo co sum/8 tylko szybciej.
f=sum%8;
sum=sum/8;
e=sum%8;
d=sum>>3;
zamiast "/" można użyć A >> 3;

algorytm jest podobny i polega na wyliczaniu wartości przez mnożenie liczby przez wagę, a następnie obliczenie kolejnych znaków od prawej storny przez dzielenie wartości z resztą.

tutorialspoint | Online Assembly Compiler

Execute | Beautify | Share | Source Code | Help | Login ×

```
section .text
    global _start        ;must be declared for using gcc
_start:                      ;tell linker entry point
    mov ah,[D]
    mov CL,3
    SHR AX,CL
    MOV CL,05
    SHR AL,CL
    ADD AL,0x30 ; zamiana na ascii przez dodanie x30
    MOV [h],AL
    SHR AX,1;
    MOV AH,[C]
    SHR AX,1
    SHR AX,1
    SHR AL,CL
    ADD AL,0x30 ; zamiana na ascii
    MOV [g],AL
    SHR AX,1
    SHR AX,1
    MOV AH,[B]
    SHR AX,1
    SHR AL,CL
    ADD AL,0x30 ; ...
    MOV [f],AL
    ADD AH,0x30
    MOV [e],AH
    MOV AX,[B]
    ADD AX,0x30 ; zmiana na ascii
    MOV [B],AX
    MOV AX,[C]
    ADD AX,0x30
    MOv [C],AX
    MOV AX,[D]
    ADD AX,0x30
    MOv [D],AX


    mov edx, len     ;message length
    mov ecx, B    ;message to write
    mov ebx, 1      ;file descriptor (stdout)
    mov eax, 4      ;system call number (sys_write)
    int 0x80        ;call kernel
    mov eax, 1      ;system call number (sys_exit)
    int 0x80        ;call kernel

section .data
B db 5
C db 6
D db 7
xx db 0x3d
e db 0
f db 0
g db 0
h db 0


msg db  'Hello, world!',0xa ;our dear string
len equ $ - msg         ;length of our dear string
```

Terminal

```
567=2547Hello,
```

Dlaczego hexadecymalny?

proste - aby wyświetlić wartości liczbowe zapisane w pamięci musimy ją wielokrotnie dzielić z resztą przez podstawę, czyli dla liczb „zwykłych" przez 10 - co jest kłopotliwe dla maszyn przetwarzających liczby o podstawie 2.

Natomiast dzielenie przez 2, 4, 8, 16 i kolejne potęgi liczby 2 polega na przesuwaniu bitów w prawo o 1, 2, 3 lub 4 pozycje.

Łatwiej jest wyświetlić liczbę przy podstawie 8 lub 16. podstawa 16 ma tę zaletę że ma zawsze 2 znaki dla każdych 8 bitów, podstawa 8 może mieć 2 lub 3 znaki.

Ponadto zapis hexadecymalny od razu widać jeśli pojawi się znak "10" A "11" B lub następne.

zapis 16 jest po prostu wygodny. A poza tym jest trochę elitarny, co ma także źródło przy powstaniu H4CK-mowy czyli zapisu słów z wykorzystaniem cyfr i znaków. bywa to używane do konstruowania haseł np.: "h@sło" zamiast "hasło".

Zadanie 3

| | | | |
|---|---|---|---|
| 0 | ···· | ×16 | 0 |
| 1 | ···1 | | 16 |
| 2 | ··1· | | 32 |
| 3 | ··11 | | 48 |
| 4 | ·1·· | | 64 |
| 5 | ·1·1 | | 80 |
| 6 | ·11· | | 96 |
| 7 | ·111 | | 112 |
| 8 | 1··· | | 128 |
| 9 | 1··1 | | 144 |
| A | 1·1· | | 160 |
| B | 1·11 | | 176 |
| C | 11·· | | 192 |
| D | 11·1 | | 208 |
| E | 111· | | 224 |
| F | 1111 | | 240 |

$-58_{(10)} \rightarrow$

$58_{(10)} = 48+10 = 0x30 + A = 0x3A$

0x003A

0x0000 003A

$0x3A = \cdot\cdot 11.1\cdot 1\cdot$

$-58 =_{(u1)} -0x3A = 11\cdot\cdot.\cdot 1\cdot 1 = 0xC5$ $= \underline{0xC5} = 0xFFC5$

