Politechnika Wrocławska

Architektura Komputerów 2 - laboratorium

sprawozdanie

Autor:
Jarosław Piszczała 209983

Prowadzący:
Prof. Janusz Biernat

Wydział Elektroniki
Informatyka
IV rok

14 listopada 2016

Spis treści

Laboratorium 1

1.1 Tematyka

Tematem laboratorium było zapoznanie się z podstawowymi działaniami z wykorzystaniem instrukcji assemblerowych. Głównym celem zajęć było przetwarzanie ciągów znaków.

1.2 Zakres prac

Zadaniem było stworzenie trzech aplikacji.

1. Program który zamieniałby, w ciągu znaków, znaki duże na małe oraz małe na duże
2. Program który implementowałby prosty szyfr cezara
3. Program który szyfrował i deszyfrowałby liczby całkowite (wykorzystując program z zadania drugiego) Program szyfrujący miałby zostać stworzony w wariancie 32 oraz 64 bitowym.

1.3 Rozwiązanie

1.3.1 Zamienianie znaków

Pierwsze zadanie nie było trudne. Dzięki obszernej instrukcji do zajęć laboratoryjnych bezproblemowo udało się wykonać ten prosty typ programu. Poniższy kod jest odpowiedzialny za pobranie, zmiane rozmiaru i ponowne zapisanie znaku.

Listing 1.1: Funkcja zmieniająca rozmiar znaków

```
function:
 movb in(,%esi,1), %al
 cmp $END_OF_LINE, %al
 je end
 cmp $'Z', %al
 jle big
 sub $DISTANCE, %al
 movb %al, out(,%esi,1)
 inc %esi
 jmp function
```

```
big:
 add $DISTANCE, %al
 movb %al, out(,%esi,1)
 inc %esi
 jmp function
```

Nie jest to najbardziej optymalny typ rozwiązania. Z aktualną wiedzą stwierdzam, że wystarczyłoby na każdej z liter użyć funkcji XOR z odpowiednią maską która zamieniałaby jeden bit na odwrotny, który odpowiada za rozmiar litery. Takie rozwiązanie zostało wprowadzone podczas pisania funkcji XOR'ującej na kolejnych laboratoriach.

1.3.2 Szyfr cezara

Szyfr cezara to jeden z najbardziej znanych typów szyfrowania. Pierwszy znak wyznacza przesunięcie znaków w alfabecie. Przyjęto, iż znak „A" oznacza brak szyfrowania. Przyjęto także, że duży pierwszy znak ma powodować szyfrowanie, a mały znak deszyfrowanie. A co dalej następuje, pierwszy znak należy najpierw wyciągnąć i zweryfikować z czym mamy do czynienia.

Listing 1.2: Rozpoznanie znaku

```
Sign:
 mov $0, %ecx
 movb in(,%esi,1), %cl

 cmp $'A', %cl
 jl end
 cmp $'z', %cl
 jg end
 cmp $'Z', %cl
 jle Decryption

Encryption:
 sub $'a', %ecx
eFunction:
 inc %esi
 movb in(,%esi,1), %al
 cmp $END_OF_LINE, %al
 je end
```

Po zapoznaniu się ze znakiem i typu działania, rozpoczynamy działanie na reszcie ciągu znaków. Każdy ze znaków należy przesunąć o odpowiednią wartość w lewo lub prawo, a następnie zweryfikować, czy nie wyszliśmy poza zakres liter. W przypadku przekroczenia zakresu liter, należy odjąć litery, aby doprowadzić do zapętlenia.

