Wrocław, 04-kwietnia-19r

Kamil Paczos, 218377
PT-N-12

prowadzący: Janusz Biernat

Laboratorium Architektury Komputerów

(1) *Proste konstrukcje programowe z użyciem instrukcji asemblera.*

1.1 Treść ćwiczenia

Zakres i program ćwiczenia:
- zapoznanie z działaniem instrukcji dzielenia *div* i debuggera gdb
- utworzenie programu zmieniającego wielkość liter we wprowadzonym ciągu znaków
- implementacja kodowania ciągu z wykorzystaniem szyfru cezara

Zrealizowane zadania:
- podstawowe wykorzystanie instrukcji *div*
- utworzenie programu przekodowania znaków z wykorzystaniem konstrukcji programowych i instrukcji systemowych (read, write, exit)
- wykonanie szkieletu programu implementującego szyfr cezara

1.2 Przebieg ćwiczenia

1.2.1 Program wykorzystujący instrukcję *div*

Instrukcja div, w odróżnieniu od bliźniaczej instrukcji idiv, wykorzystywana jest do wykonania dzielenia liczb bez znaku. Dzielna umieszczana jest w dwóch rejestrach: część młodsza w rejestrze eax, część starsza w edx. Przy pracy w trybie 64-bit są to odpowiednio rejestry rax i rdx. Dzielnik może być umieszczony w dowolnym z pozostałych rejestrów, a także pod wskazanym adresem w pamięci.

Zgodnie z przytoczonym wyżej działaniem instrukcji div zaimplementowany został szereg operacji arytmetycznych sprawdzających jej działanie.

W pierwszej z nich przeprowadzono dzielenie liczby 100 przez 25:

```
movl $0, %edx #wyzerowanie części starszej dzielnej
movl $100, %eax #część młodsza dzielnej
movl $25, %ecx #dzielnik
div %ecx #wykonanie operacji dzielenia
```

W wyniku wykonania programu pod kontrolą debuggera gdb stwierdzono, że w rejestrze eax znajduje się wynik operacji równy 4, a w rejestrze edx reszta z dzielenia równa 0.

W kolejnym kroku wykonania programu otrzymany powyżej wynik podzielony został przez 2:

```
movl $2, %ecx #dzielnik w rejestrze ecx
div %ecx #wykonanie operacji dzielenia
```

Zgodnie z przewidywaniami wynik operacji wyniósł 2, natomiast reszta 0. W kolejnym kroku w identyczny sposób wykonano dzielenie otrzymanego powyżej wyniku przez liczbę 4. Tym razem

wynik sprawdzony za pomocą debuggera wynosił 0, a reszta z dzielenia w rejestrze edx wynosiła 2, co jest poprawnym wynikiem operacji dzielenia liczby 2 przez liczbę 4.

W kolejnym etapie wykonania programu zostało wykonane dzielenie dużej liczby. W tym celu wykonano następujące operacje:

```
movl $1, %edx #1 w starszej części dzielnej
movl $0, %eax #część młodsza dzielnej równa 0
movl $10000, %ecx #dzielnik
div %ecx #wykonanie operacji dzielenia
```

W wyniku wykonania powyższych instrukcji pod kontrolą debuggera otrzymano wynik dzielenia równy 429496, natomiast reszta z dzielenia wyniosła 7296.

Program, jak wszystkie opisane w tym sprawozdaniu, kończy się wywołaniem systemowej funkcji

```
EXIT:
movl $EXIT, %eax
int $SYSCALL32 #wywołanie systemowe
```

1.2.2 Program zmieniający wielkość liter we wprowadzanym ciągu znaków

Wykonany program korzysta z funkcji systemowych READ, WRITE oraz konstrukcji programowych, takich jak pętle czy skoki warunkowe, w celu zmiany wszystkich znaków z małych na wielkie i odwrotnie. Pierwszym krokiem wykonania programu jest wczytanie danych z klawiatury wraz z zapamiętaniem ich rozmiaru:

```
 #CZYTAJ Z KLAWIATURY
 movl $BUF_SIZE, %edx
 movl $BUF, %ecx
 movl $STDIN, %ebx
 movl $READ, %eax
 int $SYSCALL32 #wywołanie systemowe
 movl %eax, TEXT_SIZE #zapamiętanie rozmiaru danych w zmiennej TEXT_SIZE
```

