lab_4 - Instrukcja do ćwiczenia

Teoria (lab_4.pdf):

Zadanie polega na wyświetleniu w postaci liczb szesnastkowych danych o dowolnej wielkości (**byte**, **word**, **long** i **quad**). Punktem wyjścia jest funkcja **byte2hex** opracowana w trakcie wcześniejszych zajęć (lab_2), zamieniająca bajt na dwie cyfry szesnastkowe. Konieczne będzie więc stworzenie funkcji dokonującej dekompozycji danych na sekwencję bajtów, które po konwersji do postaci znakowej i zapisaniu uzyskanych znaków w odpowiedniej kolejności, utworzą zapis w postaci szesnastkowej.

Zamiana bajtu na dwie cyfry szesnastkowe może być dokonana na wiele sposobów. Wersja oryginalna (lab_2) dokonuje podziału bajtu na dwie połówki, następnie każda z uzyskanych liczb (**0..15**) jest zamieniana na odpowiednią cyfrą szesnastkową poprzez użycie odpowiedniej formuły – wybór odpowiedniej zależy od tego, czy liczba jest mniejsza od **10** czy też nie.

Wersja oryginalna (ver. 1):

```
      .type byte2hex,@function
byte2hex:
      MOVB  %al, tmp

      MOVB  tmp,%al
      ANDB  $0x0F,%al         # first nibble
      CMPB  $10,%al
      JB    digit1
      ADDB  $('A'-10),%al
      JMP   insert1
digit1:
      ADDB  $'0',%al
insert1:
      MOVB  %al,%ah

      MOVB  tmp,%al           # second nibble
      SHR   $4,%al
      CMPB  $10,%al
      JB    digit2
      ADDB  $('A'-10),%al
      JMP   insert2
digit2:
      ADDB  $'0',%al
insert2:
      RET
```

Kod funkcji zawiera **17** instrukcji (instrukcja zaznaczona na czerwono może być usunięta – wtedy będzie ich **16**), wykorzystywane dane to **0/1** bajt (zmienna **tmp** to jeden bajt, ale można zamiast niej użyć rejestru 8-bitowego).

Wersja druga eliminuje wariantową zamianę liczby **0..15** na znak (cyfrę szesnastkową) poprzez zastosowanie tablicy jednowymiarowej **hex_digit** o **16** elementach. Konwertowana liczba jest

indeksem tablicy, zaś zawartość tablicy o tym indeksie jest szukanym znakiem (cyfrą szesnastkową).

Wersja alternatywna (ver. 2):

```
      .type byte2hex,@function

byte2hex:
      MOVB  %al, tmp

      MOVB  tmp, %al
      ANDB  $0x0F, %al              # first nibble
      MOVZX %al, %rbx               # rbx = al; zeros in empty space
      MOVB  hex_digit(%rbx), %ah    # ah = hex_digit[ rbx ]

      MOVB  tmp, %al                # second nibble
      SHR   $4, %al
      MOVZX %al, %rbx               # rbx = al; zeros in empty space
      MOVB  hex_digit(%rbx), %al    # al = hex_digit[ rbx ]
      RET
```

W tym przypadku kod funkcji to **10** instrukcji (instrukcja zaznaczona na czerwono może być usunięta – wtedy będzie ich **9**), wykorzystywane dane to **16/17** bajtów – tablica **hex_digit** (+ ewentualnie **tmp**).

Trzecia wersja wykorzystuje podobne podejście, ale bez dekompozycji bajtu na połówki – indeks jest liczbą z zakresu **0..255**, więc tablica jednowymiarowa **hex_digits** zawiera **256** elementów w postaci dwuznakowych napisów, będących wartością liczby (indeksu) zapisaną w postaci szesnastkowej.

Wersja alternatywna (ver. 3):

```
      .type byte2hex,@function

byte2hex:
      MOVZX %al, %rbx                   # rbx = al; zeros in empty space
      MOVW  hex_digits(,%rbx,2), %ax    # ax = hex_digits[ rbx ]
      RET
```

Tym razem kod funkcji to **3** instrukcje, zaś dane zajmują **512** bajtów (tablica **hex_digits**).

