Laboratorium Architektury Komputerów

Ćwiczenie 4

Programowanie mieszane

Wprowadzenie

Współcześnie, bardziej złożone oprogramowanie tworzone jest przez zespoły kilku lub kilkunastu programistów. Zazwyczaj każdy z nich koduje (programuje) jeden lub więcej modułów funkcjonalnych, realizujących wyraźnie wyodrębnione operacje tworzonej aplikacji. W tego rodzaju pracach pożądane jest, by każdy z programistów miał szeroką swobodę działania, ograniczoną jedynie przez te elementy oprogramowania, które wiążą ze sobą poszczególne moduły funkcjonalne.

Powyższy postulat realizuje się poprzez kodowanie oprogramowania w postaci wielu oddzielnych plików, zawierających kod źródłowy programu. Zazwyczaj pojedynczy programista tworzy kilka takich plików. Są one następnie tłumaczone na kod zrozumiały przez procesor i scalane (konsolidowane), tworząc kompletny, gotowy program zapisany w pliku *.EXE* (system Windows) czy *.out* (system Linux).

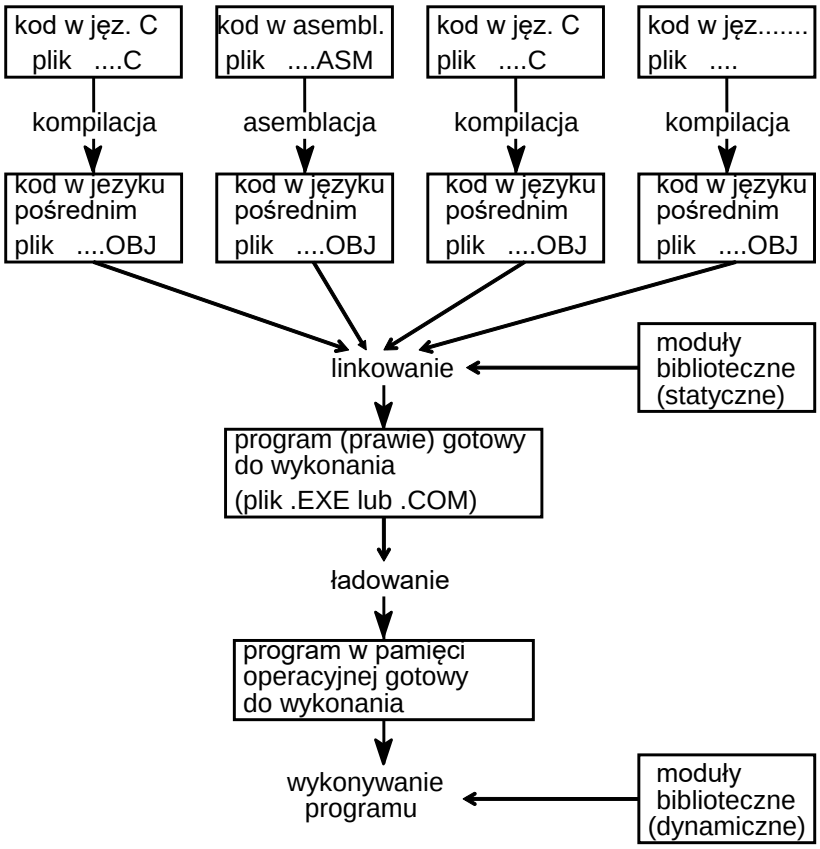
W przypadku tworzenia oprogramowania współpracującego z urządzeniami niestandardowymi, zadaniem jednego z modułów funkcjonalnych jest organizowanie współpracy z tym urządzeniem. W wielu przypadkach, ze względu na specyficzne wymagania urządzenia, taki moduł musi być kodowany w asemblerze, niekiedy przez konstruktora urządzenia. W rozpatrywanej sytuacji kod asemblerowy musi być przystosowany do współdziałania z pozostałym oprogramowaniem, kodowanym zazwyczaj w języku wysokiego poziomu (np. C/C++).

Niniejsze opracowanie przybliża zagadnienia związane z współdziałaniem kodu napisanego w asemblerze z kodem w języku C (i po pewnych rozszerzeniach C++), w środowisku systemu Windows. Bardzo podobne, lub identyczne mechanizmy stosowane są w innych systemach operacyjnych.

Kompilacja, konsolidacja (linkowanie) i ładowanie

W wielu środowiskach programowania wytworzenie programu wynikowego wykonywane jest w dwóch etapach. Najpierw kod źródłowy każdego modułu programu zostaje poddany *kompilacji* (jeśli moduł napisany jest w języku wysokiego poziomu) lub *asemblacji* (jeśli moduł napisany jest w asemblerze). W obu tych przypadkach uzyskuje się plik w języku pośrednim (rozszerzenie .OBJ). Następnie uzyskane pliki .OBJ poddaje się konsolidacji czyli *linkowaniu*. W trakcie linkowania dołączane są także wszystkie niezbędne programy biblioteczne. W rezultacie zostaje wygenerowany plik zawierający program wynikowy z rozszerzeniem .EXE. Plik ten zawiera kod programu w języku maszynowym (czyli zrozumiałym przez procesor), aczkolwiek niektóre jego elementy wymagają korekcji uzależnionej od środowiska, w którym program będzie wykonany. Korekcja ta następuje w trakcie *ładowania* programu.

Niektóre programy biblioteczne mają charakter uniwersalny i są wykorzystywane przez wiele programów użytkowych. Wygodniej byłoby więc dołączać te programy dopiero w trakcie wykonywania programu, co pozwoliłoby na zmniejszenie rozmiaru pliku .EXE. W takim przypadku mówimy, że program korzysta z biblioteki dynamicznej (zapisanej w pliku z rozszerzeniem DLL). Omawiane fazy translacji pokazane są na poniższym rysunku.



Pliki .OBJ generowane przez różne kompilatory (w danym środowisku) zawierają kod w tym samym języku, który możemy uważać za język pośredni, stanowiący jak gdyby "wspólny mianownik" dla różnych języków programowania.

