lab_5 - Instrukcja do ćwiczenia

Teoria (params_64.pdf):

Przekazywanie parametrów:

Zadanie polega na użyciu w kodzie asemblerowym funkcji znajdujących się w bibliotece standardowej LIBC. Wymaga to zrozumienia zasad dotyczących przekazywania parametrów do wywoływanych funkcji oraz zwracania przez te funkcje ewentualnych rezultatów. Szczegółowe informacje na ten temat można znaleźć w dokumencie „System V Application Binary Interface - AMD64 Architecture Processor Supplement" (**abi.pdf**).

Aby uprościć proces tworzenia kodu wykonywalnego, w programie użyto jako punktu startowego etykiety **main** (a nie **_start**) – pozwala to na wykorzystanie do kompilacji i linkowania programu **gcc** oraz zwalnia nas od konieczności podawania dokładnych nazw bibliotek i ich lokalizacji. Dodatkową zaletą takiego podejścia jest też możliwość prostego i łatwego dostępu do damych przekazywanych standardowo do funkcji **main** (**argc**, **argv** i **env**).

Do przekazywania parametrów będących liczbami całkowitymi (dane liczbowe i adresy) wykorzystywane są następujące rejestry procesora:

%rdi – parametr nr 1
%rsi – parametr nr 2
%rdx – parametr nr 3
%rcx – parametr nr 4
%r8 – parametr nr 5
%r9 – parametr nr 6

Jeśli funkcja wymaga większej liczby parametrów, to muszą one być przekazane poprzez stos.

Funkcja nie może zmienić wartości rejestrów **%rbp**, **%rbx**, **%r12**, **%r13**, **%r14**, **%15** oraz **%rsp** – co oznacza, że jeśli chcemy ich użyć, to konieczne jest zapisanie ich na stosie (np. na początku kodu) a później odtworzenie ich oryginalnej zawartości (np. na końcu kodu). Rejestry wykorzystywane do przekazywania parametrów mogą być modyfikowane (ich wyjściowa zawartość może być różna od wejściowej). Funkcje o zmiennej liczbie parametrów (np. **printf**, **scanf**, itp.) wymagają dodatkowego parametru: rejestr **%al** musi zawierać liczbę wykorzystywanych rejestrów wektorowych (**xmm**, **ymm** – są konieczne do przekazywania liczb zmiennoprzecinkowych (**float**, **double**) jako parametrów) – nawet jeśli nie używamy żadnych rejestrów wektorowych, to musimy do rejestru **%al** wstawić liczbę **0**.

Zwracany rezultat musi znaleźć się w rejestrze **%rax**.

Przykład:

język C – `printf(„Value=%d\n", value);`

Aby użyć funkcji **printf** w języku asemblera konieczne jest zdefiniowanie odpowiednich danych:

```
value:    .long 6
fmt:      .asciz „Value=%d\n”
```

a następnie przekazanie w rejestrach parametrów dla funkcji **printf** i jej wywołanie:

```
mov value, %esi # printf(char *fmt,long num) – 2nd argument to %rsi/%esi
mov $fmt, %rdi # printf(char *fmt,long num) – 1st argument to %rdi
xor %rax, %rax # printf( char *fmt,long num) - number of vector regs to %al
call printf
```

Funkcja **printf** zwraca liczbę danych wyświetlonych zgodnie z zawartością łańcucha formatującego (w powyższym przykładzie będzie to **1**), ale zwykle można ten rezultat pominąć.

Największy wspólny dzielnik (NWD) – Greatest Common Divisor (GCD):

Typowym sposobem na wyliczenie NWD/GCD dla dwóch liczb dodatnich jest zastosowanie algorytmu Euklidesa. Implementacja tego algorytmu w języku C może wyglądać następująco:

```
unsigned int NWD( unsigned int a, unsigned int b )
{
     while( a != b )
     {
          if( a > b )
               a = a - b;
          else
               b = b - a;
     }
     return a;
}
```