Ponieważ wszystkie funkcje mają te same nazwy, nie mogą być widoczne jednocześnie dla kompilatora. Ukrycie zbędnych elementów uzyskano poprzez użycie dyrektyw kompilacji warunkowej (**.ifdef**/**.endif**) – zdefiniowanie jednego z symboli (**FUNC_V1**, **FUNC_V2** lub **FUNC_V3**) w trakcie kompilacji pozwala na wykorzystanie odpowiedniej funkcji łącznie z używanymi przez funkcję danymi. Definiowanie symbolu dokonywane jest poprzez użycie dodatkowej opcji **`--defsym SYMBOL=VALUE`** na etapie kompilacji.

Dana której wartość ma być wyświetlona w postaci szesnastkowej powinna znaleźć się w rejestrze procesora – takim, w którym się zmieści. Ponieważ największa dana to **8** bajtów to

konieczne jest użycie rejestru 64-bitowego (np. **rax**). W takim przypadku dana **quad** wypełni rejestr całkowicie, zaś użycie innych danych (**long**, **word**, **byte**) wypełni tylko mniej znaczące części rejestru (**eax**, **ax**, **al**). Oznacza to konieczność konwersji danych począwszy od najmniej znaczącego bajtu (**al**) i w konsekwencji zapis liczby od najmniej znaczących cyfr do najbardziej znaczących (od prawej do lewej).

Ponieważ zapis danych o różnej wielkości (rozmiarze) wymaga różnej liczby znaków (**byte**=**2**, **word**=**4**, **long**=**8**, **quad**=**16**) - by zapis odzwierciedlał wielkość danych – konieczne jest przekazanie do funkcji adresu miejsca przeznaczonego na zapis znaków i adres ten musi uwzględniać sposób zapisu (od prawej do lewej), zmienną liczbę znaków oraz to, że znaki zapisywane są parami (dwie cyfry szesnastkowe jednocześnie). Modyfikacją adresu miejsca zajmie się sama funkcja – należy tylko zadba c o to, aby początkowa wartość była prawidłowa – szczegółowo pokazano to w końcowej części opisu ćwiczenia (**lab_4.pdf**).

Algorytm (idea):

- pętla **for k = 0 to size – 1** (size to rozmiar danych – [1,2,4,8])
    - pobranie bajtu **B0** (najmniej znaczącego)
    - konwersja bajtu na dwa znaki hex
    - zapis dwóch znaków do pamięci (adres zawarty w **%rdi**)
    - zmniejszenie **%rdi** o **2** (przejście o 2 pozycje w lewo)
    - przesunięcie **B1** na miejsce **B0**, **B2** na miejsce **B1**, itd.
    - zwiększenie **k** o **1**

Algorytm (implementacja):

```
#-----------------------------------------------------------------
# num2hex - converts number to hexadecimal string
#     Arguments:  %rax - number (%al, %ax, %eax)
#                 %cl - size of number (1,2,4,8)
#                 %rdi - address (where to put hex digits)
#-----------------------------------------------------------------

      .type num2hex, @function
num2hex:
      mov %rax, %rdx          # store original value in %rdx
next_byte:
      call byte2hex           # convert byte in %al to two hexdigits (in %ax)
      movw %ax, (%rdi)        # store digits in memory (string)
      sub $2, %rdi            # move two chars to the left
      shr $8, %rdx            # shift original value right
      mov %rdx, %rax          # copy value to %rax
      dec %cl                 # size--;
      jnz next_byte           # more bytes to convert

      ret
```

Nowe instrukcje:

- **MOVZX** reg8, reg64 - przeniesienie zawartości rejestru 8-bitowego (reg8) do rejestru 64-bitowego z jednoczesnym wypełnieniem pustych miejsc zerami – tzw. **z**ero e**x**tend)

Nowe dyrektywy:

- **.ifdef** – początek bloku dla kompilacji warunkowej
- **.endif** – koniec bloku dla kompilacji warunkowej