0xFFFF FFC5

---

$-128_{(10)} \rightarrow$ $128_{(10)} = 0x80+0 = 0x80 = 1\cdots.\cdots$

$-128_{(u1)} =$ $= \cdot 111.1111 = \underline{0x7F}$

$-128_{(u1)} =$ 0xFF7F / 0xFFFF FF7F

---

$-1023_{(10)}$ $u_1 \rightarrow$ ( $1024 = 2^{10}$ $1023 = 0011.1111.1111$

0x3 F.F

$-1023 =$ 0xFC00

$-1023_{(u1)} =$ 0xFC00 / 0xFFFF FC00

---

Zadanie 4 $-58\ u_2 = -58\ u_1 + 1 =$ 0xFFC5+1 = $\underline{0xFFC6}$

0xFFFF FFC5+1=0xFFFF $\underline{FFC6}$

$58\ u_2 = 58 u_1 =$ 0x3A = 0x$\underline{003A}$

0x$\underline{0000\ 003A}$

$-128_{(u2)} = -128_{(u1)}+1 =$ 0xFF7F+1 = 0x$\underline{FF80}$

= 0x$\underline{FFFF\ FF80}$

$-1023_{(u2)} = -1023_{(u1)}+1 =$

0xFC00 +1 = 0x$\underline{FC01}$

0x$\underline{FFFF\ FC01}$

Zad 5 Czy dodanie liczb 1111 1111 da poprawny wynik?
Zobaczmy:

11111111 to (1)1111111
−1111111 −1
−1111110 obracamy
−0000001
= −1 (10)

−1 + −1 = −2

| | | |
|---|---|---|
| | | 2 |
| 0 ... | 0010 | 1 |
| 0 ... | 0001 | 1 |
| 0 ... | 0000 | 0 |
| 1 ... | 1111 | −1 |
| 1 | 1110 | −2 |
| 1 | 1101 | −3 |

Suma sumatora

```
   1111 1111
+  1111 1111
(1) 1111 1110  = 1111 1110
               = −111 1110 (−1)
                 −111 1101 obracamy
                 −000 0010

               =   −2
```

Wynik jest prawidłowy, $-1+(-1)=-2$.

Uzupełnienie do 2 jest najpopularniejsze, ponieważ daje poprawne wyniki przy dodawaniu/odejmowaniu liczb dodatnich i ujemnych, o ile nie przekroczymy zakresu.

Zad 5a czy dodanie liczb 1000 0000 i 10000000 da poprawny wynik?

1000 0000 U2 = (−) 000 0000 => (−1) = (−)0 1111111 odwracamy

−10000000 czyli jest to liczba −128.

dodanie dwu liczb −128 + (−128) da w wyniku −256. czyli wynik nie zmieści się zakresie 8bitowym.

  1000 0000
+ 1000 0000
(1) 0000 0000 = 0

wynik nie jest prawidłowy, ustawione prawdopodobnie flagi przeniesienia i

CF OF
+ −

na 8086 ustawione Carry Flag CY +
NIE ustawione Overflow Flag NV −

OF+ CF+
przy dodawaniu 100000 flagi OV CY

Zad 6 spakowany BCD 8421

BCD

$$\begin{array}{rl} & \overset{1}{1000}\ 1000 \\ + & 0001\ 1000 \\ \hline & 1010\ 0000 \end{array} \qquad \begin{array}{r} 88 \\ +18 \\ \hline 10.6 \end{array}$$

AF

Krok 1) dodawanie zwykłe

ale jesteśmy w BCD i musimy sprawdzić flagę AF!

Krok 2) sprawdzamy flagę AF

Krok 3) jeśli ustawiona flaga to:

– komenda AAA – wyrównuje po dodawaniu BCD.

+ AAA ⇒

AAA dodaje +6 gdy wartość w AL > 9 i wyzerowuje 4 starsze bitów.

jeśli AL było korygowane (flaga AF +)

po korekcie dostaniemy 06 z przeniesieniem.

czyli (1)06