Wrocław, 17-maja-2019

Kamil Paczos, 218377
PT-N-12

Prowadzący: Janusz Biernat

Laboratorium Architektury Komputerów

(3) Łączenie języka C i Assembler, funkcje rekurencyjne

3.1 Treść ćwiczenia

Zakres i program ćwiczenia

- Zapoznanie ze sposobami łączenia kodu pisanego w językach C i Assembler
- Zapoznanie z *C calling convention*
- Zapoznanie ze sposobem tworzenia funkcji rekurencyjnych w połączonych językach C i Assembler

Zrealizowane zadania

- Utworzenie programu wywołującego funkcję napisaną w języku Assembler z poziomu języka C
- Utworzenie programu wywołującego funkcję napisaną w języku C z poziomu kodu Assembler'a
- Utworzenie programu w języku C zawierającego wstawkę *inline Assembler*
- Utworzenie rekurencyjnej funkcji języka Assembler wywoływanej z języka C
- Utworzenie rekurencyjnej funkcji w języku C wywoływanej z programu w języku Assembler

3.2 Przebieg ćwiczenia

3.2.1 Program wywołujący funkcję napisaną w języku Assembler z poziomu języka C

Do wykonania programu łączącego język Assembler z C niezbędne są dwa pliki o rozszerzeniach odpowiednio ".s" i ".c.". W przypadku tego programu plik Assemblera zawierał deklarację i implementację funkcji, która następnie wywoływana była w kodzie napisanym w języku C. Plik Assembler rozpoczynał się od zadeklarowania etykiety z nazwą funkcji oraz zadeklarowania, że jest ona funkcją. Na potrzeby programu utworzona została funkcja licząca potęgę argumentu i dzieląca ją przez dwa o nazwie pow_and_div_fct. Deklaracja wygląda więc następująco:

```
.globl pow_and_div_fct
.type pow_and_div_fct, @function
```

Wszystkie funkcje napisane na potrzeby laboratoriów posiadają na początku zapisanie, a na końcu rekonstrukcję stosu. Odpowiednie fragmenty wyglądają następująco:

```
#zapis stanu stosu
pushq %rbp
movq %rsp, %rbp
.
.
```

```
.
#przywrócenie stosu
movq %rbp, %rsp
popq %rbp
```

Pierwszym etapem działania funkcji było przeniesienie argumentu przekazanego do niej do rejestru rax celem podniesienia go do kwadratu. Zgodnie z C *calling convention* argumenty przekazywane do funkcji zapisywane są kolejno w rejestrach: rdi, rsi, rdx, rcx itd. Fragment odpowiedzialny za wyliczenie kwadratu argumentu wygląda więc następująco:

```
movq %rdi, %rax #przeniesienie argumentu do rejestru rax
mulq %rdi #pomnożenie przez samego siebie
```

Następnie wykonywane było dzielenie instrukcją div. Jego wynik zapisywany był w rejestrze rax, w którym należało umieścić również wartość zwracaną z funkcji. Można było więc bez konieczności wykonywania przenoszenia wykonać instrukcję powrotu ret.

Kolejnym etapem było wywołanie napisanej funkcji w języku C. Aby było to możliwe, należało zadeklarować ją jako funkcję zewnętrzną:

```
extern int pow_and_div_fct();
```

Następnie została ona wywołana w standardowy dla języka C sposób, a wynik jej działania wyświetlony na ekranie:

```
int arg = 4;
int result = pow_and_div_fct(arg);
printf("wynik: %d \n", result);
```

Aby zbudować wykonany program należało wykonać plik makefile umożliwiający wykorzystanie kompilatora GCC oraz zawierający informację o dołączeniu pliku zawierającego funkcję w języku Assembler.

3.2.2 Program wywołujący funkcję napisaną w języku C z poziomu języka Assembler

Na potrzeby ćwiczenia utworzony został program zawierający kod główny w języku Assambler oraz dwie funkcje w języku C: jedną służącą do liczenia potęgi argumentu i wypisania jej na ekran (o nazwie *powerfunc*) i drugą służącą do podniesienia argumentu do kwadratu, pomnożenia go przez drugi argument i wypisania wyników obu działań na ekran o nazwie *printfunc*. Obie funkcje zostały zadeklarowane jako typ void, nie zwracały więc żadnej wartości.

