Wydział Fizyki i Informatyki Stosowanej
Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie

Wprowadzenie do assemblera x86-64

Damian Płóciennik

22 kwietnia 2020

Spis treści

1 Wprowadzenie

1.1 Proces asemblacji i konsolidacji

1.1.1 Asemblacja

W celu dokonania asemblacji z wykorzystaniem asemblera **nasm** należy wykonać polecenie:

```
nasm -f elf64 nazwa.asm -o nazwa.o
```

gdzie **elf64** to format pliku wyjściowego, **nazwa.asm** to nazwa pliku źródłowego a **nazwa.o** to nazwa pliku wyjściowego.

1.1.2 Konsolidacja (linkowanie)

Aby dokonać konsolidacji (linkowania) należy wykonać polecenie:

```
ld nazwa.o -o nazwa
```

gdzie **nazwa.o** to nazwa pliku wejściowego a **nazwa** to nazwa pliku wyjściowego.

1.2 Interfejsowanie z C

Możliwe jest wykorzystywanie funkcji napisanych w języku assembly w programach utworzonych w języku C. Aby tego dokonać, w pliku C funkcja musi zostać oznaczona jako zewnętrzna, np.:

```
extern int suma(int, int);
```

natomiast w pliku asemblerowym należy zdefiniować odpowiedni symbol jako globalny, aby był widoczny w innych jednostkach kompilacji:

```
global suma
suma:
; tutaj nalezy umiescic cialo funkcji suma
ret
```

Na koniec każdy z plików należy skompilować osobno i zlinkować razem. Aby tego dokonać można wykonać ciąg instrukcji:

```
nasm -f elf64 suma.asm -o suma.o
gcc main.c suma.o
```

1.2.1 Konwencja wywołań System V AMD64 ABI

W trybie 64 bitowym stosowana jest konwencja wywołań **System V AMD64 ABI**. Jej najważniejsze założenia to:

- pierwsze sześć argumentów, które są liczbami całkowitymi lub wskaźnikami (w kolejności od lewej do prawej) są umieszczane kolejno w rejestrach RDI, RSI, RDX, RCX, R8, R9,
- argumenty zmiennoprzecinkowe (poza long double) są umieszczane w rejestrach SSE: XMM0, XMM1, ..., XMM7,
- rejestr RAX powinien zawierać liczbę liczb zmiennoprzecinkowych umieszczonych w SSE,
- pozostałe argumenty są umieszczane na stosie w kolejności od prawej do lewej,
- funkcja musi zachowywać wartość rejestrów: RBX, RBP, RSP, R12, R13, R14, R15,
- pozostałe rejestry (w tym zmiennoprzecinkowe) mogą być dowolnie modyfikowane,
- Wynik jest umieszczany w:
  - RAX - całkowitoliczbowy, wskaźnik,
  - XMM0 - zmiennoprzecinkowy,
  - ST0 - long double,
  - pamięci pod adresem wskazanym przez RDI jeżeli zwraca się typ złożony.

Pełna specyfikacja System V AMD64 ABI jest dostępna pod adresem `https://github.com/hjl-tools/x86-psABI/wiki/X86-psABI`.

1.3 Wstawki asemblerowe

Innym sposobem na wykorzystanie języka assembly w programach napisanych w C i C++ są wstawki asemblerowe.

Wstawki asemblerowe rozpoczynają się od wyrażenia "asm(" lub "__asm(", natomiast kończą się nawiasem zamykającym ")". Po każdej instrukcji należy umieścić znak przejścia do nowej linii lub kolejne instrukcje oddzielać średnikiem i umieszczać w osobnych łańcuchach znakowych.

```
asm("movl %ecx, %eax\n
     addl %ebx %eax\n");

asm("movl %ecx, %eax;"
    "addl %ebx %eax;");
```

Listing 1: Przykłady wstawek asemblerowych.

Jak łatwo zauważyć domyślnie wykorzystywana jest składnia AT&T. Możliwe jest jednak wymuszenie składni Intela poprzez wykorzystanie dyrektywy ". intel_syntax noprefix" i na koniec przywrócenie ponownie składni AT&T z której korzysta GCC używając dyrektywy ".att_syntax prefix".

```
asm(".intel_syntax noprefix\n"
    "mov eax, ecx \n"
    "add eax, ebx \n"
    ".att_syntax prefix");
```

Listing 2: Zmiana składni na składnię Intela i ponowne przywrócenie składni AT&T.

Możliwe jest także tworzenie wstawek złożonych, które pozwalają m.in. na przekazywanie zmiennych do konkretnych rejestrów. Ogólna postać wstawki asemblerowej ma postać:

```
 asm ( "szablon instrukcji asemblerowych"
: wyjsciowe operandy              /* optional */
: wejsciowe operandy              /* optional */
: lista niszczonych obiektow      /* optional */
);
```

Listing 3: Ogólna postawć wstawki asemblerowej.