1.3.3 Szyfrowanie liczb

Z szyfrowaniem liczb nie było dużego problemu. Postawione założenie mówi, iż gdy w ciągu znaków zostaną odnalezione cyfry, należy je poprzedzić znakiem „X", a każdą z cyfr potraktować jak kolejne znaki alfabetu. Dla przykładu dla

liczby „1994” należałoby przeprowadzić konwersje na „xAIID”. Budzi to pewne obawy, gdyż przy dekodowaniu po trafieniu na znak „X” powinniśmy znaki z zakresu od A do I traktować jako liczby, a co jeśli to już będzie zakodowana treść a nie liczba? Tak więc do kodu z zadania drugiego została dorobiona funkcja która sprawdza nasz ciąg znaków i w miarę potrzeby zamienia wykryte w nim liczby według powyższego schematy

Listing 1.3: Szyfrowanie liczb całkowitych

```
 movb text(,%esi,1), %cl
 cmp $END_OF_LINE, %cl
 je trEnd
 cmp $'0', %cl
 jl trSave
 cmp $'9', %cl
 jg trSave

trNumberStart:
 movb $'X', in(,%edi,1)
 inc %edi

trNumber:
 subb $NUMBER, %cl
 addb $'A', %cl
 movb %cl, in(,%edi,1)
 inc %esi
 inc %edi
```

1.4 Wnioski

Pisanie w assemblerze nie jest prostym tematem. Należy zwracać uwagę na wiele szczegółów na które przy pisaniu chociażby w C nikt nie zwraca uwagi. Najtrudniej było rozpocząć pracę z instrukcjami, pamiętać co dzieje się z rejestrami w przypadku niektórych instrukcji. W przypadku pisania aplikacji pod system x86, przy małej ilości rejestrów bardzo pomocny okazał się bufor i stos. Inny problem sprawiło późniejsze zrozumienie co dzieje się w kodzie, gdzie z pomocą przyszedł debugger GDB. Dzięki opanowaniu tego narzędzia z powodzeniem można było przejrzeć krokowo zmiany w pamięci i rejestrze.

Laboratorium 2

2.1 Tematyka

Tematem laboratorium było zapoznanie się z działaniem stosu i tworzeniem funkcji go wykorzystujących.

2.2 Zakres prac

Zadaniem było stworzenie trzech aplikacji.

1. Program z funkcją do xor'owania liczb
2. Program z funkcją wykorzystujący pętlę sterującą (np. silnia, fibonacci)
3. Program z funkcją rekurencyjną (np. silnia, fibonacci)

Programy powinny zostać wykonane w wariancie 32 oraz 64 bitowym. Celem jest możliwość porównania czasu działania, oraz przejrzystości kodu dla dwóch typów pisania funkcji (rekurencyjnej oraz z użyciem pętli)

2.3 Rozwiązanie

2.3.1 XOR'orwanie liczb

Jako pierwsze zadanie, aby zapoznać się ze sposobem tworzenia funkcji i przekazywania argumentów stworzono funkcje xor. Funkcja ta ma stworzoną maskę do zmiany rozmiaru znaków wprowadzanych jako argument funkcji. Ważna jest tutaj przyjęta konwencja, aby argumenty na stos przed wywołaniem funkcji wprowadzać w odwrotnej kolejności. Na początku funkcji zabezpieczamy aktualną wartość rejestru „ebp" aby nie utracić adresu powrotu z funkcji. Na końcu przywracamy ten rejestr i używamy instrukcji return która wróci do miejsca gdzie została wywołana funkcja.

Listing 2.1: Funkcja XORująca

```
push %eax
call xor_func
pop %ecx

mov $EXIT, %eax
mov $ERROR, %ebx
```

```
 int $SYSCALL

xor_func:
 push %ebp
 mov %esp, %ebp
 mov 8(%ebp),%ebx

 xor $0b00100000, %ebx

 mov %ebx, 8(%ebp)
 mov %ebp, %esp
 pop %ebp
 ret
```

2.3.2 Silnia wykorzystująca rekurencje

Funkcja rekurencyjna powinna być tak zaprojektowana, aby wywoływać samą siebie w przypadku gdy nie zostanie spełniona pewna własność. W przypadku silni, uruchamiamy kolejną funkcje dopóki nie zejdziemy do pierwszej liczby. Większość kodu funkcji rekurencyjnej to wprowadzanie argumentów, ich odczyt, i wywoływanie funkcji.