Program w języku Assembler rozpoczynał się od deklaracji globalnej etykiety main, gdyż w przypadku kompilacji kompilatorem gcc wymagane jest użycie jej zamiast etykiety _start. Następnie, zgodnie z przytoczoną wcześniej *C calling convention*, wykonywane jest przeniesienie parametru wywołania funkcji do rejestru rdi. Sama funkcja wywoływana jest za pomocą instrukcji call z nazwą funkcji:

```
movq $7, %rdi
call powerfunc #funkcja powerfunc wywołana z parametrem o wartości 7
```

W identyczny sposób została wywołana druga z opisanych wyżej funkcji. Jedyną różnicą była konieczność przekazania dwóch argumentów, które zostały umieszczone odpowiednio w rejestrach rdi i rsi.

3.2.3 Program w języku C zawierający wstawkę *inline Assembler*

W tym programie wykorzystany został jedynie plik z rozszerzeniem c zawierający kod w tym języku. Kod w języku Assembler został umieszczony wewnątrz kodu C w formie odpowiedniej wstawki. Na początku kodu aplikacji zostały zadeklarowany zmienne a i b o wartości odpowiednio 12 i 4. W trakcie wykonania programu liczona była ich suma, a następnie różnica. Dodatkowo w programie została zaimplementowana funkcja licząca wyznacznik równania kwadratowego (deltę) dla podanych współczynników funkcji kwadratowej.

Pierwsza wstawka Assembler'owa zawierała instrukcję dodawania oraz przeniesienia wyniku do rejestru ecx, którego wartość była przypisywana zmiennej c, która była argumentem wstawki oznaczonym jako "tylko do zapisu". Zmienne a i b zostały przekazane odpowiednio do rejestrów rax i rbx, co również zostało oznaczone w deklaracji wstawki:

```
asm(
    "addl %%ebx, %%eax \n\t"
    "movl %%eax, %%ecx \n\t"
    : "=c"(c)
    : "b" (b), "a" (a)
    );
```

Jak można zauważyć z powyższego kodu, wstawki Assembler'a mają kilka cech charakterystycznych. Ich struktura wygląda następująco: "Kod wstawki : argument zwrotny : argumenty przekazywane". Dodatkowo wszystkie odniesienia do rejestrów muszą być poprzedzone podwójnym znakiem '%' zamiast pojedynczego, jak ma to miejsce w kodzie Assembler'a. Argument „tylko do zapisu" oznaczmy za pomocą znaku '='.

Fragment odpowiedzialny za odejmowanie jest bliźniaczo podobny. Wykorzystano w nim jednak mechanizm tzw. "placeholders". Korzystając z niego programista nie odwołuje się w kodzie wstawki do konkretnych rejestrów. Zamiast tego wykorzystywane jest numeracja argumentów rozpoczynająca się od 0 i odpowiadająca kolejności ich przekazania:

```
asm(
    "sub %2, %1 \n\t"
    "movl %1, %0 \n\t"
    : "=r" (c)
    : "r" (a), "r" (b)
    ); //0 - wartość zwrotu, argumenty - 1,2...
```

W kodzie ostatniej zrealizowanej wstawki wykorzystano zarówno odwołania do placeholder'ów, jak i bezpośrednie odwołania do rejestrów. W czasie budowania programu kompilator zadbał, aby placeholder'y odpowiadały rejestrom nie wykorzystywanym w programie.

Argumenty przekazane do funkcji w języku C zostały przekazane do wstawki, w której następowały właściwe obliczenia. Obliczona wartość była zwracana z funkcji.

```
int delta(int a, int b, int c){
    int ret = 0;
    asm(
    "movl %1, %%eax \n\t" //operacja na placeholderze i rejestrze
    "mul %1 \n\t"
    "movl %%eax, %1\n\t"
    "movl %2, %%eax \n\t"
    "movl $4, %%edx \n\t"
    "mul %%edx \n\t"
    "movl %%eax, %2 \n\t"
```

```
    "movl %3, %%eax \n\t"
    "mul %2 \n\t"
    "movl %%eax, %3 \n\t"
    "sub %3, %1 \n\t"
    "movl %1, %0 \n\t"
    :"=r"(ret)
    :"r"(a), "r"(b), "r"(c)
    ); return ret; //zwrot wartosci z funkcji
}
```