Implementacja w asemblerze jest w zasadzie odpowiednikiem w skali 1:1 – główna różnica dotyczy odwołań do argumentów funkcji znajdujących się w rejestrach (a w %rdi, b w %rsi):

```
#-----------------------------------------------------------------
# gcd - computes greatest common divisor
#     Arguments:  %rdi - first numer
#                 %rsi - second numer
#     Returns:    %rax - gcd value
#-----------------------------------------------------------------
      .type gcd, @function
gcd:
      cmp %rdi, %rsi    # (a==b)?
      jz computed       # yes
```

```
      jb b_below_a            # if(b < a) goto b_below_a
      sub %rdi, %rsi          # else b=b-a
      jmp gcd
b_below_a:
      sub %rsi, %rdi          # a=a-b
      jmp gcd
computed:
      mov %rdi, %rax          # result (a==b)
      ret
```

Parametry funkcji main:

Zmiana nazwy punktu startowego z **_start** na **main** i wykorzystanie programu **gcc** w procesie kompilacji i linkowania skutkuje tym, że instrukcje programu stanowią faktycznie zawartość funkcji main (głównej funkcji programu). Funkcja ta jest wywoływane instrukcją **CALL**, co pozwala na zakończenie działania poprzez użycie instrukcji **RET**. Bardziej przydatną właściwością funkcji **main** jest możliwość prostego dostępu do danych przekazywanych z zewnątrz do programu. Deklaracja funkcji **main** w języku **C** może mieć następującą postać:

```
int/void main( int argc, char *argv[], char *env[] );

int/void main( int argc, char **argv, char **env );
```

Ponieważ funkcja **main** jest taką samą funkcją jak np. **printf** (w sensie przekazywania parametrów), to parametry funkcji **main** można znaleźć w odpowiednich rejestrach:

- **argc** – w **%rdi** (bo to pierwszy argument)
- **argv** – w **%rsi** (bo to drugi argument)
- **env** – w **%rdx** (bo to trzeci argument)

Celowe jest przeniesienie tych parametrów do zmiennych, bo te same rejestry będą używane przy wywołaniach innych funkcji wewnątrz funkcji **main** – najlepiej jest to zrobić na samym początku programu.

Liczba argumentów programu (**argc**) jest większa lub równa **1** (**1**, jeśli nie podano żadnych argumentów – wtedy **argv[0**] jest nazwą programu (polecenia użytego do uruchomienia programu) – może być wyświetlona tak, jak każda inna liczba całkowita. Argumenty programu (**argv**) są wskaźnikami do łańcuchów znaków (zgodnymi z językiem **C**), więc do ich wyświetlenia konieczne jest użycie formatu **%s**, a samo wyświetlanie można zrealizować w pętli typu **for** (liczba elementów jest znana z góry). Przykładowy kod ma następującą postać:

```
fmt_str:    .string "%s\n"

      mov argv, %rbp          # %rbp = argv;
next_argv:
      mov (%rbp), %rsi        # printf( fmt, str ) – 2nd argument to %rsi;
      mov $fmt_str, %rdi      # printf( fmt, str ) – 1st argument to %rdi;
      xor %rax, %rax          # printf - number of vector regs to %al
      call printf
      add $8, %rbp            # next argv
      decl argc               # argc;
      jnz next_argv
```

Rejestr **%rbp** wykorzystany jest do adresowania kolejnych elementów tablicy **argv** (zmienne **argv** i **argc** powinny zostać zainicjalizowane na początku programu na podstawie parametrów przesłanych do funkcji **main**). Tablica **argv** jest tablicą wskaźników, więc do dostępu do samego wskaźnika, konieczne jest użycie adresowania pośredniego: `mov (%rbp), %rsi`. Po tej operacji **%rsi** zawiera wskaźnik do łańcuch znaków (zgodnego z **C**), będącego kolejnym argumentem programu (**argv[0]**, **argv[1]**, ..., itd.). Licznikiem pętli jest zmienna **argc**, co oznacza, że jej oryginalna wartość jest niszczona – gdyby była ona jeszcze potrzebna, konieczne jest użycie jakiegoś rejestru jako licznika pętli, albo innej zmiennej, będącej kopią **argc**. Każdy element tablicy **argv** jest wskaźnikiem (adresem) i zajmuje w pamięci **8** bajtów, więc o tyle modyfikowana jest zawartość rejestru **%rbp** przy przejściu do kolejnego elementu tablicy.

