Konwersje liczb całkowitych bez znaku do postaci ciągów tekstowych (system szesnastkowy i dziesiętny)

Zarówno dane (niezależnie od ich typu), jak i instrukcje procesora przechowywane są w pamięci w postaci liczb. Bardziej konkretnie: w postaci odpowiedniej długości ciągów „0" i „1", które to dwa stany odpowiadają ustawionym / zgaszonym przerzutnikom pamięci statycznej, lub zgromadzonemu ładunkowi w kondensatorach komórek pamięci dynamicznej, itp.

Aby wydrukować dane z pamięci komputera potrzebna jest procedura konwersji z postaci binarnej na bardziej zrozumiałą dla człowieka.

Konwersja liczby z systemu binarnego na szesnastkowy jest bardzo prosta (podstawy obu systemów są potęgami dwójki). Dodatkową zaletą systemu szesnastkowego jest wykorzystanie tylko dwóch cyfr do zapisu wartości zmiennej jednobajtowej bez znaku (00…FF), podczas gdy w systemie dziesiętnym, w ogólnym przypadku, trzeba użyć trzech cyfr (000 … 255).

„Ciąg" binarny wystarczy podzielić (od prawej – najmłodszej strony) na czterobitowe grupy (*półbajty – nibbles*), a następnie każdej z nich przyporządkować symbol odpowiadający cyfrze systemu szesnastkowego.

Np. 14(16)-bitowe „słowo" 11010111001111_{BIN} można „podzielić" na czterobitowe części:
0011_{BIN} 0101_{BIN} 1100_{BIN} 1111_{BIN} odpowiadające liczbie: $35CF_{HEX}$ w systemie szesnastkowym.

W praktycznym „algorytmie" konwersji należy uwzględnić kolejność znaków odpowiadających cyfrom 0 - 9 oraz literom A - F w tablicy ASCII. Cyfrom 0 - 9 odpowiadają znaki o numerach 48 - 57.
Kolejne 7 znaków (: ; < …) należy pominąć (58 - 64), a cyfrom systemu szesnastkowego A - F odpowiadają kody od 65 do 70.

Przykładowe pseudokody różnych „algorytmów" konwersji liczba -> znak (znaki) ASCII

1. Konwersja półbajtu

Zał.: argument przekazany w czterech młodszych bitach rejestru *%al*, cztery starsze bity są wyzerowane.
Zwracana wartość: numer znaku ASCII (8bit): również w rej. *%al*.

```
convert_nibble:

  if (%al < 10) %al += 48 else %al += 55

  return %al
```

(oczywiście nie jest to jedyny sposób – patrz p. 5.)

2. Konwersja bajtu, (na dwa znaki ciągu tekstowego ASCII), wykorzystująca powyższą funkcję:

Zał.: argumenty: liczba-bajt w rejestrze *%al*, adres zapisu wyniku konwersji (dwóch bajtów) w *%rdi*.

```
convert_byte:

  temp = %al       //kopia wartości %al, %al będzie „niszczony"
  %al &= 0x0F      //pozostaw tylko 4 młodsze bity w %al

  convert_nibble   //konwersja młodszego półbajtu %al

  %ah = %al        //wynik konwersji (bajt w %al) do %ah*
  %al = temp
  %al >>= 4        //przesuń starszy półbajt na miejsce młodszego (4 bity w prawo)
  convert_nibble   //konwersja starszej połówki bajtu

  @(%rdi)=%ax      //*zapisz oba znaki: starszy w %al, młodszy w %ah
                   //odpowiednio pod adresy:     %rdi    i      %rdi+1
                   //porządek bajtów: little endian!
  Return
```

3. Konwersja wielobajtowych typów danych

Zał.: argumenty: liczba w rejestrze *%a* (*%rax* odpowiedniej „długości”), adres zapisu w *%rdi* (zaczynając od najmłodszej pozycji), rozmiar zmiennej podlegającej konwersji (w bajtach: 2, 4 lub 8) w *%ecx*. Funkcja wykorzystuje poprzednią procedurę *convert byte* (i pośrednio *convert_nibble*).

```
convert:

  for (%ecx=2/4/8; %ecx>0; %ecx--){

  temp=%a           //kopia rej. %a
  convert_byte      //wywołaj konwersje najmłodszego bajtu
  temp >>= 8        //ustaw kolejny bajt do konwersji na miejscu najmłodszego
  %a=temp           //przygotuj argument dla kolejnego wywołania convert_byte
  %rdi -= 2         //przesuń wskaźnik miejsca zapisu o dwa bajty
                    //kierunek „w ciągu tekstowym”: od prawej do lewej
  }
return
```