3.2.4 Rekurencyjna funkcja języka Assembler wywoływana z języka C

Zadaniem wykonanym w tym ćwiczeniu było stworzenie funkcji w języku Assebmler umożliwiającej liczenie silni liczby podanej jako parametr. Zaimplementowana funkcja była funkcją rekurencyjną. Na jej początku następowało sprawdzenie, czy przekazany do funkcji parametr jest równy zero. Jeżeli tak, to oznacza to, że funkcja rekurencyjna osiągnęła najwyższy poziom i następuje zwrot wartości jeden. Jeżeli wartość parametru nie jest równa zero, to jest on dekrementowany i funkcja wywołuje samą siebie.

```
dec %rdi #dekrementacja parametru i wywołanie funkcji rekurencyjnej
call fact
```

Po wykonaniu funkcji najwyższego poziomu jej wynik zwracany był poziom niżej. Tam mnożony był przez parametr, który na ten poziom był przekazany. Następnie obliczona wartość zwracana była na niższy poziom. Na najniższym poziomie wywołania wartość zwracana była do programu w języku C, z którego funkcja została wywołana.

3.2.5 Rekurencyjna funkcja w języku C wywoływana z programu w języku Assembler

W ostatnim programie zadaniem było stworzenie programu bliźniaczego do poprzedniego, tj. przeznaczonego do obliczania silni. Tym razem główna część programu została napisana w języku Assembler, a sama funkcja rekurencyjna w języku C. Przyjmowała ona jako parametr liczbę. Jeżeli była ona równa 0, to funkcja zwracała 1. W przeciwnym wypadku funkcja zwracała parametr pomnożony przez wartość zwróconą z wywołania samej siebie z parametrem pomniejszonym o 1:

```
int fact(int n){
      if(n == 0) return 1;
      return(n * fact(n-1));
}
```

W kodzie głównym programu liczba, której silnia była liczona, umieszczona została w rejestrze rdi, pierwszym w kolejce wg. *C calling convention*. Po zakończeniu wykonywania funkcji rekurencyjnej wynik wyświetlany był na ekranie za pomocą funkcji systemowej printf. W tym celu łańcuch formatujący został umieszczony w rejestrze rdi, a jego parametry w rdx i rsi. Rejetr rax został załadowany wartością 0. Następnie nastąpiło wywołanie funkcji printf za pomocą instrukcji *call printf*

3.3 Wnioski

Łączenie kodu napisanego w języku C z kodem języka Assembler jest stosunkowo łatwe i bardzo praktyczne przy pisaniu wydajnych aplikacji. Należy pamiętać przede wszystkim o kolejności przekazywania i zwracania wartości w używanych funkcjach. Najmniej praktyczne wydają się być wstawki Assembler'owe umieszczane bezpośrednio w kodzie. W trakcie laboratoriów zrealizowano cały jego zaplanowany program. Podczas testowania wykonanych programów stwierdzono poprawność ich działania.

(4) Jednostka zmiennoprzecinkowa FPU

4.1 Treść ćwiczenia

Zakres i program ćwiczenia:

- Zapoznanie ze rejestrem kontrolnym i rejestrem stanu jednostki zmiennoprzecinkowej.
- Badanie zachowań w momencie wygenerowania błędów.
- Stworzenie aplikacji wykonującej obliczenia zmiennoprzecinkowe.

Zrealizowane zadania:

- Program umożliwiający konfigurację jednostki zmiennoprzecinkowej.
- Program odczytujący i interpretujący stan wybranych bitów rejestru statusu FPU.
- Program generujący wyjątki w operacjach zmiennoprzecinkowych.
- Program obliczający pole trójkąta. *[wykonany po zajęciach]*

4.2 Przebieg ćwiczenia

4.2.1 Program konfigurujący jednostkę zmiennoprzecinkową.

Jednostka zmiennoprzecinkowa konfigurowana jest za pomocą specjalnego rejestru. Na zajęciach wykonany został program umożliwiający jej konfigurację w zakresie precyzji obliczeń i trybu zaokrąglania. W języku C został napisany kod, który na drodze interakcji z użytkownikiem pobierał informację o żądanym ustawieniu parametrów. Następnie informacja ta przekazywana była do funkcji stworzonej w języku Assembler.