Następnym etapem było stworzenie pętli iterującej po wszystkich znakach tekstu. Przed jej wykonaniem rejestr edi został zainicjowany wartością 0. Po każdym wykonaniu pętli jest on inkrementowany, a jego aktualna wartość porównywana z wartością zmiennej TEXT_SIZE. Jeżeli obie wartości są równe, następuje skok do miejsca w kodzie odpowiedzialnego za wypisanie danych na ekran i zakończenie programu. W przeciwnym razie wykonywany jest skok na początek pętli. Całość zrealizowana została w następujący sposób:

```
#WARUNEK WYJSCIA Z PETLI
inc %edi #inkrementacja licznika
cmp %edi, TEXT_SIZE #porównanie licznika z długością tekstu
je write #jeżeli ostatni znak to zakończ pętlę
jmp loop #wróć na początek pętli
```

Sama zmiana wielkości znaku zaimplementowana została jako operacja xor wykonywana na znaku oraz masce, której zapis bitowy to 100000, a wartość dziesiętna to 32. Każdy znak przenoszony jest ze zmiennej BUF do rejestru al. Po sprawdzeniu, czy znak nie jest znakiem końca linii, następuje wykonanie właściwej operacji xor z maską przeniesioną do rejestru cl. Po wykonaniu operacji znak w zmiennej BUF zastępowany jest nowym znakiem. Kod wygląda następująco:

```
movb BUF(%edi), %al #aktualny znak
movb $'\n', %cl #znak końca linii
cmp %al, %cl #nie zmieniamy końca linii
je write
movb MASK, %cl #przeniesienie maski
xor %al, %cl #zmiana wielkosci litery
movb %cl, BUF(%edi) #podmiana znaku w zmiennej BUF
```

Po wykonaniu zmiany tekst wypisywany jest na ekran. Sposób wypisania analogiczny jest do fragmentu kodu odpowiedzialnego za wczytanie tekstu, tj. wywoływana jest systemowa funkcja WRITE z odpowiednimi parametrami.

Po wypisaniu tekstu następuje obsługa zewnętrznej pętli programu. Została ona utworzone w celu kilkukrotnej zmiany wielkości znaków w trakcie jednego wykonania programu. Ilość powtórzeń zdefiniowana jest w zmiennej LOOPC. Po każdym wypisaniu tekstu na ekran zmienna ta jest dekrementowana. Jeżeli jej wartość osiągnie 0, następuje wyjście z programu. Jeżeli jest różna od 0, to następuje powrót do linii poprzedzającej pętlę zmiany wielkości znaków.

1.2.3 Program realizujący szyfrowanie szyfrem cezara

Ze względu na ograniczenia czasowe w trakcie trwania laboratorium udało się zrealizować jedynie fragment programu implementującego szyfr cezara. Jego struktura podobna jest do opisanego powyżej programu zmieniającego rozmiar znaku. Zaimplementowana została więc pętla, która wykonywana jest tyle razy, ile znaków ma wprowadzony ciąg.

Różnice pojawiają się w momencie wykonywania operacji na znakach. Zamiast operacji xor, wykonywana jest operacja or na znaku i masce o wartości równej 32. Powoduje to zmianę każdego wielkiego znaku w ciągu na jego mały odpowiednik. Małe znaki nie są zmieniane. Następnie wykonywane jest przesunięcie znaku poprzez wykonanie operacji add z maską o wartości 3 (binarnie 11). Powoduje to zmianę kodu znaku o 3, tak więc przykładowo litera A staje się literą D, B staje się E, itd.

```
movb MASK, %cl
orb %cl, %al #zamień na mala literę
movb SHIFT, %cl
add %al, %cl #szyfruj
```

Po osiągnięciu końca linii program wypisuje tekst i kończy się.