Podprogramy w technice programowania mieszanego

Problem tworzenia programu, którego fragmenty napisane są w różnych językach programowania wymaga m.in. ustalenia sposobu komunikowania się poszczególnych fragmentów ze sobą. Komunikacja taka staje się stosunkowo łatwa do zrealizowania, jeśli poszczególne fragmenty programu mają postać podprogramów (procedur). Podprogramy stanowią, ze swej natury, w pewien sposób wyizolowaną część programu, a komunikacja z nimi odbywa się wg ściśle ustalonego protokołu, określającego formaty danych i wzajemne obowiązki programu wywołującego i wywoływanego podprogramu. Protokół ten nazywany jest także opisem interfejsu podprogramu (procedury). W ten sposób, w trakcie wykonywania programu, wywoływanie fragmentów napisanych w różnych językach programowania,

sprowadza się do wywoływania odpowiednich podprogramów. W przypadku języka C wywołanie podprogramu oznacza po prostu wywołanie funkcji języka C, której kod został zdefiniowany w innym pliku, niekoniecznie napisanym w języku C.

Powyższe rozważania wskazują, że interfejs do podprogramów musi być jasno i przejrzyście zdefiniowany, a zarazem musi być na tyle uniwersalny, by mógł być implementowany przez kompilatory różnych języków programowania. Z tego powodu producenci oprogramowania (m.in. firma Microsoft) ustalają pewne niskopoziomowe protokoły wywoływania podprogramów, przeznaczone dla wytwarzanych przez nich kompilatorów języków programowania.

Omawiane protokoły, a także związane z nimi różne reguły i ustalenia opisujące współpracę między modułami tego samego programu, jak również między modułami programu a systemem operacyjnym czy bibliotekami, określane są jako interfejs ABI (ang. Application Binary Interface). Interfejs ABI różni się tym od interfejsu API, że dotyczy programów w wersji binarnej lub skompilowanej (w języku pośrednim) podczas gdy interfejs API dotyczy kodu źródłowego.

Interfejs ABI definiuje sposób wywoływania funkcji, przekazywania argumentów i wyników, określa wymagania dotyczące zachowania rejestrów, postępowania z parametrami przekazywanymi przez stos, itp. W dalszym ciągu rozpatrzymy szczegóły interfejsu ABI dotyczące trybu 32-bitowego, a w dalszej części omówimy nieco bardziej złożony interfejs stosowany w trybie 64-bitowym.

Konwencje wywoływania podprogramów stosowane w trybie 32-bitowym

W oprogramowaniu komputerów osobistych rodziny PC, wyłoniły się trzy typy interfejsu procedur. Jeden z nich używany jest przez kompilatory języka C (*standard C*), drugi przez kompilatory Pascala (*standard Pascal*), a trzeci standard *StdCall* stanowiący połączenie dwóch poprzednich, używany jest w systemie Windows do wywoływania funkcji wchodzących w skład interfejsu Win32 API.

Główne różnice między standardami dotyczą kolejności ładowania parametrów na stos i obowiązku usuwania parametrów, który należy najczęściej do wywołanego podprogramu (funkcji), jedynie w standardzie C zajmuje się tym program wywołujący. W standardzie Pascal parametry wywoływanej funkcji zapisywane są na stos kolejności od lewej do prawej, natomiast w standardzie C i StdCall od prawej do lewej. Istnieją też opisane dalej inne różnice.

Standard	Kolejność ładowania na stos	Obowiązek zdjęcia parametrów
Pascal	od lewej do prawej	wywołany podprogram
C	od prawej do lewej	program wywołujący
StdCall	od prawej do lewej	wywołany podprogram

Dalsze wymagania są następujące.

1. W trybie 32-bitowym parametry podprogramu przekazywane są przez stos. W standardach C i StdCall parametry ładowane są na stos w kolejności odwrotnej w

stosunku do tej w jakiej podane są w kodzie źródłowym, np. wywołanie funkcji `calc (a,b)` powoduje załadowanie na stos wartości `b`, a następnie `a`.

2. Jeśli parametr ma postać pojedynczego bajtu, to na stos ładowane jest podwójne słowo (32 bity), którego najmłodszą część stanowi podany bajt.
3. Jeśli parametrem jest liczba 64-bitowa (8 bajtów), to najpierw na stos ładowana jest starsza część liczby, a następnie jej młodsza część. Taki schemat ładowania stosowany jest w komputerach, w których liczby przechowywane są w standardzie *mniejsze niżej* (ang. little endian) i wynika z faktu, że stos rośnie w kierunku malejących adresów.
4. Obowiązek zdjęcia parametrów ze stosu po wykonaniu podprogramu w przypadku standardu C należy do programu wywołującego. Funkcje systemowe Windows stosują standard Stdcall, w którym parametry zapisane na stosie zdejmowane są wewnątrz wywołanej funkcji. Również w standardzie Pascal parametry zdejmowane są wewnątrz wywołanej funkcji.
5. W standardzie C jeśli parametrem funkcji jest nazwa tablicy, to przekazywany jest adres tej tablicy.
6. Wyniki podprogramu przekazywane są przez rejestr EAX. Wyniki 8-bitowe przekazywane są przez rejestr AL, a 16-bitowe przez rejestr AX. Jeśli wynikiem podprogramu jest adres (wskaźnik), to przekazywany jest także przez rejestr EAX. Jeśli wynikiem jest liczba zmiennoprzecinkowa typu float lub double, to wynik ten dostępny jest na wierzchołku stosu rejestrów koprocesora.
7. Jeśli podprogram zmienia zawartość rejestrów EBX, EBP, ESI, EDI, to powinien w początkowej części zapamiętać je na stosie i odtworzyć bezpośrednio przed zakończeniem. Pozostałe rejestry robocze mogą być używane bez konieczności zapamiętywania i odtwarzania ich zawartości. Uwaga: rejestr ESP jest wskaźnikiem stosu i nie może być używany do przechowywania danych.
8. Ponadto znaczniki operacji arytmetycznych i logicznych (w rejestrze znaczników) mogą być używane bez ograniczeń. Znacznik DF powinien być zerowany zarówno przed wywołaniem podprogramu, jak i wewnątrz podprogramu przed rozkazem RET, jeśli używane były rozkazy operacji blokowych (np. MOVSB).
9. Obok typowych dyrektyw do definiowania danych: `db`, `dw`, `dd`,... w asemblerze dostępne są także ich odpowiedniki w postaci `byte`, `word`, `dword`,... Przykładowo, dwa poniższe wiersze są równoważne:

```
liczba    dw        1234
liczba    word      1234
```

10. Podana dalej tabela zawiera zestawienie dyrektyw używanych do definiowania danych wraz z odpowiadającymi im typami danych języka C/C++. Typowe dyrektywy `db`, `dw`, `dd`,... zachowują swoją uniwersalność i mogą być nadal stosowane do definiowania liczb stało- i zmiennoprzecinkowych ze znakiem lub bez znaku. Jednak użycie ich odpowiedników w postaci dyrektyw `byte`, `sbyte`, `word`, ... pozwala na bardziej precyzyjne określanie właściwości danych i ogranicza możliwość występowania błędów.