Jako pierwsza w programie wywoływana była instrukcja inicjalizacji jednostki zmiennoprzecinkowej:

```
finit
```

Powodowała ona ustawienie rejestru kontrolnego na domyślną wartość. Następnie wykonywane były operacje przesunięcia w lewo rejestrów zawierających dane przekazane do funkcji. Było to konieczne ze względu na strukturę rejestru kontrolnego. Informacja o precyzji została pobrana od użytkownika w postaci liczby dwu bitowej. Natomiast w rejestrze bity odpowiadające za ustalenie precyzji znajdują się na miejscu 8 i 9. Konieczne było wobec tego przesunięcie bitów pobranej liczby o 8 miejsc. „Wolne" miejsce zostało wypełnione zerami. Podobnie przesunięte zostały bity kontroli zaokrąglenia, z tym, że tutaj przesunięcie nastąpiło o 10 miejsc, na pozycje 10 i 11.

```
finit #inicjalizacja FPU
#przesuń rejestr o 8
shlq $8, %rdi
#przesuń rejestr o 10
shlq $10, %rsi
```

W kolejnym kroku przy użyciu instrukcji *fstcw* pobierana była aktualna konfiguracja rejestru sterującego FPU. Następnie wykonywana była operacja AND na pobranej wartości i masce bitowej o wartości *000011111111.* Miało to na celu „wyzerowanie" bitów odpowiedzialnych za sterowanie ustawianymi wartościami. W kolejnym kroku następowała operacja OR z oboma przesuniętymi wcześniej rejestrami.

Cały schemat ustawienia sterowania prezentuje tabela 1:

Tabela 1 Przebieg operacji generowania wartości rejestru sterującego

Wartość bazowa rejestru kontrolnego	011010010101
Maska bitowa	000011111111
Operacja AND	000010010101
Maska ustawienia precyzji	001000000000
Operacja OR	001010010101
Maska ustawienia trybu zaokrąglania	110000000000
Operacja OR	111010010101

Do załadowania nowej wartości do rejestru kontrolnego wykorzystana została instrukcja *fldcw*. Następnie, w celu weryfikacji, jego wartość została pobrana przy wykorzystaniu używanej wcześniej instrukcji *fstcw.* Weryfikacja wykazała, ze wartości podane przez użytkownika zostały ustawione poprawnie.

4.2.2 Program odczytujący i interpretujący stan wybranych bitów rejestru statusu FPU.

W celu zrealizowania postawionego zadania został wykonany program w języku C oraz funkcja w języku Assembler sprawdzająca i zwracająca wartość wskazanego bitu rejestru stanu FPU. Na początku programu w języku C zadeklarowano dwie tablice stałych. Jedna z nich zawierała komunikaty odpowiadające każdemu z 16 bitów rejestru. Druga zawierała potęgi liczby 2, począwszy od 1 do 32768. Jak wiadomo, potęgi liczby 2 odpowiadają kolejnym bitom w zapisie binarnym, tj. 1 = 0000..001, 2 = 0000…10, 4 = 0000…100 itd. Wykorzystano ten fakt do generowania masek bitowych dla kolejnych bitów rejestru stanu. Program zawierał pętlę for dla iteratora ‘i’ z przedziału <0;15>. W każdym przebiegu pętli wywoływana była Assembler’owa funkcja, a jako jej parametr przekazywana była kolejna wartość z tablicy potęg liczby 2. W zależności od wartości zwróconej z funkcji był wyświetlany lub pomijany komunikat błędu odpowiadający danemu bitowi.

Funkcja sprawdzająca bity rejestru przygotowana została z myślą o pracy na prawdziwym rejestrze stanu. Jednak do celów testowych wykorzystany został rejestr rax załadowany wartością *01001010*. Na masce przekazanej do funkcji i rejestrze rax wykonywana była operacja *test*. W momencie, gdy na ustawiona została flaga *ZF* wiadomo było, że wskazany bit miał wartość zero. Wynika to z zasady działania funkcji test, która jest porównywalna do działania funkcji AND, jednak nie nadpisuje rejestrów biorących udział w porównaniu.

```
#maska przekazana jest do funkcji w postaci dziesiętnej potęgi 2
test %rdi, %rax
#jeżeli test zwróci flagę ZF, to znaczy, że zadany bit był równy 0.
```

Następnie, w zależności od ustawionej bądź nie flagi *ZF* rejestr rax ładowany był wartością 1 dla wskazanego bitu równego 1 i 0 dla bitu równego 0.

```
jz ret_no_e
movq $1, %rax #zwróć - bit ustawiony był na 1
jmp end #zakończ
ret_no_e:
movq $0, %rax #zwróć - bit ustawiony był na 0
jmp end #zakończ
```

Wykonany program poprawnie zinterpretował przykładową wartość rejestru symulującego rejestr stanu.