Jako *temp* w p. 2. i 3. mogą służyć dowolne nieużywane (również w funkcjach *convert_byte* i *convert_nibble*!) rejestry ogólnego przeznaczenia (odpowiedniego rozmiaru). Ew. można zaalokować odpowiedni obszar pamięci.

Uwaga: kod powyższych funkcji można uprościć/zoptymalizować np. wykorzystując licznik pętli (*%ecx*) do adresowania miejsca zapisu (patrz p. 6.) oraz rozkaz *xchg* do ulokowania obu znaków reprezentujących bajt w odpowiednim porządku w rejestrze *%ax*. Zamiast ustawionej stałej liczby iteracji, obliczenia można powtarzać dopóki w *%a* jest wartość niezerowa.

4. Konwersja wielobajtowej liczby na system dziesiętny

… generalnie jest również prosta, przedstawiony tutaj „algorytm” wymaga jednak obliczania reszty z dzielenia (mod 10).

Operacja dzielenia jest stosunkowo skomplikowana (w porównaniu np. do dodawania) i większość prostych mikrokontrolerów 8-bitowych (8048, MC6805) i procesorów (np. MC6800, 8085/Z80) jej sprzętowo nie wykonuje. Należy wtedy napisać dodatkową funkcję. W architekturze x86 modulo obliczane jest podczas wykonywania dzielenia.

Proszę zwrócić uwagę na składnię instrukcji *div* i miejsce zapisywanych wyników. Np. *div* dzieli 64-bitową liczbę (bez znaku) zapisaną w parze rejestrów: *%edx : %eax* (high : low) przez wartość (32bit, bez znaku) **jedynego argumentu** tego rozkazu (32-bitowego rejestru lub adresu pamięci). W wyniku operacji, 32-bitowy iloraz zapisywany jest w *%eax*, a modulo w *%edx*.

Ponieważ zakładamy, że dzielna jest 32-bitowa (np. typ unsigned int), przed wykonaniem *div* rejestr *%edx* należy wyzerować. Jeśli się o tym zapomni wynik będzie błędny. Gdy obliczony iloraz przekroczy 2^{32}-1 (max. unsigned int) - zgłoszony zostanie wyjątek – w Linuxie: błąd obliczeń… zmiennoprzecinkowych (sic!).

Analogicznie sprawa wygląda z użyciem rejestrów 16- i 64-bitowych. Dzielenie „ośmiobitowe” wymaga jednak umieszczenia dzielnej w rejestrze *%ax* (tj. parze *%ah* : *%al*), dzielnika w ośmiobitowym argumencie rozkazu, wynik zwracany jest w: *%ah* – modulo i w *%al* – iloraz.

Zał.: jak poprzednio: liczba w *%a*, adres zapisu (zaczynamy najmłodszej pozycji) w *%edi*:

```
convert2dec:

  do {
    %q = %a  /  10        //%q - iloraz
    %m = %a mod 10        //%m – modulo 10 – wartość: od 0 do 9
    %m |= 48              //przesunięcie ASCII
    @(%edi) = %m          //zapisz znak
    %edi--                //przesuń wskaźnik zapisu
    %a = %q
  } while (%a != 0)       //powtarzaj dopóki masz co dzielić
```

5. Look-Up Table (LUT)

Inną metodą konwersji, eliminującą konieczność wykonywania większości obliczeń (za wyjątkiem np. obliczania adresów) jest wykorzystanie stablicowanych, gotowych elementów wyjściowego ciągu tekstowego.

Idea jest bardzo prosta:

wartość argumentu funkcji jest jednocześnie numerem (indeksem) elementu tablicy (Look-Up Table), w którym to elemencie przechowywana jest żądana informacja wyjściowa – zwracana przez funkcję.

