Marta Głowacka, 234999 Wrocław, 09.04.2018r.

WT-N-10

LABOLATORIUM ARCHITEKTURY KOMPUTERÓW

2. Stos, wczytywanie i zapisywanie do pliku

1. Treść ćwiczenia

Zakres i program ćwiczenia:

- Wczytywanie i zapis do pliku
- Zamiana systemu – bazy skojarzone
- Sklejanie bajtów – Little endian

Zrealizowane zadania:

- Wczytanie z pliku dwóch dużych liczb w reprezentacji czwórkowej
- Wpisać liczby do pamięci
- Wykonanie dodawania liczb z użyciem rejestrów 8B i flagi CF (użycie stosu)
- Zamiana wartości liczby na ciąg znaków w reprezentacji szesnastkowej
- Zapis wyniku do pliku

2. Przebieg ćwiczenia

Wstęp

Cały program został wysłany w pliku *.s, poniżej omówione zostały fragmenty kodu – pominięto trywialne inicjowanie zmiennych funkcji systemowych na samym początku, gdyż omówione zostało ono we wcześniejszych sprawozdaniach.

Początek

W sekcji **.data** inicjujemy:

Nazwy plików do wczytania dwóch liczb oraz zapisu wyniku.

```
file_in_1: .ascii "in_1.txt\0"    #w tej ścieżce znajduje się pierwsza liczba zapisana czwórkowo
```

```
file_in_2: .ascii "in_2.txt\0"    #-||- druga liczba zapisana czwórkowo

file_out: .ascii "result.txt\0" #wynik dodawania
```

Długości buforów.

```
max_input_len = 246

max_val_len = 64

max_out_len = 128
```

W sekcji **.bss** inicjujemy bufory, które posłużą nam do wczytania liczb w systemie czwórkowym (in_1 oraz in_2). Bufory zaczynające się od słów „result" trzymają w sobie liczby w postaci bajtowej (konkretnie sklejone wartości – konwencja little endian, o której więcej będzie w dalszej części sprawozdania). Tworzymy również bufor out, w którym zapiszemy wynik w ascii, by później wyświetlić go w terminalu lub zapisać do pliku.

Po etykiecie **_start** przechodzimy do zerowania buforów. Jest to konieczna procedura, gdyż nie chcemy, aby nasz bufor miał z przodu niezidentyfikowane wartości, jeśli go w pełni nie zapełnimy cyframi. Pożądane jest, aby z przodu były zera.

```
movq $max_val_len, %r8  #licznik dla pętli, która zeruje
movb $0, %al            #wstawiamy 0


petla_zerujaca:
dec %r8
mov %al, result_in_1(, %r8, 1)
mov %al, result_in_2(, %r8, 1)
mov %al, out(, %r8,1)


cmp $0, %r8             #sprawdzamy licznik
jg petla_zerujaca
```

Wczytywanie z pliku

Po tej procedurze możemy wczytać pierwszą dużą liczbę z pliku in_1.txt. Aby to zrobić, należy najpierw objaśnić, jak wygląda otwarcie pliku do odczytu i wczytanie zawartości do bufora. Procedura wygląda podobnie do odczytu danych z klawiatury, ale zamiast strumienia

STDIN podajemy identyfikator otwartego pliku. Jeśli plik podany przez nas nie istnieje, może on zostać utworzony. Należy podać jakie prawa dostępu powinien on mieć.

Wczytanie pierwszej liczby z pliku:

```
movq $SYSOPEN, %rax
movq $file_in_1, %rdi
movq $FREAD, %rsi
movq $0, %rdx
syscall
mov %rax, %r10          #identyfikator pliku będzie w r10



#Z pliku do bufora
movq $SYSREAD, %rax
movq %r10, %rdi         #podajemy id pliku, które jest w r10
movq $in_1, %rsi
movq $max_input_len, %rdx
syscall
movq %rax, %r8

#Zamknięcie pliku
movq $SYSCLOSE, %rax
movq %r10, %rdi
movq $0, %rsi
movq $0, %rdx
syscall
```

Little endian

Następnie należy zdekodować wartość w pliku. Będziemy to robić w następujący sposób. Wczytujemy jeden bajt, w którym zapisana jest liczba czwórkowa (taka liczba zapisana jest na maksymalnie 2 bitach, ponieważ największą wartością jest 11, czyli 3 w systemie czwórkowym). Będziemy sklejać po kolejne 2 bity ze sobą, aby efektywnie użyć pamięci, a później łatwo przekonwertować na system hexadecymalny za pomocą baz skojarzonych.