1.3 Wnioski

Zrealizowane programy wykonywały się bezbłędnie. Program realizujący szyfr cezara powinien zostać uzupełniony o obsługę znaków z końca zakresu, np. litera „z” powinna zostać zmieniona na literę „c”, a nie tak jak obecnie na znak „}”.

(2) *Tworzenie i używanie funkcji, funkcje rekurencyjne*

2.1 Treść ćwiczenia

Zakres i program ćwiczenia:
- wykonanie programu pokazującego równicę pomiędzy działaniem instrukcji idiv oraz div, mul i imul.
- wykonanie w trzech wariantach (z przekazaniem parametrów przez rejestr, przez stos oraz z wykorzystaniem funkcji rekurencyjnej) programów realizujących:
a) obliczanie elementu ciągu Fibonacciego
b) obliczanie największego wspólnego dzielnika (NWD)
c) obliczanie liczby kombinacji k-elementowych ze zbioru n-elementowego.

Zrealizowane zadania:
- program pokazujący różnicę pomiędzy instrukcjami div/idiv oraz mul/imul
- wykonie programów w trzech wersjach dla zadania obliczenia elementu ciągu Fibonacciego i NWD
- wykonanie programu obliczenia liczby kombinacji w wariancie z przekazaniem wartości przez rejestr.

2.2 Przebieg ćwiczenia
2.2.1 Program wykonujący dzielenie i mnożenie

Wykonany program jest modyfikacją programu z laboratorium pierwszego. Jako pierwsze ponownie zostało wykonane działanie dzielenia liczby 100 przez liczbę 4. Następnie wynik został pomnożony przez liczbę 4. Aby to wykonać, do rejestru ebx została wprowadzona wartość 4. Następnie wykonane zostało mnożenie komendą:

```
mul %ebx
```

Wynik (16) został zapisany w rejestrze eax. Z uwagi na ten fakt można było bez przenoszenia wykonać na nim kolejne operacje. Do rejestru ecx wprowadzona została wartość 2, a następnie wykonane operacje:

```
idiv %ecx
div %ecx
```

Jako, że wszystkie liczby były dodatnie, nie było żadnej różnicy pomiędzy użyciem instrukcji div i idiv. Wyniki kolejno wynosiły 8 i 4.

W kolejnym kroku do rejestru bx załadowano liczbę -4. Następnie wykonano instrukcję:

```
movsx %bx, %ecx
```

Służy ona do przeniesienia zawartości mniejszego rejestru do większego, wraz z rozszerzeniem bitu znaku. Następnie wykonano mnożenie ze znakiem instrukcją:

```
imul %ecx
```

Wynik w rejestrze eax to -16, co jest poprawne.

Kolejna operacja sprawdzała różnicę pomiędzy instrukcjami movsx i movzx. Wykonano dwa podobne fragmenty kodu:

```
mov $-2, %bx #załadowanie liczby -2 do rejestru bx
movsx %bx, %ecx
idiv %ecx #dzielenie
```

oraz:

```
mov $-2, %bx #załadowanie liczby -2 do rejestru bx
movzx %bx, %ecx
idiv %ecx #dzielenie
```

Argumentami dzielenia były więc liczby -16 i -2. Tylko operacja poprzedzona instrukcją movsx dała poprawny wynik, tj. 8. Wynika to z tego, że instrukcja movsx służy do przenoszenia pomiędzy rejestrami danych ze znakiem i rozszerzania ich bitem „1”. Instrukcja movzx służy do przenoszenia danych bez znaku, więc bitem rozszerzenia jest „0”.