Listing 2.2: Silnia - rekurencyjnie

```
Silnia:
 push %ebp
 mov %esp, %ebp
 mov 8(%ebp), %eax
 cmp $1, %eax
 je EndSilnia
 dec %eax
 push %eax
 call Silnia
 mov 8(%ebp), %ebx
 mul %ebx
EndSilnia:
 mov %ebp, %esp
 pop %ebp
 ret
```

Dla porównania ilości kodu oraz jego skomplikowania, została napisana silnia iteracyjna. Jest dużo prostsza, gdyż od razu przechodzi do liczenia, i wymnaża kolejne wartości aż do momentu gdy mnożnik będzie równy wprowadzonemu jako argument funkcjii.

Listing 2.3: Silnia - iteracyjnie

```
 mov $1, %eax
 mov $0, %ecx
Silnia:
 inc %ecx
 mul %ecx
 cmp %ebx, %ecx
 jne Silnia
```

Jak widać, silnia iteracyjna jest dużo prostsza w zapisie i łatwiejsza do czytania niż jej wersja rekurencyjna. Jest też dużo mniej awaryjna, gdyż jest mniejsza szansa na utracenie danych.

2.4 Wnioski

Pisanie funkcji w assemblerze jest podobne do pisania funkcji w C. Oczywiście nie jest ono dokładnie tak samo, gdyż wymaga od nas pamiętania o wielu szczegółach takich jak kolejność wprowadzania wartości na stos, czy zapamiętanie adresu powrotu. Mimo wszystko pisanie funkcji nie jest trudne.

Laboratorium 3

3.1 Tematyka

Tematem laboratorium było zapoznanie się ze sposobem łączenia instrukcji assemblerowych z kodem napisanym w C.

3.2 Zakres prac

Zadaniem było stworzenie trzech aplikacji wykorzystując łączenie asemblera z C:

1. Program który wczyta dane w C, przetworzy je w ASM a następnie wyświetli w C
2. Program który wczyta dane w ASM, przetworzy i wyświetli w C
3. Program który wczyta dane w C, przetworzy je za pomocą wstawki ASM a następnie wyświetli w C

3.3 Rozwiązanie

Od strony C zadanie nie wymagało dużej wiedzy. Aby skorzystać z funkcji napisanej w ASM należało dodać tutaj regułkę „extern". Dzięki temu już mogliśmy działać z naszą funkcją, a jeśli wystąpiłby błąd - były on spowodowany na pewno po stronie kodu ASM.

Listing 3.1: Funkcja obliczająca silnie iteracyjnie

```
#include <stdio.h>

extern int silnia(int num);

int main(){
        int wartosc;
        scanf("%d", &wartosc);
        int wynik = silnia(wartosc);
        printf("Silnia z %d = %d\n",wartosc, wynik);
}
```

Do połączenia wykorzystano funkcję z poprzednich zajęć do obliczania iteracyjnie silni.

Listing 3.2: Funkcja obliczająca silnie iteracyjnie

```
.text
.globl silnia
silnia:
 pushq %rbp
 movq %rsp, %rbp
 movq %rdi, %rbx

 mov $1, %eax
 mov $0, %ecx
Silnia_func:
 inc %ecx
 mul %ecx
 cmp %ebx, %ecx
 jne Silnia_func

 movq %rbp, %rsp
 pop %rbp
ret
```

Ze względu na problemy przy kompilacji oraz problem niedoczytania dokumentacji kompilatora, nie udało się wykonać na zajęciach więcej zadań. Na sprzęcie laboratoryjnym brakowało biblioteki do kompilacji kodu 32 bitowego, przez co należało przejść na kod 64 bitowy. Ten natomiast przy kompilacji za pomocą „gcc" nie wprowadzał pierwszych 6 argumentów na stos, a do rejestrów. Z tego kompilatora wycofano opcje dodania flagi, aby argumenty trafiły na stos.