Rozmiar	Dyrektywa	Synonim	Odpowiednik C/C++
1 bajt	`byte`	`db`	`unsigned char`
	`sbyte`		`char`
2 bajty	`word`	`dw`	`unsigned short`
	`sword`		`short`
4 bajty	`dword`	`dd`	`unsigned int, unsigned long`
	`sdword`		`int, long`
	`real4`		`float`
6 bajtów	`fword`	`df`	
8 bajtów	`qword`	`dq`	
	`sqword`		
	`real8`		`double`
10 bajtów	`tbyte`	`dt`	
	`real10`		

Podprogramy kodowane w asemblerze

Omawiany wyżej *standard C* jest standardem domyślnym dla programów napisanych w językach C i C++ (programy w C++ wymagają dodatkowych działań — zob. dalszy opis). Opcjonalnie można zdefiniować funkcję (podprogram), która będzie wywoływana w standardzie *StdCall* lub *Pascal*.

Podprogram w asemblerze przystosowany do wywoływania z poziomu języka C musi być skonstruowany dokładnie wg tych samych zasad co funkcje w języku C. Wynika to z faktu, że program w języku C będzie wywoływał podprogram w taki sam sposób, w jaki wywołuje inne funkcje w języku C.

Wszystkie nazwy globalne zdefiniowane w treści podprogramu w asemblerze muszą być wymienione na liście dyrektywy PUBLIC. Jednocześnie nazwy innych używanych zmiennych globalnych i funkcji muszą być zadeklarowane na liście dyrektywy EXTERN (lub EXTRN).

Ze względu na konwencję nazw stosowaną przez kompilatory języka C, każdą nazwę o zasięgu globalnym wewnątrz podprogramu asemblerowego należy poprzedzić znakiem podkreślenia _ (nie dotyczy to standardu *StdCall*).

Technika przekazywania parametrów przez stos

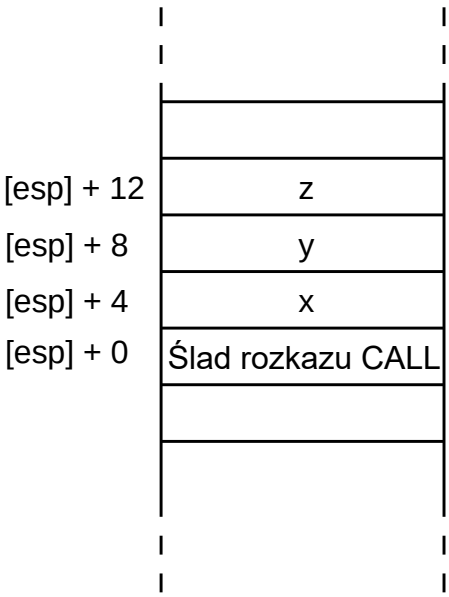
Mechanizmy przekazywania parametrów przez stos rozpatrzmy na przykładzie funkcji (podprogramu)

```
int szukaj_max (int a, int b, int c);
```

która wyznacza największą liczbę całkowitą, spośród trzech liczb podanych jako argumenty funkcji. Podana funkcja, wraz z odpowiednimi parametrami, zostanie wywołana na poziomie języka C, ale kod funkcji zostanie napisany w asemblerze. Przykładowy program w języku C, w którym wywoływana jest omawiana funkcja może mieć postać:

```
#include <stdio.h>
int szukaj_max (int a, int b, int c);

int main()
{
  int x, y, z, wynik;
  printf("\nProszę podać trzy liczby całkowite ze znakiem: ");
  scanf_s("%d  %d  %d", &x, &y, &z, 32);

  wynik = szukaj_max(x, y, z);

  printf("\nSpośród podanych liczb %d, %d, %d, \
  liczba %d jest największa\n", x,y,z, wynik);

  return 0;
}
```

Uwaga: znak \ umieszczony na końcu wiersza sygnalizuje, że dalszy ciąg wiersza umieszczony jest w następnej linii. Znak \ jest zbędny, jeśli cała instrukcja da się zapisać w jednym wierszu.



W trakcie wykonywania podanego programu przez procesor, bezpośrednio przed wywołaniem funkcji `szukaj_max` zostaną wykonane trzy rozkazy `push`, które umieszczą na stosie aktualne wartości zmiennych `z`, `y`, `x` (parametry ładowane są na stos w kolejności od prawej do lewej). Następnie zostanie wykonany rozkaz `call`, który wywoła omawianą funkcję (podprogram). Zarówno trzy rozkazy `push`, jak i rozkaz `call` stanowią fragment kodu programu, który został wygenerowany przez kompilator języka C. Po wykonaniu rozkazu `call` procesor rozpocznie wykonywanie kolejnych rozkazów podprogramu (funkcji) `szukaj_max`.

W celu wyznaczenia największej liczby spośród podanych `x,y,z`, wywołany podprogram musi oczywiście odczytać te liczby ze stosu. Jednak odczytywanie parametrów ze stosu za pomocą rozkazu `pop` byłoby kłopotliwe: wymagałoby uprzedniego odczytania śladu rozkazu `call`, a po wykonaniu obliczeń należało by ponownie załadować tę wartość na stos. Odczytane parametry można by umieścić w rejestrach ogólnego przeznaczenia — rejestry te jednak używane są do wykonywania obliczeń i przechowywania wyników pośrednich. W tej sytuacji umieszczenie wartości `x,y,z` w rejestrach ogólnego przeznaczenia mogłoby znacznie utrudnić kodowanie podprogramu ze względu na brak wystarczającej liczby rejestrów.