Operandy należy podawać jako listę oddzieloną przecinkami elementów postaci "X" (wartosc), gdzie X określa wymagania, co do miejsca umieszczenia wartości np.:

- "a"(123) - wartość 123 ma zostać umieszczona w rejestrze EAX,
- "r"(rozmiar) - wartość zmiennej rozmiar ma zostać umieszczona w dowolnym rejestrze ogólnego przeznaczenia,
- "m"(dlugosc) - wartość zmiennej dlugosc ma zostać umieszczona w pamięci i przekazany będzie adres jej lokalizacji,
- "i"(0 xfffffc00) - wartość 0 xfffffc00 ma zostać wstawiona do kodu jako argument instrukcji.

Przed operandami wyjściowymi dodatkowo należy umieścić znak = np. "=r" (wartosc_zwracana). Operandy są dostępne wewnątrz wstawki asemblerowej pod kolejnymi literałami %0, %1, %2... Najczęściej stosowane restrykcje zostały zaprezentowane na poniższym listingu:

```
; Rejestry ogolnego przeznaczenia
"a" = rax, eax, ax, al
"b" = rbx, ebx, bx, bl
"c" = rcx, ecx, cx, cl
"d" = rdx, edx, dx, dl
"S" = rsi, esi, si
"D" = rdi, edi, di
"r" = dowolny rejestr ogolnego przeznaczenia
"q" = jeden z rejestrow a, b, c, d

; Adres pamieci
"m" = zmienna w pamieci,
operacje sa wykonywane bezposrednio na pamieci,
powinno byc uzywane gdy nie chcemy przechowywac
```

```
wartosci w rejestrze

; Rejestry zmiennoprzecinkowe
"f" = dowolny rejestr zmiennoprzecinkowy
"t" = ST0
"u" = ST1

; Stale
"i" = stala calkowita,
"I" = stala z zakresu 0..31 (dla przesuniec)
"N" = stala z zakresu 0..255
"E" = stala zmiennoprzecinkowa
```

Listing 4: Najczęściej stosowane restrykcje w stawkach assemblerowych.

Coraz częściej odchodzi się od wstawek asemblerowych na rzecz funkcji wewnętrznych (intristic functions), które zazwyczaj wykonują konkretną operację asemblerową. Temat ten nie zostanie poruszony w czasie tych laboratoriów, jednak osoby chętne zachęcam do zapoznania się z literaturą oraz The Intel Intrinsics Guide, dostępnym pod adresem: `https://software.intel.com/sites/landingpage/IntrinsicsGuide/`.

1.4 Materiały z seminarium

Seminarium znajduje się pod adresem `https://youtu.be/NW8NP7TxnLI`, natomiast slajdy i przykłady z tego wystąpienia znajdują się w dołączonym do projektu pliku assembler-seminar.pdf.

2 Zadania do wykonania

2.1 Zadanie 1

2.1.1 Cel zadania

Celem zadania jest zapoznanie z pisaniem kodu w języku asembly, poznanie konwencji wywołań System V AMD64 ABI wykorzystywanej w systemach 64 bitowych, a także pokazanie możliwości łączenia funkcji napisanych w asembly z kodem w języku C.

2.1.2 Treść zadania

Zaimplementuj algorytm wyszukiwania binarnego uzupełniając funkcję find w załączonym pliku (find.asm). Funkcja powinna mieć możliwość wywołania z poziomu języka C, zgodnie z konwencją wywołań System V. Przyjmij, że parametry tej funkcji to kolejno wskaźnik do pierwszego elementu posortowanej niemalejąco tablicy liczb całkowitych (int), rozmiar tej tablicy (int) oraz poszukiwana wartość (int). Kod testujący poprawne wykonane ćwiczenia znajduje się w pliku test_find.c.

```
1 section .text
2 global find
3 find:
4     ;Tutaj zaimplementuj algorytm
5     ret
```

Listing 5: Zawartość pliku find.asm.

```
1  #include <stdio.h>
2  #define ARRAY_SIZE(arr) (sizeof(arr) / sizeof((arr)[0]))
3  #define TEST_FUNCTION(arr, val, ind) (printf("%s", find((arr), ARRAY_SIZE((arr)
       ), (val)) == (ind) ? "Test PASSED!\n" : "Test FAILED!\n"))
4
5  extern int find(int *array, int size, int value);
6
7  int main(void) {
8      int array1[] = {7};
9      TEST_FUNCTION(array1, 7, 0);
10     TEST_FUNCTION(array1, 9, -1);
11     int array2[] = {1, 3, 4, 6, 8, 9, 11};
12     TEST_FUNCTION(array2, 6, 3);
13     TEST_FUNCTION(array2, 12, -1);
14     TEST_FUNCTION(array2, 1, 0);
```

```
15      TEST_FUNCTION(array2, 9, 5);
16      int array3[] = {1, 3};
17      TEST_FUNCTION(array3, 7, -1);
18      TEST_FUNCTION(array3, 1, 0);
19      TEST_FUNCTION(array3, 3, 1);
20      int array4[] = {3, 9, 161, 342, 7777};
21      TEST_FUNCTION(array4, 3, 0);
22      TEST_FUNCTION(array4, 7777, 4);
23      TEST_FUNCTION(array4, 161, 2);
24      TEST_FUNCTION(array4, 341, -1);
25  }
```

Listing 6: Zawartość pliku test_find.c testującego poprawność napisanego kodu.