3.4 Wnioski

Łączenie języków C i assembler przebiega w dużej mierze bezkolizyjnie. Może się to przydać w sytuacji gdy chcemy wykorzystać w naszym kodzie assemblerowym jakąś funkcję napisaną w C, której nie chcemy przepisywać na assemblera bądź gdy chcemy przyspieszyć pewne operacje poprzez wykonanie operacji za pomocą instrukcji assemblerowych. Problem zaczyna się, gdy chcemy przenieść się na platformę 64 bitową, gdyż nowa wersja „gcc" nie wspiera wprowadzania wszystkich argumentów na stos, przez co kod wymaga dużo większej refaktoryzacji.

Laboratorium 4

4.1 Tematyka

Tematem laboratorium było zapoznanie się z działaniem FPU.

4.2 Zakres prac

Zadaniem było stworzenie trzech aplikacji wykorzystujących łączenie ASM i C:

1. Program który odczyta i zinterpretuje dane z rejestru stanu (FPSR) oraz ustawi odpowiednie bity rejestru sterującego (FPCR)
2. Program który wytwarza poszczególne wyjątki zmiennoprzecinkowe i wyświetla odpowiednie komentarze
3. Program który pozwoli na przeprowadzenie operacji zmiennoprzecinkowych w dwóch wersjach. Pierwsza jako proste liczenie pola trójkąta, druga wykorzystująca iteracje (np. metoda Newtona-Raphsona)

Przy każdym z zadań działania powinny być przeprowadzone w kodzie ASM. Cała reszta jak pobranie argumentów czy wyświetlenie wyników powinno być wykonane w C.

4.3 Rozwiązanie

4.3.1 Odczytanie rejestru stanu

Do odczytania rejestru stanu wykorzystano specjalne instrukcje. Następnie po uzyskaniu wartości statusu, należało zweryfikować jego poszczególne bity. Nas interesowało pierwsze sześć bitów. Aby tego dokonać w jak najprostszy sposób skorzystano z intrukcji „shr" która przesuwa bity o zadaną ilość bitów w prawo, przy przesuwaniu ich o jeden bit, ten trafiał do flagi TODO (flagi przeniesienia), a tą można zweryfikować instrukcją „lea". W ten sposób sprawdzono każdy kolejny bit, i wyświetlono odpowiedni komunikat gdy natrafiono na przeniesienie.

Listing 4.1: Odczytanie i weryfikacja rejestru stanu

```
fstsw statusword
fwait
mov   statusword, %bx
```

```
  mov $1, %cl
  and $0, %rax
check_operation:
  shr %rbx
  jnc check_denormal
  leaq err_operation, %rdi
  call printf
```

Ze względu na dopieszczenie kodu do perfekcji, zmieniając metody przetwarzania statusu, to zadanie zajęło zbyt dużą część laboratorium. Odczytanie rejestru sterującego byłoby rozwiązane w sposób podobny, jednakże tam uwagę należałoby zwrócić na dwie pary bitów: 11-10 oraz 9-8. Bity sprawdzamy tak jak powyżej, z pomocą instrukcji „shr".

4.4 Wnioski

Ze względu na problemy związane ze zrozumieniem działania jednostki zmiennoprzecinkowej oraz problemu związanego z pewnymi instrukcjami na zajęciach laboratoryjnych nie udało się ukończyć drugiej części zadania. Jest to na pewno jednostka bardzo przydatna do przeprowadzania dokładnych obliczeń, z wartościami po przecinku. Pisanie kodu w assemblerze pozwala nam także na zmianę precyzji tych obliczeń oraz sposobu zaokrąglania co także można uznać za plus.