4.2.3 Program generujący wyjątki w obliczeniach zmiennoprzecinkowych

Pierwszym problemem napotkanym przy realizacji programu były wyjątki przerywające działanie programu w momencie próby wykonania nieprawidłowej operacji, na przykład dzielenia przez 0. Aby je obejść, rejestr kontrolny ustawiono na wartość 0 na bitach od 0 do 5. Odpowiadają one za maskowanie wyjątków zgłaszanych przez jednostkę zmiennoprzecinkową. Następnie wywołana została funkcja Assembler'owa mająca wykonać dzielenie z nieprawidłowym argumentem. W tym celu na stosie umieszczono przy wykorzystaniu instrukcji *flds* kolejno wartości 0.00 i 45.23. Następnie wykonano operację:

```
fdiv %st(1)
```

W momencie jej wykonania stos FPU wyglądał tak jak w tabeli 2:

Tabela 2 Zawartość stosu FPU

st(0)	45.23
st(1)	0
st(2)	0
.	.
.	.
.	.

Po wykonaniu nieprawidłowego dzielenia zawartość stosu nie zmieniła się. Zmianie uległa za to wartość rejestru stanu. Po wykonaniu dzielenia wyniosła ona w zapisie szesnastkowym B084, co odpowiada binarnej wartości 1011 0000 1000 0100. Po zapoznaniu się z dokumentacją widać, ze są to bity odpowiedzialne za flagi *„Zero divide exception"* i *„Error summary flag".* Widać więc, że jednostka poprawnie zinterpretowała błąd i udało się go odczytać. Dalsze wykonanie programu zostało przerwane.

Drugim programem generującym wyjątki był program generujący błąd operacji na zbyt dużych liczbach. W celu wywołania tego wyjątku na stos dwukrotnie przekazano liczbę 340028230000000000000000000000000000000.123. Następnie wykonano jej podniesienie do kwadratu. W kolejnym kroku ponowiono tę operację. Działania te spowodowały powstanie błędu w obliczeniach zmiennoprzecinkowych. Po sprawdzeniu rejestru statusu zaobserwowano wartość B2A0, czyli binarnie: 1011 0010 1010 0000. Zgodnie z dokumentacją bity te odpowiadają za *„Error summary flag"* i *„Precision exception".* Widać więc, że operacje na tak dużych liczbach spowodowały błędy związane z niemożliwością zapewnienia pożądanej precyzji obliczeń.

4.2.4 Program liczący pole trójkąta *[wykonany po zajęciach].*

W ostatnim programie został stworzony algorytm liczący pole trójkąta. Do przykładowych obliczeń wykorzystany został trójkąt o podstawie równej 37823478123812781273812482175 6.012412313213 i wysokości równej 0.000123123. Po inicjacji jednostki zmiennoprzecinkowej na stos przeniesiona została podstawa trójkąta oraz wartość 0.5.

```
fmul %st(1), %st(0) #mnożenie 0.5 * a
```

Po ich pomnożeniu na stosie umieszczona została wysokość trójkąta. Ostatnim krokiem było pomnożenie wcześniej obliczonej części 0.5*a przez h. Otrzymany wynik był polem zadanego trójkąta.

4.3 Wnioski

Wykonane programy w większości działały zgodnie z zamierzeniami. W programie liczącym pole trójkąta pojawił się problem z dokładnością – obliczenia zostały zaokrąglone do liczb całkowitych. Problematyczne było również generowanie wyjątków, wskutek czego udało się zaobserwować jedynie działanie dwóch z nich.

Literatura

[1]"The 64 bit x86 C Calling Convention", Aaronbloomfield.github.io. [Online]. Available: https://aaronbloomfield.github.io/pdr/book/x86-64bit-ccc-chapter.pdf. [Accessed: 13- May- 2019].

[2]"Chapter Eleven Real Arithmetic", Plantation-productions.com. [Online]. Available: http://www.plantation-productions.com/Webster/www.artofasm.com/Windows/HTML/RealArithmetic.html. [Accessed: 13- May- 2019].

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