Operacja odbywa się poprzez pętlę realizującą przesuwanie bitów i dodawanie do nich tych poprzednich. Weźmy ciąg 3121. Najpierw odczytana zostanie wartość 01. Później 10, ale żeby skleić te wartości ze sobą, 10 należy przesunąć o 2 bity w lewo (1000). A następnie dodać do przesuniętych poprzednio bitów 01. W rezultacie otrzymujemy: 1001. W ten sposób skleiliśmy 21, czyli uzyskaliśmy 1001 – ciąg bitów, o który nam chodziło. Analogicznie postępujemy z dalszą częścią cyfr. Z tym, że przesunąć musimy później o 4 bitów, a następnie o 6, tak, aby zapełnić cały 8bitowy rejestr. Tak powstały bajt wczytujemy do bufora.

Poniżej przedstawiony jest kod wraz z komentarzami realizujący opisaną sytuację:

```
petla_dekodujaca_1:
dec %r8                 #obniżamy licznik liczby znaków, żeby liczył od 0
dec %r9                 #obniżamy licznik, który zapisuje od końca bufora


#Dekodowanie pierwszych 2 bitów
movb in_1(, %r8, 1), %al #wczytanie kodu ascii do rejestru al
sub $48, %al            #uzyskiwanie liczby z ascii


cmp $0, %r8             #jeśli pozostała liczba znaków do odczytania jest równa zero, to ciąg się skończył i możemy zapisać do bufora
jle zakoduj_do_bufora


#Dekodowanie kolejnych 2 bitów
dec %r8
mov in_1(, %r8, 1), %bl      #Pobranie kolejnej cyfry
sub $48, %bl            #Zdekodowanie z ascii na liczbę
shl $2, %bl             #Przesunięcie bitowe w lewo o 2 (na dwóch miejscach można zapisać
                    #nawiększą cyfrę - 3 w systemie czwórkowym), bl jest rejestrem
                    #pomocniczym, służącym do przesuwania aktualnie umieszczanej cyfry
```

```
add %bl, %al            #Dodanie tych właśnie przesuniętych dwóch bitów do poprzednich


cmp $0, %r8             #Porównianie, czy cyfry się nie skończyły i nie trzeba skończyć
jle zakoduj_do_bufora


#Kodowanie kolejnych 2 bitów
dec %r8
mov in_1(, %r8, 1), %bl      #Pobranie kolejnej cyfry
sub $48, %bl            #Zdekodowanie z ascii na liczbę
shl $4, %bl             #Przesunięcie bitowe w lewo o 4 (na dwóch miejscach można zapisać
                    #nawiększą cyfrę - 3 w systemie czwórkowym)
add %bl, %al            #Dodanie tych właśnie przesuniętych dwóch bitów do poprzednich


cmp $0, %r8             #Porównianie, czy cyfry się nie skończyły i nie trzeba skończyć
jle zakoduj_do_bufora


#Kodowanie kolejnych 2 bitów
dec %r8
mov in_1(, %r8, 1), %bl      #Pobranie kolejnej cyfry
sub $48, %bl            #Zdekodowanie z ascii na liczbę
shl $6, %bl             #Przesunięcie bitowe w lewo o 6 (na dwóch miejscach można zapisać
                    #nawiększą cyfrę - 3 w systemie czwórkowym)
add %bl, %al            #Dodanie tych właśnie przesuniętych dwóch bitów do poprzednich


cmp $0, %r8             #Porównianie, czy cyfry się nie skończyły i nie trzeba skończyć
```

```
jle zakoduj_do_bufora


zakoduj_do_bufora:
mov %al, result_in_1(, %r9, 1)    #Zapisanie     zdekodowanego
bajtu do bufora wynikowego


cmp $0, %r8             #Powrót  na  początek,  aby  dekodować
dalej cyfry, jeśli się nie skończyły
jg petla_dekodujaca_1
```

Wczytanie drugiej liczby z pliku i jej „rozkodowanie” odbywa się w ten sam sposób, kod zamieszczony z pliku źródłowym.

Dodanie liczb na rejestrach 8B

Dodawanie liczb realizowane będzie przez stos. Przed operacją czyścimy carry flag i wyrzucamy rejestr flagowy na stos. Dodajemy kolejne ciągi cyfr (binarne), zapisane w rejestrach al i bl (z przeniesieniem). Wynik zapisujemy do bufora resut_out.

```
clc                #Wyczyszczenie   flagi   przeniesienia   z
poprzedniej pozycji
pushfq                  #Włożenie rejestru z flagą na stos
mov $max_input_len, %r8  #Licznik pętli


petla_dodajaca:


mov result_in_1(, %r8, 8), %rax   #Zapis wartości z budora do
al (pierwsza liczba)-jej część końcowa
mov result_in_2(, %r8, 8), %rbx   #Zapis drugiej do bl
popfq                   #Pobranie     zawartości     rejestru
flagowego ze stosu, bo instrukcja
                   #cmp modyfikuje CF
adc %rbx, %rax          #Dodanie z propagacją i przeniesieniem
pushfq                  #Umieszczenie  rejestru  flagowego  na
stosie
mov %rax, result_out(, %r8, 8)    #Zapis wyniku do bufora
```

```
dec %r8

cmp $8, %r8             #Powrót na początek pętli, jeśli licznik != 0

jnz petla_dodajaca
```

Konwersja na szesnastkowy system

Używamy metody baz skojarzonych. Nie musimy sklejać bajtów, gdyż (1B = 8b, 4|8). Wyłuskujemy kolejne czwórki bitów (2^4 = 16).