W celu zorganizowania wygodnego dostępu do parametrów umieszczonych na stosie przyjęto, że obszar zajmowany przez parametry będzie traktowany jako zwykły obszar danych. W istocie stos jest bowiem umieszczony w pamięci RAM i nic nie stoi na

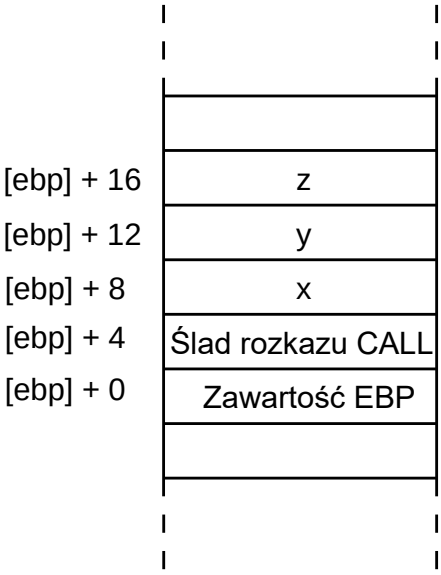
przeszkodzie, by w pewnych sytuacjach traktować jego zawartość jako zwykły obszar danych.

Jeśli więc przyjąć, że obszar stosu stanowi zwykły obszar danych, to dostęp do danych znajdujących się w obszarze stosu wymaga znajomości ich adresów. W każdej chwili znane jest położenie wierzchołka stosu: wskaźnik stosu ESP określa adres komórki pamięci, w której znajduje dana ostatnio zapisana na stosie, czyli wierzchołek stosu. Po wykonaniu rozkazu `call` na wierzchołku stosu znajduje się ślad tego rozkazu, a powyżej wierzchołka stosu (posuwając się górę, czyli w głąb stosu) znajduje się wartość `x`, jeszcze dalej `y`, i w końcu `z`. Ponieważ każda wartość zapisana na stosie zajmuje 4 bajty, więc wartość `x` znajduje się w komórce pamięci o adresie równym zawartości rejestru ESP powiększoną o 4, co na rysunku oznaczone jest jako `[esp] + 4`. Analogicznie wartość `y` dostępna jest pod adresem `[esp] + 8`, a wartość `z` pod adresem `[esp] + 12`.

Ponieważ zawartość rejestru ESP może się zmieniać w trakcie wykonywania podprogramu (np. wskutek wykonywania rozkazów `push` i `pop`), pożądane jest użycie innego rejestru, którego zawartość, ustalona przez cały czas wykonywania podprogramu, będzie wskazywała obszar parametrów na stosie — rolę tę pełni, specjalnie do tego celu zaprojektowany rejestr EBP. Jeśli zawartość rejestru EBP będzie równa zawartości ESP, to w podanych wyrażeniach symbol `esp` można zastąpić przez `ebp`.

Zgodnie z podanymi wcześniej wymaganiami interfejsu ABI, użycie w podprogramie rejestru EBP wymaga zapamiętania jego zawartości na początku podprogramu i odtworzenia w końcowej części podprogramu. Zatem przed skopiowaniem zawartości rejestru ESP do EBP konieczne jest zapamiętanie zawartości rejestru EBP na stosie. Ostatecznie więc dwa pierwsze rozkazy podprogramu będą miały postać:

```
push  ebp      ; zapisanie zawartości EBP na stosie
mov   ebp,esp  ; kopiowanie zawartości ESP do EBP
```



Rozkazy te występują prawie zawsze na początku podprogramu i określane są jako standardowy prolog podprogramu (funkcji).

Zapisanie zawartości rejestru EBP na stosie spowodowało zmianę wyrażeń adresowych opisujących położenie wartości `x,y,z`. Aktualna sytuacja na stosie pokazana jest na rysunku obok.

W tym momencie można przystąpić do poszukiwania liczby największej. W kodzie programu w języku C określono typ parametrów funkcji `szukaj_max` jako `int`, co oznacza że parametry te są 32-bitowymi liczbami ze znakiem (kodowanymi w systemie U2). W trakcie porównywania liczb używać będziemy więc rozkazów `jg jge`, `jl`, `jle`. Najpierw porównywane są wartości `x` i `y` — jeśli liczba `x` jest większa lub równa od `y`, to następnie wartość `x` jest porównywana z wartością `z`, a w przeciwnym razie wykonywane jest porównywanie wartości `y` i `z`. Wynik końcowy, stanowiący największą liczbę spośród trzech porównywanych, pozostawia się w rejestrze EAX, skąd zostanie później odczytany przez rozkazy wygenerowane przez kompilator języka C.

Zauważmy ponadto, że w standardzie C parametry ze stosu usuwane przez program wywołujący, czyli nie wykonujemy tej operacji wewnątrz podprogramu. Należy też pamiętać, że w języku C małe i wielkie litery nie są utożsamiane. Asembler MASM (`ml.exe`) odróżnia małe i wielkie litery tylko wówczas, jeśli w linii wywołania asemblera podano opcję `-Cp`. Kod podprogramu w asemblerze podany jest poniżej.

```
.686
.model flat

public _szukaj_max

.code

_szukaj_max    PROC
     push      ebp       ; zapisanie zawartości EBP na stosie
     mov       ebp, esp  ; kopiowanie zawartości ESP do EBP

     mov       eax, [ebp+8]   ; liczba x
     cmp       eax, [ebp+12]  ; porownanie liczb x i y
     jge       x_wieksza      ; skok, gdy x >= y

; przypadek x < y
     mov       eax, [ebp+12]  ; liczba y
     cmp       eax, [ebp+16]  ; porownanie liczb y i z
     jge       y_wieksza      ; skok, gdy y >= z

; przypadek y < z
; zatem z jest liczbą największą
wpisz_z:  mov  eax, [ebp+16]  ; liczba z

zakoncz:
     pop       ebp
     ret

x_wieksza:
     cmp       eax, [ebp+16]  ; porownanie x i z
     jge       zakoncz        ; skok, gdy x >= z
     jmp       wpisz_z

y_wieksza:
     mov       eax, [ebp+12]  ; liczba y
     jmp       zakoncz

_szukaj_max    ENDP
               END
```

Asemblacja, kompilacja i konsolidacja w przypadku programowania mieszanego

Podane tu kody programów w języku C i asemblerze trzeba umieścić w plikach z rozszerzeniem `.c` i `.asm`.