Ponieważ liczba elementów tablicy **env** nie jest znana z góry, konieczne jest zastosowanie pętli typu **while**. Warunkiem stopu jest znalezienie wskaźnika o wartości **0** (**NULL**). Przykładowy kod ma następująca postać:

```
      mov env, %rbp           # %rbp = env;
next_env:
      cmp $0,(%rbp)           # while( env[i] != NULL )
      jz no_more_env
      mov (%rbp), %rsi        # printf( fmt, str ) – 2nd argument to %rsi;
      mov $fmt_str, %rdi      # printf( fmt, str ) - 1st argument to %rdi;
      xor %rax, %rax          # printf - number of vector registers to %al
      call printf
      add $8, %rbp            # next env
      jmp next_env
no_more_env:
```

Konwersja łańcuchów znaków na liczby:

Ponieważ argumenty programu są dostępne w formie łańcuchów znaków, tam gdzie niezbędne są ich wartości liczbowe, konieczne jest dokonanie odpowiedniej konwersji. Można do tego celu wykorzystać np. funkcję atoi – jej deklaracja jest następująca:

```
int atoi( const char *string );
```

Funkcja zwraca wartość liczby zapisanej w postaci ciągu znaków lub 0, jeśli znaki nie dają się zinterpretować jako liczba. Przykładowe zastosowanie może wyglądać następująco:

```
num_str:  .string   "15"
num_val:  .long     0

mov $num_str, %rdi  # 1st and only argument
call atoi           # call function
mov %eax, num_val   # store converted value in variable
```

Praktyka (lab_5.s):

Działania:

1. Testujemy użycie funkcji **printf** do wyświetlenia wartości liczby **val_1** i funkcji **exit** do zakończenia działania programu – inne instrukcje są zakomentowane.
2. CL (Compile&Link) – polecenie: **gcc –no-pie –o lab_5 lab_5.s**
3. R (Run) – polecenie: **./lab_5**
4. Pojawia się napis: „Value = 6” i program kończy swoje działanie, co świadczy o prawidłowym przebiegu przekazywania parametrów i wywoływania funkcji z biblioteki C.
5. Komentujemy linie dotyczące wywołania funkcji **exit**:

```
#  xor %rdi, %rdi          # exit( code ) - first argument to %rdi
#  call exit
```

6. CLR
7. Efekt jest taki sam jak wcześniej – funkcja **main** kończy swoje działanie poprzez wykonanie instrukcji **RET**.
8. Usuwamy komentarze w kolejnych liniach programu – celem jest przetestowanie wyświetlania dwóch liczb (**var_a**, **var_b**) przy pomocy funkcji **printf**, przetestowania działania funkcji **gcd** (argumentami dla **gcd** są właśnie wyświetlone liczby) oraz przetestowanie wyświetlania trzech liczb (argumentów funkcji **gcd** oraz jej rezultatu):

```
#  mov var_b, %rdx   # printf( fmt, num1, num2 ) - third argument to %rdx;
#  mov var_a, %rsi   # printf( fmt, num1, num2 ) - second argument to %rsi;
#  mov $fmt_4, %rdi  # printf( fmt, num1, num2 ) - first argument to %rdi;
#  xor %rax, %rax    # printf - number of vector registers to %al
#  call printf
#  mov var_a, %edi   # nwd( long num1, long num2 ) - 1st argument to %rdi
#  mov var_b, %esi   # nwd( long num1, long num2 ) - 2nd argument to %rsi
#  call gcd
#  mov %rax, %rcx    # printf(fmt,num1,num2,result) - 4th argument to %rcx
#  mov var_b, %rdx   # printf(fmt,num1,num2,result) - 3rd argument to %rdx
#  mov var_a, %rsi   # printf(fmt,num1,num2,result) - 2nd argument to %rsi
#  mov $fmt_2, %rdi  # printf(fmt,num1,num2,result) - 1st argument to %rdi
#  mov $0, %al       # printf - number of vector registers to %al
#  call printf
```