Tzn., że jeśli w rejestrze mamy ciąg 1011 0100, to szesnastkowo zapisujemy liczbę B4.

W pętli konwersji mamy pętlę zagnieżdżoną, która wykona się 2 razy, dlatego, że mamy 2 czwórki bitów w bajcie. Pętla zewnętrzna wykona się tyle razy, ile bajtów mamy przekonwertować.

```
movq $max_val_len, %r8          #licznik bajtów z bufora result

movq $max_out_len, %r9          #licznik znaków 16stkowych z bufora out



petla_konwersji_na_0x:

movq $0, %rax           #odczyt kolejnych bitów - 4 i przesuniecia bitowe, aby pobrac z bufora result do rejestru rax 4 kolejne bity

dec %r8

movb result_out(, %r8, 1), %al

movq $2, %r10           #Zagniezdzona petla bedzie wykonywac sie 2 razy, dla ostatniej czworki bitow i przedostatniej z 8bitowego rejestru

zagniezdzona_petla:

movb %al, %bl           #w bl ciąg 8 bitowy

andb $0xf, %bl          #usuniecie wszystkich bitow poza 4rema najmniej znaczacymi, logiczne AND
```

W systemie 0x, gdy wartość zapisana na 4 bitach będzie powyżej 10, zakodować musimy literę, a jeśli poniżej, to cyfrę.

```
cmp $10, %bl

jl cyfra
```

```
#W przeciwnym razie kodujemy litere
add $39, %bl


cyfra:
add $48, %bl
```

Bity, które zostały przekonwertowane, przesuwamy tak, aby ich nie było w źródłowym rejestrze.

```
movb %bl, out(, %r9, 1)       #Zapis znaku ascii do bufora wyjsciowego


shr $4, %rax              #Przesuniecie bitowe dotychczasowej linii, tak aby pozbyć się zdekodowanych juz 4 bitów


dec %r9                   #Zmniejszenie licznika petli
dec %r10


cmp $0, %r10              #Skok z zagnieżdżonej pętli 298
jg zagniezdzona_petla


cmp $0, %r8
jg petla_konwersji_na_0x
```

Na koniec formalność – wpisanie ‘\n’ i powiększenie długości bufora.

```
movq $0, %rdi
movq $max_out_len, %r8
inc %r8
movb $0x0A, out(, %r8, 1)
inc %r8
```

W ostatnim kroku realizujemy wpisanie wartości wyniku do pliku result.txt.

```
#Zapis wyniku 0x do pliku
#Otworzenie pliku
```

```
movq $SYSOPEN, %rax
movq $file_out, %rdi
movq $FWRITE, %rsi
movq $0644, %rdx
syscall
movq %rax, %r9

#Zapis z bufora out do pliku
movq $SYSWRITE, %rax
movq %r9, %rdi
movq $out, %rsi
movq %r8, %rdx
syscall

#Zamkniecie pliku
movq $SYSCLOSE, %rax
movq %r9, %rdi
movq $0, %rsi
movq $0, %rdx
syscall
```

Zakańczamy program, dobrze znaną nam już instrukcją.

Wnioski

Sklejanie bitów jest realizowane po to, by możliwe było dodawanie z użyciem flagi CF.

Jeśli zamienialibyśmy np. na system ósemkowy (również skojarzony z bazą 2), musielibyśmy sklejać po 3 bajty (3 bajty = 24 bity, a 3 jest dzielnikiem 24), gdyż 8 nie jest wielokrotnością trójki (3 dlatego, że 2^3 = 8). W sytuacji w zaproponowanym w sprawozdaniu programie, nie było konieczności przeprowadzenia takiej operacji, gdyż zamienialiśmy na system szesnastkowy. (2^4 = 16, 4 jest wielokrotnością 8, a bajt składa się akurat z 8 bitów).

Źródła:

https://onedrive.live.com/?authkey=%21AIn%5F%5FYY0ZBMW0ZQ&cid=85C3A90A20A8892D&id=85C3A90A20A8892D%2110481&parId=85C3A90A20A8892D%2110477&o=OneUp

Professional Assembly Language, Richard Blum