> Nazwy plików rozszerzeniem `.c` i `.asm`. nie mogą być jednakowe!

W celu wytworzenia programu wynikowego zazwyczaj korzystamy ze środowiska zintegrowanego Microsoft Visual Studio. Postępowanie jest prawie takie samo jak opisano w instrukcji do ćwiczenia 1. W trakcie tworzenia projektu trzeba wybrać odpowiedni asembler. W tym celu, w oknie Solution Explorer należy kliknąć prawym klawiszem myszki na nazwę projektu i z rozwijanego menu wybrać opcję Build Dependencies / Build Customizations. W rezultacie na ekranie pojawi się okno dialogowe, w którym należy zaznaczyć pozycję masm i nacisnąć OK. Następnie, do projektu należy dodać pliki zawierające kod w języku C i kod w asemblerze. W przypadku programowania mieszanego nie wpisuje się nazwy biblioteki `libcmt.lib` do opcji linkera.

Możliwe jest także wykorzystanie kompilatora i asemblera zewnętrznego, tak jak to opisano w instrukcji do ćwiczenia 1. Tworzenie programów w kodzie wykonywalnym w przypadku programowania mieszanego wymaga stosowania, nieznacznie rozbudowanych, technik opisanych w instrukcji do ćwiczenia 1. I tak plik wsadowy `a32.bat` opisany w instrukcji ćwiczenia 1 przyjmie teraz postać pliku `ac32.bat` o następującej zawartości:

```
@echo Asemblacja, kompilacja i linkowanie programu 32-bitowego
cl  -c %2.c
if errorlevel 1 goto koniec
ml  -c  -Cp  -coff  -Fl  %1.asm
if errorlevel 1 goto koniec
link -subsystem:console -out:%2.exe  %1.obj  %2.obj
:koniec
```

Korzystając z tego pliku, przetłumaczenie pliku źródłowego zawartego w plikach, np. `podaj_m.asm` i `oblicz.c` wymaga wprowadzenia polecenia

```
ac32  podaj_m  oblicz
```

Zauważmy, pierwszym parametrem jest nazwa pliku zawierającego kod źródłowy w asemblerze, ale bez rozszerzenia `.asm`, a drugim parametrem jest nazwa pliku zawierającego kod w języku C, także bez rozszerzenia.

Jeśli asemblacja (programu w asemblerze), kompilacja (programu w języku C) i konsolidacja (linkowanie) zostaną wykonane poprawnie, to powstanie plik `. . . .exe` zawierający kod programu gotowy do wykonania. W celu wykonania programu wystarczy wpisać nazwę programu do okienka konsoli i nacisnąć klawisz Enter (nie trzeba podawać rozszerzenia `.exe`).

W fazie konsolidacji (linkowania) programu pojawia się czasami błąd `unresolved external symbol`. Błąd ten wynika z braku jednej lub kilku funkcji (podprogramów) niezbędnych do utworzenia programu wynikowego. Najczęstszą przyczyną tego błędu jest pominięcie asemblacji pliku `.asm` — w takim przypadku należy wskazać odpowiedni asembler poprzez kliknięcie prawym klawiszem myszki na nazwę projektu (okno Solution

Explorer), wybranie opcji Build Dependencies / Build Customizations.i zaznaczenie kwadracika dla wymaganego asemblera.

Omawiany błąd może być także spowodowany pominięciem znaku podkreślenia _ przed nazwą funkcji w kodzie asemblerowym (ale w trybie 64-bitowym znak podkreślenia nie jest stosowany).

Zadanie 4.1. Napisać podprogram `szukaj4_max`, stanowiący rozszerzenie przykładu podanego na str. 8. Prototyp podprogramu ma postać:

```
int szukaj4_max (int a, int b, int c, int d);
```

Podprogram powinien wyznaczyć największą liczbę spośród podanych jako parametry podprogramu. Napisać także krótki program w języku C ilustrujący sposób wywoływania podprogramu.

Przykład przekazywania parametrów przez adres

Szerokie możliwości tworzenia efektywnych rozwiązań programistycznych otwierają się poprzez wykorzystanie techniki przekazywania wartości parametrów przez adres — na poziomie języka C wymaga to przekazywania wskaźnika do zmiennej. Podana niżej funkcja `plus_jeden`, zakodowana w asemblerze, powoduje zwiększenie o 1 wartości zmiennej, wskaźnik do której jest argumentem funkcji. Prototyp tej funkcji ma postać:

```
void  plus_jeden (int * a);
```

Zauważmy, że wynik działania funkcji nie jest zwracany przez nazwę, ale jest wpisywany do zmiennej zdefiniowanej w programie w języku C — wskaźnik do tej zmiennej jest argumentem funkcji `plus_jeden`. Poniżej podano przykładowy program w języku C, w którym wywoływana jest omawiana funkcja. W trakcie wykonywania programu na ekranie zostanie wyświetlona liczba `-4` .

```
#include <stdio.h>
void plus_jeden(int * a);
int main()
{
  int  m;
  m = -5;

  plus_jeden(&m);

  printf("\n m = %d\n", m);
  return 0;
}
```