„Idąc dalej" – na podstawie jednego znaku ASCII (bajtu) sterownik wyświetlacza (albo drukarki tekstowej) „wybiera" z kolejnej tablicy – generatora znaków - jego graficzną reprezentację – font. Karta VGA (np. w trybie graficznym 320 x 200 x 256 kolorów) na podstawie jednej „wartości" piksela (bajt) odczytuje z wcześniej zaprogramowanej, większej palety, wartości trzech składowych koloru R, G, B itp., itd.

Tym sposobem można również zastąpić obliczanie wartości funkcji np. sin – odczytem „gotowej" wartości z tablicy, której element wybiera argument tablicowanej funkcji. Oczywiście jest to reprezentacja dyskretna – dokładne wartości funkcji są wyznaczone tylko dla zbioru konkretnych argumentów, w pewnym przedziale.

Wadą tablicowania może być (nie musi!) sama tablica, zajmująca pewien obszar pamięci (tutaj: pamięci RAM - danych, w mikrokontrolerze np. fragment nieulotnej pamięci programu). Wszystko zależy od konkretnego zastosowania i przyjętego kryterium optymalizacji (rozmiar kodu, rozmiar danych, szybkość wykonania, rozdzielczość - dyskretyzacja tablicy…).

Np. wartość półbajtu (4 bity - od 0 do 15) adresuje jeden z szesnastu (2^4) elementów tablicy, spod którego odczytywany jest znak – kodowany zgodnie tablicą ASCII. Tablica zajmuje obszar szesnastu bajtów i przykładowo może mieć postać:

```
lut8:  .ascii "0", "1", ... ,"9", "A", ... , "E", "F"
```

natomiast odpowiednik procedury *convert_byte* (założenia jak w punkcie 2.) wykorzystujący LUT:

```
convert_byte:

  %dh = %al                //kopia argumentu
  %eax &= 0x0F             //zostaw tylko 4 młodsze bity
                           //do adresowania używamy rej. 32/64 bitowych
  %al = lut8[%eax]         //odczytaj znak z tablicy

  %al ⇔ %dh                //zamień (eXCHanGe) wartości %al z %dh
                           //(wynik konwersji do %dh)
                           //docelowo znak odpowiadający młodszemu półbajtowi
                           //zostanie zapisany pod starszym adresem

  %al >>=4                 //starsza połówka bajtu adresuje tablicę
  %dl = lut8[%eax]         //odczytaj drugi (starszy) znak
  @(%edi)=%dx              //zapisz w odpowiedniej kolejności w pamięci

return
```

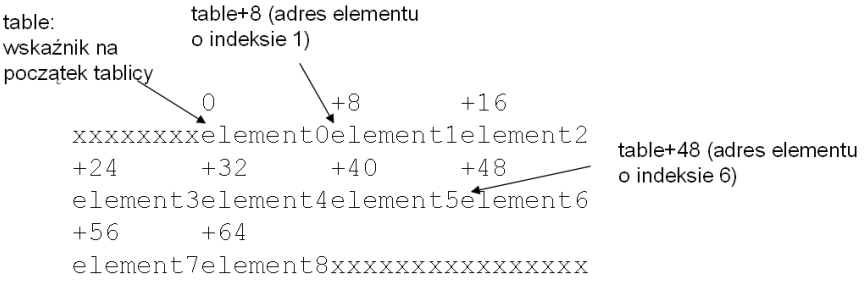
W tym przypadku powyższa metoda eliminuje konieczność np. sprawdzania, w którym podzakresie (0-9 czy 10-15) mieści się argument funkcji oraz uwzględniania stosownych przesunięć (względem tablicy ASCII).

Tablicę można również zbudować z 256 elementów dwuznakowych – tak, aby pozwalała „w locie" zamienić cały bajt na fragment wyjściowego ciągu tekstowego. Kosztem większej tablicy – jeszcze bardziej uprości się sama procedura i skróci czas konwersji.

6. Adresowanie tablic

„Ułożenie” w pamięci kolejnych elementów 8-bajtowych:

```
table: .quad element0, element1, element2, element3, element4, ...
```



Czyli: adres *i*-tego elementu (licząc od zera) = adres_początku_tablicy + indeks * rozmiar_elementu (w bajtach).