Praktyka (lab_4.s):

Dane:

- **big_hex_str** – łańcuch znaków przeznaczony na zapis liczby po konwersji do postaci szesnastkowej
- **hex_digit** – tablica znaków (cyfr szesnastkowych) wykorzystywana w wersji 2 funkcji byte2hex (16-elementowa)
- **hex_digits** – tablica par znaków (licz w postaci szesnastkowej) wykorzystywana w wersji 3 funkcji byte2hex (256-elementowa)
- **varb** – dana o rozmiarze 1 bajta
- **varw** – dana o rozmiarze 2 bajtów
- **varl** – dana o rozmiarze 4 bajtów
- **varq** – dana o rozmiarze 8 bajtów

Kod:

- **program główny**
  - wyświetlenie liczb z zakresu 0..255 w postaci szesnastkowej, w formie tablicy 16x16 – to samo co w programie lab_2). Konwersja jest dokonywana przez jedną z trzech wersji funkcji byte2hex
  - wyświetlenie wartości zmiennej varb
  - wyświetlenie wartości zmiennej varw
  - wyświetlenie wartości liczby varl
  - wyświetlenie wartości liczby varq
  - zakończenie działania programu
- **num2hex** – funkcja konwertująca daną o rozmiarze [1,2,4,8] bajtów na postać szesnastkową (wywołuje funkcję byte2hex)
- **byte2hex** – funkcja konwertująca pojedynczy bajt na postać szesnastkową (dwie cyfry szesnastkowe)

Działania:

1. C (Compile) – polecenie: **as --defsym FUNC_V1=1 -o lab_4.o lab_4.s**
2. L (Link) – polecenie: **ld –o lab_4 lab_4.o**
3. R (Run) – polecenie: **./lab_4**
4. Wyświetlone liczby w postaci szesnastkowej powinny wyglądać identycznie jak w przypadku **lab_2** (efekt końcowy zajęć).
5. Sprawdzamy efekty działania kolejnych wersji funkcji **byte2hex** (można też zwrócić uwagę na wielkości plików wykonywalnych i dokonać ich porównania).
6. C (Compile) – polecenie: **as --defsym FUNC_V2=1 -o lab_4.o lab_4.s**
7. LR (Link, Run)
8. Rezultat powinien być identyczny jak wcześniej.

9. C – polecenie: **as --defsym FUNC_V3=1 -o lab_4.o lab_4.s**
10. LR
11. Okazuje się, że rezultat jest taki sam dla wszystkich wersji funkcji **byte2hex**.
12. Dalsze prace prowadzimy wykorzystując dowolnie wybraną wersję funkcji **byte2hex** (poprzez zdefiniowanie odpowiedniego symbolu w trakcie kompilacji).
13. Usuwamy komentarze w następujących liniach programu:

```
#   disp_str_64 $stdout, $new_line, $1

#   movb varb, %al               # convert byte to hex string
#   movb $1, %cl                 # it's byte, so size = 1
#   mov $big_hex_str+2, %rdi     # address of most significant digit …
#   call num2hex
#   disp_str_64 $stdout, $big_hex_str, $4     # 0x + 2 digits
#   disp_str_64 $stdout, $new_line, $1
```

14. CLR
15. Powinien pojawić się dodatkowy napis „**0xBE**” – świadczy to prawidłowym działaniu funkcji **num2hex** w przypadku konwersji danej o wielkości jednego bajta. Teraz kolej na sprawdzenie danych o innych wielkościach.
16. Usuwamy pozostałe komentarze w programie głównym (aż do etykiety **theend**).
17. CLR
18. Wyświetlane są wartości danych o wielkości **2**, **4** i **8** bajtów – poprawność konwersji można sprawdzić przy użyciu kalkulatora dostępnego w systemie (po przełączeniu w tryb programisty). Warto zwrócić uwagę, że dla pewnych wartości liczb, stosując notację szesnastkową można uzyskać sensowne napisy.
19. We are the champions, my friends!