9. CLR
10. Liczby wyświetlane są prawidłowo – wartość **gcd(3084, 1424)** jest równa **4**.
11. Usuwamy kolejne komentarze – tym razem chodzi o przetestowanie działania funkcji **scanf**, której chcemy użyć do wprowadzania danych przez użytkownika. Funkcja **scanf** ma wczytać dwie liczby, więc przekazujemy do niej trzy parametry: adres łańcucha formatującego (podobnie jak dla funkcji **printf**) oraz adresy zmiennych, w których zostaną umieszczone wczytane dane: **$var_a** i **$var_b**. Dodatkowym parametrem zawartym w rejestrze **%al** jest liczba wykorzystanych rejestrów wektorowych (**scanf** jest funkcją o zmiennej liczbie argumentów) – testy wskazują, że parametr ten może być pominięty, bo tak naprawdę wszystkie argumenty funkcji **scanf** są adresami. Funkcja **scanf** zwraca liczbę danych jakie udało się wczytać – zapamiętujemy zwróconą liczbę w zmiennej **ok_num**. Aby poprawić wygląd ekranu po użyciu funkcji **scanf**, wyświetlamy

znak **\n** (w ten sposób wymusimy wyświetlanie kolejnych informacji od początku kolejnej linii ekranu niezależnie od tego jak przebiegało wprowadzanie danych). Poprawne wprowadzenie dwóch liczb będzie skutkować przejściem do etykiety **again** – w ten sposób można będzie wielokrotnie wprowadzać dane i wyznaczać dla nich wartość funkcji **gcd** (łącznie z wyświetlaniem samych danych i rezultatów, czyli tego co już w programie działa poprawnie). Jeśli dane zostaną wprowadzone nieprawidłowo (nie podano żadnej albo podano tylko jedną liczbę), to nastąpi wyjście z pętli i przejście do kolejnych instrukcji (w aktualnym stanie do instrukcji **RET** i zakończenia programu).

```
again:
#     mov $fmt_3, %rdi
#     mov $var_a, %rsi
#     mov $var_b, %rdx
#     mov $0, %al
#     call scanf
#     mov %eax, ok_num

#     mov $fmt_lf, %rdi       # printf(fmt) – 1st argument to %rdi;
#     xor %rax, %rax          # printf - number of vector regs to %al
#     call printf

#     cmp $2, ok_num
#     jnz no_more_numbers


#     jmp again
```

12. CLR
13. Program nie działa prawidłowo – po dojściu do wywołania funkcji **scanf** następuje naruszenie ochrony pamięci. Przyczyną jest wrażliwość (wręcz nadwrażliwość bo funkcja **printf** w identycznych warunkach działa poprawnie) funkcji **scanf** na kwestię wyrównania stosu do wielokrotności 16 bajtów (po wywołaniu funkcji **main** na dotychczas wyrównanym stosie znajduje się tylko adres powrotu (**8** bajtów)). Wyrównujemy stos poprzez umieszczenie na nim dodatkowej danej o wielkości **8** bajtów – zawartości rejestru **%rbp**.
14. Wyszukujemy w kodzie (na początku i na końcu) poniższe linie i usuwamy w nich znaki komentarza:

```
main:
#     push %rbp


#     pop %rbp
```

15. CLR
16. Teraz program działa prawidłowo – możemy wprowadzać liczby i testować działanie funkcji **gcd**. Aby wyjść z pętli wczytywania danych należy wprowadzić znaki, które nie dadzą się zinterpretować jako liczba – np. literę/litery.
17. Teraz zajmiemy się wykorzystaniem danych przekazanych do funkcji **main** z zewnątrz. Ponieważ argumenty funkcji **main** znajdują się w rejestrach, które będą też używane

przy wywołaniach innych funkcji (np. **printf**, **scanf**, itp.), należy przenieść je z rejestrów do zmiennych na początku funkcji **main** (zanim zostaną użyte do innych celów). W tym celu musimy usunąć komentarze w następujących liniach programu:

```
#      mov %edi, argc
#      mov %edi, argc_tmp
#      mov %rsi, argv
#      mov %rdx, env
```