W podanym kodzie programu argumentem funkcji `plus_jeden` jest wskaźnik do zmiennej `m`. Oznacza to, że bezpośrednio przed wywołaniem tej funkcji na stosie zostanie umieszczony adres zmiennej `m`. Z kolei w kodzie asemblerowym podprogramu (funkcji) można odczytać ten adres, następnie znając adres zmiennej można wyznaczyć jej wartość,

potem dodać 1, a uzyskany wynik wpisać do zmiennej. Operacje te wykonywane są przez niżej podany podprogram w asemblerze.

```
.686
.model flat
public  _plus_jeden
.code

_plus_jeden PROC
     push      ebp       ; zapisanie zawartości EBP na stosie
     mov       ebp,esp   ; kopiowanie zawartości ESP do EBP
     push      ebx       ; przechowanie zawartości rejestru EBX

; wpisanie do rejestru EBX adresu zmiennej zdefiniowanej
; w kodzie w języku C
     mov       ebx, [ebp+8]

     mov       eax, [ebx]     ; odczytanie wartości zmiennej
     inc       eax            ; dodanie 1
     mov       [ebx], eax     ; odesłanie wyniku do zmiennej

; uwaga: trzy powyższe rozkazy można zastąpić jednym rozkazem
; w postaci:  inc   dword PTR [ebx]

     pop       ebx
     pop       ebp
     ret
_plus_jeden    ENDP
               END
```

Zadanie 4.2. Wzorując się przykładem funkcji `plus_jeden` napisać w asemblerze kod funkcji `liczba_przeciwna`, która wyznaczy liczbę przeciwną do znajdującej się w zmiennej. Napisać krótki program w języku C do testowania opracowanej funkcji.

Zadanie 4.3. Poniższy program w języku C wczytuje liczbę całkowitą z klawiatury, następnie zmniejsza ją o 1 i wyświetla na ekranie wynik obliczenia. Zmniejszenie liczby o 1 wykonuje podprogram zakodowany w asemblerze, przystosowany do wywoływania z poziomu języka C w trybie 32-bitowym, którego prototyp na poziomie języka C ma postać:

```
void  odejmij_jeden (int ** liczba);
```

Argument `liczba` jest adresem zmiennej, w której przechowywany jest adres, pod którym przechowywana jest liczba (adres adresu).
Napisać podprogram w asemblerze dokonujący opisanego obliczenia i uruchomić program składający się z plików źródłowych w języku C i w asemblerze.

```
#include <stdio.h>
void odejmij_jeden(int ** a);
```

```
int main()
{
  int k;
  int * wsk;

  wsk = &k;
  printf("\nProsze napisac liczbe: ");
  scanf_s("%d", &k, 12);

  odejmij_jeden(&wsk);

  printf("\nWynik = %d\n", k);
  return 0;
}
```

Podprogramy wykonujące działania na elementach tablic

Jeśli argumentem funkcji w języku C jest tablica, to na stosie zapisywany jest adres tej tablicy, ściślej: adres elementu tablicy o indeksie 0. Technikę wykonywania operacji na tablicach wyjaśnimy na przykładzie funkcji `przestaw`, która zamienia kolejno pary sąsiednich elementów tablicy liczb całkowitych, jeśli są ułożone w kolejności malejącej. Prototyp omawianej funkcji ma postać:

```
void przestaw (int tabl[], int n);
```

Funkcja porównuje kolejne pary n-elementowej tablicy liczb całkowitych (typu `int`) i w przypadku stwierdzenia, że para porównywanych elementów tworzy ciąg malejący, zamienia liczby miejscami. Wielokrotne wywoływanie tej funkcji powoduje posortowanie elementów tablicy (sortowanie bąbelkowe). Kod asemblerowy funkcji `przestaw` podany jest poniżej.

```
.686
.model flat
public  _przestaw
.code

_przestaw PROC
     push      ebp       ; zapisanie zawartości EBP na stosie
     mov       ebp,esp   ; kopiowanie zawartości ESP do EBP
     push      ebx       ; przechowanie zawartości rejestru EBX

     mov       ebx, [ebp+8]   ; adres tablicy tabl
     mov       ecx, [ebp+12]  ; liczba elementów tablicy
     dec       ecx

; wpisanie kolejnego elementu tablicy do rejestru EAX
ptl: mov       eax, [ebx]

; porównanie elementu tablicy wpisanego do EAX z następnym
```

```
     cmp       eax, [ebx+4]
     jle       gotowe ; skok, gdy nie ma przestawiania

; zamiana sąsiednich elementów tablicy
     mov       edx, [ebx+4]
     mov       [ebx], edx
     mov       [ebx+4], eax

gotowe:
     add       ebx, 4  ; wyznaczenie adresu kolejnego elementu
     loop      ptl       ; organizacja pętli
     pop       ebx       ; odtworzenie zawartości rejestrów
     pop       ebp
     ret                 ; powrót do programu głównego
_przestaw      ENDP
               END
```

Zadanie 4.4. Napisać program przykładowy w języku C ilustrujący możliwości wykorzystania podanej funkcji do sortowania tablicy liczb całkowitych. Zwrócić uwagę, że w po każdym wywołaniu rozmiar tablicy podany jako drugi parametr funkcji powinien zostać zmniejszony o 1 (dlaczego ?).

Uwagi dodatkowe dotyczące kodowania programów 32-bitowych

Specyfika kompilatorów języka C++

Opisane tu zasady w pewnym stopniu dotyczą także kompilatorów języka C++. Główna trudność polega na konieczności uwzględnienia zmian nazw funkcji – zmiany nazw wykonywane są przez kompilator języka C++ w trakcie kompilacji. Zmiany opisane są zazwyczaj w dokumentacji kompilatora, ale ich uwzględnienie jest dość kłopotliwe. Z tego powodu zazwyczaj funkcje zakodowane w asemblerze wywołujemy w programie w języku C++ przy zastosowaniu interfejsu języka C. W takim przypadku obowiązują podane wyżej zasady, a prototyp funkcji musi być poprzedzony kwalifikatorem `extern "C"`, np.:

```
extern  "C"  int  szukaj_max (int * tablica, int n);
```

Specyfika programowania mieszanego dla funkcji kodowanych wg standardu `stdcall` w kodowaniu 32-bitowym[1]

Omawiany wyżej *standard C* jest standardem domyślnym dla programów napisanych w języku C. Jednak w funkcjach zdefiniowanych w interfejsie Win32 API (np. `MessageBox`) stosowany jest zazwyczaj standard `_stdcall`, który w przypadku kodowania funkcji w asemblerze wymaga stosowania nieco innych reguł. Szczegóły dotyczące tego standardu opisane są poniżej.

[1] Materiał zawarty w tym podrozdziale jest nadobowiązkowy.