Pełny tryb adresowania komórek pamięci przez procesor zgodny z X86 dopuszcza użycie dwóch rejestrów (jeden jest skalowany) oraz stałej liczby (składnia AT&T):

```
mov  stała(%rejestr1, %rejestr2, skala) , %rejestr3                    (*)

adres komórki = stała + %rejestr1 + %rejestr2 * skala                  (**)
```

gdzie:

%rejestr2 - najczęściej zawiera indeks - numer elementu tablicy liczony od zera skalowany przez rozmiar elementu *skala*,

skala - stała-potęga dwójki (1, 2, 4, 8) – zależy od rozmiaru (w bajtach) elementów tablicy (mnożenie wewnętrznie jest zastępowane przesunięciem bitowym w lewo),

%rejestr1 - może być przydatny przy adresowaniu np. obszarów alokowanych dynamicznie, tablic dwuwymiarowych itp.,

stała – np. etykieta tablicy = adres pierwszego (indeks = 0) elementu statycznego bloku danych,

%rejestr3 – miejsce docelowe, będzie zawierał dane rozpoczynające się od obliczonego (**) adresu.

Uwaga 1. Podczas adresowania tablic nie muszą być wykorzystywane wszystkie powyższe elementy. np. odczyt wartości elementu o indeksie w *%rax*, z jednowymiarowej, statycznej tablicy *table* złożonej z elementów 64-bitowych ma postać:

```
mov table( , %rax , 8) , %rdx        # %rdx = dane (8 kolejnych bajtów) spod adresu: table+%rax*8
```

Uwaga 2. Z pamięci zostaje odczytana (lub zapisana – po zamianie operandów miejscami) liczba bajtów odpowiadająca rozmiarowi *%rejestru3* (*) - np. *%eax* – 4 bajty, *%ax* – 2 bajty.

W niektórych przypadkach (np. zapis do pamięci stałej wartości - możliwe kodowanie w 1, 2, 4 lub 8 bajtach), aby jednoznacznie określić typ danych należy dodać do instrukcji odpowiedni sufiks:
q - 64, **l** - 32, **w** - 16, **b** - 8 bitów.

`mov`**`b`** `$3 , etykieta` - oznacza zapis jednego bajtu: 00000011 pod wskazany adres-etykietę
`mov`**`w`** `$3 , etykieta` - zapis słowa: 00000000 00000011 zaczynając od adresu *etykieta* (w X86 - porządek bajtów *little endian*).

Adres w instrukcji np. `mov $3 , etykieta` sam w sobie nie niesie informacji o typie danych (liczbie zapisywanych bajtów), więc kompilator *as* zgłosi błąd podczas tworzenia pliku pośredniego (*.o*).

Uwaga 3. Procesor pracuje w trybie *long mode* - do adresowania komórek pamięci używa się rejestrów 32- lub 64-bitowych.

7. Zadania na laboratoria 3. i 4. ew. dokończenie (rozbudowa) w sprawozdaniu.

W pliku *int2str*.s uzupełnić / napisać w asemblerze funkcje:

1. *convert_nibble* (p. 1.).

2. *convert_byte* (p. 2.).

3. Wykorzystując podprogramy 1. i 2. napisać funkcję *convert* (dla typów wielobajtowych)

4. Sprawdzić poprawność konwersji różnych typów całkowitoliczbowych (bez znaku).

5. Zmienić „algorytm" konwersji półbajtów na wykorzystujący tablicę LUT – jednoznakową (p. 5.)

6. Analogicznie do p. 5. utworzyć tablicę złożoną z 256 dwubajtowych elementów, np.:

```
lut_16: .ascii "00", "01", "02", "03", …, "FA", "FB", "FC", "FD", "FE", "FF"
```

pozwalającą konwertować w jednym kroku cały bajt na odpowiadający jego wartości (*hex*) dwuelementowy ciąg tekstowy. Napisać odpowiednią funkcję i wykorzystać ją do konwersji liczb wielobajtowych.

Przeanalizować zalety i wady, zyski i straty w rozwiązaniach z p. 3., 5. i 6..

7. Napisać funkcję konwersji liczby na ciąg tekstowy reprezentujący jej wartość w systemie dziesiętnym (p. 4.).