Aby wyświetlić zawartość zmiennej **argc** niezbędne jest odkomentowanie następujących instrukcji:

```
no_more_numbers:
#      mov argc, %rsi          # printf(fmt,num) – 2nd argument to %rsi;
#      mov $fmt_argc, %rdi     # printf(fmt,num) - 1st argument to %rdi;
#      xor %rax, %rax          # printf - number of vector regs to %al
#      call printf
```

18. CLR
19. W normalnych warunkach wartość **argc** jest równa **1** – aby sprawdzić czy wszystko działa prawidłowo, należy kilkukrotnie uruchomić program, podając w linii poleceń dodatkowe argumenty - np. w poniższy sposób:

    `./lab_5 Ala ma kota`

20. Dla powyższego polecenia powinien pojawić się napis: „**Argc=4**” - jeśli program reaguje na liczbę podawanych argumentów, to możemy przejść do ich wyświetlenia. Usuwamy znaki komentarza w następujących liniach:

```
#      mov argv, %rbp          # %rbp = argv;

next_argv:

#      mov (%rbp), %rdx    # printf(fmt,num,str)- 3rd argument to %rdx;
#      mov argc, %esi
#      sub argc_tmp, %esi  # printf(fmt,num,str)- 2nd argument to %rsi;
#      mov $fmt_argv, %rdi # printf(fmt,num,str)- 1st argument to %rdi;
#      xor %rax, %rax      # printf - number of vector registers to %al
#      call printf
#      add $8, %rbp        # next argv
#      decl argc_tmp       # argc_tmp--;
#      jnz next_argv
```

Jako licznik pętli wykorzystujemy zmienną **argc_tmp** (kopia **argc**) – kolejne wartości licznika dla wcześniejszego przykładu to: **4**, **3**, **2**, **1**. Aby wyświetlić numery argumentów w sposób właściwy (**0**, **1**, **2**, **3**) wykorzystujemy wyrażenie **argc – argc_tmp** (**4-4-0**, **4-3=1**, **4-2=2**, **4-1=3**). Tablica **argv** zawiera wskaźniki/adresy – każdy o rozmiarze **8** bajtów, więc o taką wartość jest modyfikowany rejestr **%rbp** (użyty jako wskaźnik do kolejnych elementów) przy przejściu do kolejnego elementu tablicy **argv**.

21. CLR
22. Sprawdzamy działanie programu uruchamiając go kilka razy jednocześnie zmieniając podając postać argumentów i ich liczbę.

23. Ostatnim etapem jest wyświetlenie wartości zmiennych środowiskowych (zawartości tablicy **env**). Dokonujemy tego przez usunięcie znaków komentarza w następujących liniach:

```
#       mov env, %rbp       # %rbp = env;

next_env:

#       cmp $0,(%rbp)       # while( env[i] != NULL )
#       jz no_more_env
#       mov (%rbp), %rdx    # printf(fmt,num,str) - 3rd argument to %rdx;
#       mov argc_tmp,%esi # printf(fmt,num,str) - 2nd argument to %rsi;
#       mov $fmt_env,%rdi # printf(fmt,num,str) - 1st argument to %rdi;
#       xor %rax, %rax      # printf - number of vector registers to %al
#       call printf

#       add $8, %rbp        # next env
#       incl argc_tmp       # argc_tmp++;
#       jmp next_env

no_more_env:
```

Ponieważ liczba elementów tablicy **env** nie jest znana (w szczególności może być równa **0**), to konieczne jest zastosowanie pętli **while** (warunek sprawdzany na początku) – warunkiem stopu jest element o wartości **0** (wskaźnik **NULL**). Do numerowania elementów wykorzystano zmienną **argc_tmp** – wcześniejsze działania spowodowały, że ma ona wartość **0** i może być użyta wprost. Wartość tej zmiennej jest zwiększana o **1** przy każdym obiegu pętli, natomiast rejestr **%rbp** będący wskaźnikiem do kolejnych elementów tablicy **env**, jest zwiększany o **8** bo każdy element tablicy **env** zajmuje **8** bajtów.

24. CLR
25. Wszystkie dane przekazane do programu z zewnątrz są już wyświetlane prawidłowo.
26. Radujemy się, bo znowu się udało!