2.1.3 Wyszukiwanie binarne

Wyszukiwanie binarne to jeden z najpopularniejszych algorytmów do wyszukiwania danych w tablicy posortowanych liczb. Algorytm ten został oparty o metodę „dziel i zwyciężaj" dzięki czemu potrafi w czasie logarytmicznym znaleźć szukaną liczbę oraz zwrócić jej indeks jeśli została odnaleziona. Dla przykładu przy tablicy liczb, która zawiera milion elementów, wyszukiwanie binarne do odnalezienia żądanej liczby potrzebuje wykonać zaledwie 20 porównań. Niestety, aby to wszystko zadziałało przeszukiwana tablica musi zostać wcześniej posortowana.

Kolejne kroki poszukiwania wartości w tablicy prezentują się następująco:

1. Wybierz środkowy element tablicy.
2. Sprawdź, czy wybrany element jest równy szukanemu elementowi. Jeśli tak, zwróć jego pozycje i zakończ szukanie. Jeśli nie to z tablicy utwórz podtablicę na zasadzie:
   - jeśli element środkowy jest większy niż poszukiwany, zawęź tablicę do jej lewej strony, tzn. w dalszej części będziemy pracować na elementach od lewego krańca do obecnie środkowego elementu,
   - jeśli element środkowy jest mniejszy niż poszukiwany, zawęź tablicę do jej prawej strony. tzn. w dalszej części będziemy pracować na elementach od obecnie środkowego elementu do prawego końca tablicy.
3. Punkty 1. i 2. powtarzaj do momentu znalezienia szukanego elementu. Jeśli to nie nastąpi zwróć -1 lub inną wartość oznaczającą brak szukanego elementu.

2.2 Zadanie 2

2.2.1 Cel zadania

Celem zadania jest nabycie umiejętności wykorzystania wstawek asemblerowych w językach C i C++, a także pokazanie różnic w szybkości wykonywania programu napisanego z wykorzystaniem wstawek ASM i bez nich przy różnych poziomach optymalizacji.

2.2.2 Treść zadania

Napisz funkcję odwracającą kolejność elementów w tablicy z wykorzystaniem wstawek asemblerowych w języku C++. Następnie napisz drugą funkcję robiącą to samo w pełni w języku C++. Zmierz i porównaj czasy wykonywania obu funkcji. Sprawdź rezultaty dla różnych poziomów optymalizacji.

Dodatkowo, zachęcam do porównania otrzymanych wyników z czasem wykonania **std::reverse** z biblioteki standardowej C++ (należy dołączyć nagłówek **algorithm**).
Dokumentacja std::reverse: `https://en.cppreference.com/w/cpp/algorithm/reverse`

2.2.3 Mierzenie czasu w C++

Do zmierzenia czasu w języku C++ najłatwiej wykorzystać bibliotekę chrono dostępną od C++11. Aby z niej skorzystać należy do naszego programu dołączyć nagłówek **chrono**.

```cpp
auto start = std::chrono::system_clock::now();

/* wykonanie tego, czego czas chcemy zmierzyc */

auto end = std::chrono::system_clock::now();
auto elapsed = std::chrono::duration_cast<std::chrono::nanoseconds>(end - start
    );
std::cout << elapsed.count() << 'ns\n';
```

Listing 7: Przykładowy fragment kodu mierzący czas wykonywania danego fragmentu kodu.

Dokumentacja chrono: `https://en.cppreference.com/w/cpp/chrono`

3 Bibliografia

Swoistą biblią dla programistów assembly jest manual Intela dostępny pod adresem `https://software.intel.com/en-us/articles/intel-sdm`,

1. Asembler - Wikipedia [online]
   `https://pl.wikipedia.org/wiki/Asembler`
   [dostęp: 10 marca 2020]

2. x86 assembly language - Wikipedia [online]
   `https://en.wikipedia.org/wiki/X86_assembly_language`
   [dostęp: 10 marca 2020]

3. Assembly - Wikibooks [online]
   `https://en.wikibooks.org/wiki/X86_Assembly`
   [dostęp: 10 marca 2020]

4. Programowanie niskopoziomowe [online]
   `http://ww2.ii.uj.edu.pl/~kapela/pn/print-lecture-and-sources.php`
   [dostęp: 10 marca 2020]