W standardzie `_stdcall` stosowanym przez kompilatory firmy Microsoft nazwa funkcji po kompilacji (zawarta w pliku .obj) zawiera także liczbę bajtów zajmowanych przez parametry przekazywane do funkcji, przy czym nazwa poprzedzona jest znakiem podkreślenia _. Przykładowo, nazwa funkcji

```
iloczyn_liczb (int a, int b, int c);
```

zawarta w programie w języku C po kompilacji przyjmie postać `_iloczyn_liczb@12`. Do podanej funkcji przekazywane są bowiem trzy parametry, z których każdy zajmuje 32 bity (4 bajty).

Jeśli funkcję w standardzie `_stdcall` zamierzamy zakodować w asemblerze, to konieczne jest przekazanie asemblerowi informacji o liczbie i rozmiarach parametrów przekazywanych do funkcji. Informacje takie podaje się w wierszu dyrektywy `PROC`, np.

```
suma_liczb  PROC  stdcall, arg1:dword, arg2:dword,arg3:dword
```

Taka konstrukcja powoduje jednak pewne dodatkowe działania asemblera:

1. Nazwa funkcji (podprogramu) zostaje poprzedzona znakiem podkreślenia _.
2. Asembler automatycznie generuje rozkazy `push ebp` oraz `mov ebp, esp`, więc należy je pominąć w kodzie funkcji w asemblerze.
3. Asembler automatycznie generuje rozkaz `pop ebp` przed rozkazem `ret` (ściśle: generowany jest rozkaz `leave`, który w tym przypadku działa tak jak `pop ebp`).
4. Do rozkazu `ret` dopisywany jest dodatkowy parametr, np. `ret 12`, tak by rozkaz ten usunął parametry ze stosu (standard `stdcall` wymaga, by parametry ze stosu zostały usunięte przez wywołaną funkcję — czynność tę wykonuje właśnie rozkaz `ret` z parametrem).
5. W kodzie asemblerowym można (ale nie jest to obowiązkowe) używać podanych argumentów `arg1`, `arg2`,... zamiast wyrażeń adresowych [EBP+8], [EBP+12], itd.

Jeśli funkcja kodowana w asemblerze ma wejść w skład biblioteki dynamicznej DLL, to po słowie `PROC` trzeba umieścić parametr `EXPORT`, np.

```
suma_liczb  PROC  stdcall EXPORT, arg1:dword, ....
```

Ponadto, w prototypie funkcji na poziomie języka C musi wystąpić parametr `__stdcall` (dwa znaki podkreślenia _), np.

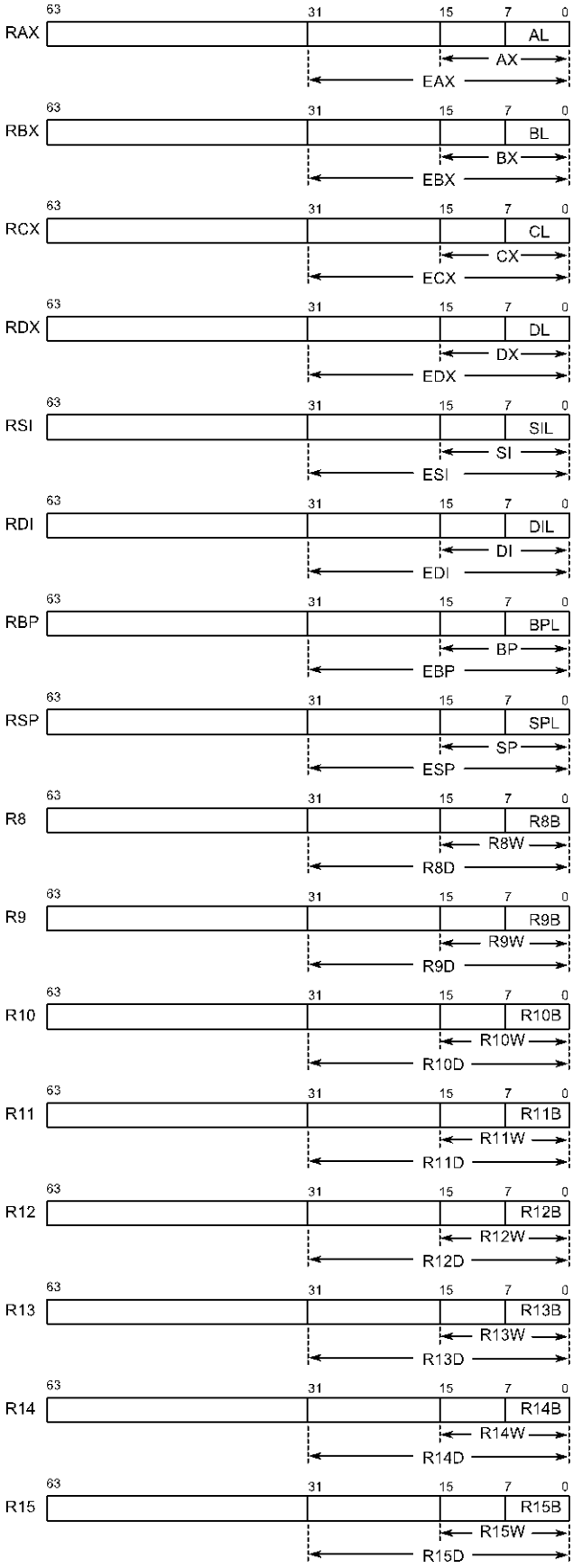
```
int __stdcall suma_liczb(int a, int b, int c);
```

Jeśli wszystkie funkcje w programie w języku C będą tworzone zgodnie ze standardem *StdCall*, to można wprowadzić opcję kompilatora `/Gz` (lub `-Gz`). Jeśli później pojawi się konieczność wprowadzenia nowej funkcji działającej zgodnie ze standardem C, to w prototypie tej funkcji musi wystąpić parametr `__cdecl`.