2.2.2 Obliczanie elementu ciągu Fibonacciego

Przy realizacji programu przyjęto założenie, że w rejestrze eax znajduje się n-ty element, w rejestrze ebx element n-1, natomiast w ecx – n-2. Pierwszy wykonany wariant opierał się na pętli wywołującej określoną ilość razy funkcję liczącą element ciągu. Konstrukcja pętli wygląda następująco:

```
movl $3, %edi #numer szukanego elementu to wartość rejestru edi + 2, bo dwa
pierwsze są ustalone
loop:
call fibo #wywołanie funkcji liczącej
dec %edi #zmniejszenie licznika pętli
jnz loop #jeżeli licznik nie równy zero – kolejne wykonanie pętli
```

Argumenty przekazane do funkcji cały czas znajdują się w rejestrach. Funkcja licząca wygląda następująco:

```
.type fibo, @function #deklaracja, że fibo jest funkcją
fibo:
        movl %eax, %ecx # zapamiętanie elementu n
        add %ebx, %eax #wynik w eax jest nowym elementem
        movl %ecx, %ebx poprzedni element n jest teraz elementem n-1
        ret #powrót z funkcji
```

Po zakończeniu pętli element n-ty ciągu znajduje się w rejestrze eax.

W drugim wariancie liczby do funkcji przekazywane były przez stos. Konstrukcja pętli została więc uzupełniona o ich przekazanie:

```
pushq %rax
pushq %rbx
```

Sama funkcja licząca poza operacją pobierania danych ze stosu musiała zostać uzupełniona o zapamiętanie położenia wskaźnika stosu. Zapamiętanie wartości wskaźnika stosu jest szczególnie istotne w momencie, w którym funkcja przechowuje na stosie zmienne lokalne. Konstrukcja funkcji wygląda następująco:

```
.type stackFibo, @function
stackFibo:
        pushq %rbp #zapamiętanie wartości rejestru rbp
        movq %rsp, %rbp #zapamiętanie wskaźnika stosu

        movl 16(%rbp), %edx #pobranie wartości ze stosu
        movl 24(%rbp), %ecx #pobranie wartości ze stosu
        add %ecx, %edx #obliczenie elementu
        movl 24(%rbp), %ebx #pobranie wartości ze stosu
        movl %edx, %eax #wartość elementu n ciągu w rejestrze eax

        movq %rbp, %rsp #przywrócenie wskaźnika stosu
        popq %rbp #przywrócenie wartości rejestru rbp
        ret #powrót
```

Ostatnim wariantem było użycie funkcji rekurencyjnej czyli takiej, która opiera się na wywoływaniu samej siebie do momentu osiągnięcia pożądanego efektu. W tym przypadku nie było konieczne konstruowanie pętli. Po wprowadzeniu dwóch pierwszych wartości do rejestrów eax i ebx oraz numeru elementu (3) do edi wywołana została funkcja o następującej konstrukcji:

```
.type recFibo, @function
recFibo:
        movl %eax, %ecx #zapamiętaj aktualny n element
        add %ebx, %eax #oblicz nowy element n
        movl %ecx, %ebx #przesuń elementy, n → n-1
        dec %edi #zmniejsz licznik
        jz nc #jeżeli wykonano pożądaną ilość razy zakończ drzewo wywołań
        call recFibo #wywołaj samą siebie, by policzyć kolejny element
nc:     ret #powrót
```

Wszystkie wykonane warianty działają poprawnie, co sprawdzono pod kontrolą debuggera.

2.2.3 Program liczący największy wspólny dzielnik (NWD)

W programie liczby a i b przechowywano odpowiednio w rejestrach ebx i ecx. Ma to związek ze specyfikacją instrukcji div, gdzie wynik operacji znajduje się w rejestrze eax.

W pierwszym wariancie parametry funkcji przekazywane były przez rejestry. Główna pętla programu wywoływała funkcję liczącą do momentu, aż reszta z dzielenia wyniosła 0:

```
loop:
call nwd #wywołanie funkcji
cmp $0, %edx #sprawdzenie, czy reszta z dzielenia równa 0
jne loop # jeżeli nie równa, powtórz wykonanie pętli
```

Sama funkcja licząca opierała na podzieleniu liczby a przez liczbę b. Jeżeli reszta z dzielenia była różna od 0, liczba b stawała się liczbą a, natomiast liczba b zastępowana była resztą z wykonanego dzielenia. Struktura funkcji była następująca:

```
.type nwd, @function
nwd:
        movl %ebx, %eax
        movl $0, %edx #przygotowanie do dzielenia
        div %ecx #podziel a przez b
        cmp $0, %edx #sprawdzenie, czy reszta równa 0
        je retf #obliczone
        movl %ecx, %ebx #przypisujemy liczbie a liczbę b
        movl %edx, %ecx #resztę przypisujemy liczbie b
retf:
        ret #powrót
```

Drugim wykonanym wariantem była funkcja otrzymująca parametry przez stos. Tak jak w poprzednim programie, pętla została uzupełniona o przekazanie argumentów na stos. Funkcja natomiast również została uzupełniona o zapamiętanie wskaźnika stosu. Sama część licząca funkcji nie różni się niczym od opisanej powyżej, poza pobraniem argumentów ze stosu.

W wariancie rekurencyjnym ponownie pętla zastąpiona została wywołaniem funkcji rekurencyjnej. Jej wywołanie pozwalało obliczyć NWD w następujący sposób:

```
.type recNWD, @function
recNWD:
        movl %ebx, %eax #przygotowanie dzielenia
        movl $0, %edx
        div %ecx #dzielenie
        cmp $0, %edx
        jz nc #jeżeli reszta równa 0, to znaleziono NWD
        movl %ecx, %ebx #zamiana elementów a i b
        movl %edx, %ecx #reszta z dzielenia nowym elementem b
        call recNWD # wywołanie rekurencyjne
nc:
        ret #powrót
```

2.2.4 Obliczanie liczby kombinacji „n po k”

Do obliczenia liczby kombinacji konieczne było zaimplementowanie funkcji liczącej permutacje. Jej argument przekazywany był w rejestrze eax i w tym samym rejestrze zwracana była jego permutacja. Funkcja zaimplementowana została w sposób następujący:

```
.type perm, @function
perm:
        movl %eax, %edi #element jako licznik
loop:
        dec %edi #zmniejszenie licznika pętli
        jz endf #jeżeli obliczono, to wyjście z funkcji
        mul %edi #mnożenie aktualnego wyniku przez zmniejszony licznik
        jmp loop
endf:
        ret #wyjście
```

Z wykorzystaniem tej funkcji możliwe było zaimplementowanie mechanizmu obliczania liczby kombinacji w sposób następujący:

```
#liczenie  permutacji  k
movl k, %eax
call perm
movl %eax, %ecx #permutacja k w rejestrze ecx
```

```
movl n, %eax #liczenie permutacji n
call perm
pushq %rax #permutacja n przeniesiona  na stos
movl n, %eax #obliczenie permutacji (n-k)
sub k, %eax #odejmowanie k od n
call perm #permutacja n-k w eax
mul %ecx #mianownik k! * (n-k)! w eax
movl %eax, %ebx #zapamiętanie mianownika
popq %rax #licznik n! w eax
movl $0, %edx
div %ebx #obliczenie ułamka, liczba kombinacji w rejestrze eax
```

2.2.5 Wnioski

Zrealizowane programy w procesie testowanie z wykorzystaniem debuggera gdb działały poprawnie. W programie liczenia liczby kombinacji cały proces mógłby zostać wyodrębniony z kodu głównego programu do oddzielnej funkcji, której argumentami byłyby n i k.

Literatura

[1]"Sign Extending with MOVSX and MOVZX", C-jump.com. [Online]. Available: http://www.c-jump.com/CIS77/ASM/DataTypes/T77_0270_sext_example_movsx.htm. [Accessed: 05- Apr- 2019].

[2]"Asembler x86/Instrukcje/Arytmetyczne - Wikibooks, biblioteka wolnych podręczników", Pl.wikibooks.org. [Online]. Available: https://pl.wikibooks.org/wiki/Asembler_x86/Instrukcje/Arytmetyczne. [Accessed: 05- Apr- 2019].

[3]R. Blum, Professional assembly language. Indianapolis, IN: Wiley, 2005.