Programowanie mieszane w trybie 64-bitowym

Rejestry używane w programowaniu 64-bitowym

Rejestry dostępne w trybie 64-bitowym pokazano na rysunku. W trybie tym stosowane są operandy 32- i 64-bitowe, ale domyślny rozmiar operandu wynosi 32 bity, Operandami 32-bitowymi mogą być typowe rejestry EAX, EBX, ..., ESP, oraz nowe rejestry R8D ÷ R15D, a operandami 64-bitowymi mogą być rejestry RAX, RBX, ..., RSP, R8 ÷ R15. Dostępne są młodsze części nowych rejestrów R8 ÷ R15, np. w rejestrze R8 można wyróżnić fragmenty: R8B (8 bitów), R8W (16 bitów), R8D (32 bitów). Istnieje także możliwość odwołania się do najmłodszych bajtów rejestrów 64-bitowych: RBP, RSP, RDI, RSI poprzez nazwy, odpowiednio, BPL, SPL, DIL, SIL. Ponadto istnieją ograniczenia dotyczące używania rejestrów 8-bitowych: AH, BH, CH, DH.


RAX
RBX
RCX
RDX
RSI
RDI
RBP
RSP
R8
R9
R10
R11
R12
R13
R14
R15
63
31
15
7
0
AL
AX
EAX
BL
BX
EBX
CL
CX
ECX
DL
DX
EDX
SIL
SI
ESI
DIL
DI
EDI
BPL
BP
EBP
SPL
SP
ESP
R8B
R8W
R8D
R9B
R9W
R9D
R10B
R10W
R10D
R11B
R11W
R11D
R12B
R12W
R12D
R13B
R13W
R13D
R14B
R14W
R14D
R15B
R15W
R15D

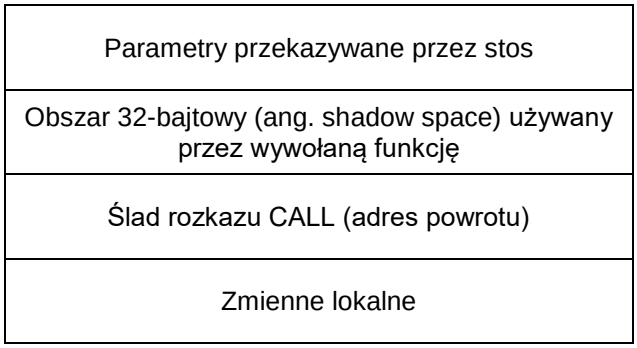
Konwencje wywoływania procedur stosowane przez kompilatory języka C w trybie 64-bitowym (w systemie MS Windows)

1. W trybie 64-bitowym pierwsze cztery parametry podprogramu przekazywane są przez rejestry: RCX, RDX, R8 i R9. Dopiero piąty parametr i następne, jeśli występują, przekazywane są przez stos, przy czym pierwszy z parametrów przekazywanych przez stos musi zajmować lokację pamięci o najniższym adresie, który musi być podzielny przez 8. Tak więc jeśli liczba parametrów przekracza 4, to parametry ładowane są na stos w kolejności od prawej do lewej, z wyłączeniem czterech pierwszych parametrów z lewej strony (które przekazywane są przez rejestry).

2. W trybie 64-bitowym do przekazywania liczb zmiennoprzecinkowych używa się odrębnych rejestrów związanych z operacjami multimedialnymi SSE: XMM0, XMM1, XMM2, XMM3 (zamiast rejestrów RCX, RDX, R8 i R9).

3. Wyniki podprogramu przekazywane są przez rejestr RAX. Jeśli wynikiem podprogramu jest adres (wskaźnik), to przekazywany jest także przez rejestr RAX. Jeśli wynikiem jest liczba zmiennoprzecinkowa typu float lub double, to wynik przekazywany jest przez rejestr XMM0. Sposób przekazywania wyników w innych trybach opisuje podana dalej tabela — zawiera ona także informacje dotyczące ograniczeń w zakresie używania rejestrów w różnych rodzajach aplikacji.

4. Bezpośrednio przed wywołaniem funkcji trzeba zarezerwować na stosie obszar 32-bajtowy. Obszar ten może wykorzystany w wywołanej funkcji (podprogramie) do przechowywania zawartości czterech rejestrów ogólnego przeznaczenia. Rezerwacja omawianego obszaru, który określany czasami angielskim terminem *shadow space,* jest wymagana także w przypadku, gdy liczba przekazywanych parametrów jest mniejsza niż 4. Rezerwację wykonuje się poprzez zmniejszenie wskaźnika stosu RSP o 32. Sytuację na stosie ilustruje poniższy rysunek.



5. Ponadto istnieje dodatkowe wymaganie: przed wykonaniem rozkazu skoku do podprogramu (tj. przed rozkazem `CALL`) wskaźnik stosu RSP musi wskazywać adres podzielny przez 16. Pominięcie tego wymagania powoduje zazwyczaj zakończenie wykonywania programu wraz z komunikatem, że program wykonał niedozwoloną operację.

6. Zauważmy, że warunek podany w pkt. 5 nie jest spełniony bezpośrednio po rozpoczęciu wykonywania kodu wywołanej funkcji — rozkaz `CALL` zapisał bowiem 8-bajtowy ślad na stosie, wskutek czego zawartość rejestru RSP nie będzie podzielna przez 16. Oznacza to, że jeśli wewnątrz wywołanej funkcji zamierzamy wywołać inną funkcję (z co najwyżej czterema parametrami), to musimy zarezerwować (32 + 8) bajtów — rezerwacja dodatkowych 8 bajtów wynika z konieczności spełnienia warunku by rejestr RSP był podzielny przez 16.
7. Dodatkowo, liczba bajtów obszaru zajmowanego przez parametry (zob. pkt. 1) musi stanowić wielokrotność 16. Przykładowo, jeśli wywoływana funkcja ma 7 parametrów, to przed wywołaniem tej funkcji trzeba zarezerwować 64 bajty:
    - 8 bajtów – dopełnienie do wielokrotności 16 bajtów (bo liczba bajtów obszaru zajmowanego przez parametry musi stanowić wielokrotność 16),
    - 24 bajty – trzy parametry przekazywane przez stos,
    - 32 bajty – obszar przewidziany do wykorzystania przez wywołaną funkcję (*shadow space*),

8. Liczba bajtów zajmowanych przez tak skonstruowany obszar stanowi wielokrotność 16. Jednak konieczne jest także uwzględnienie wymagania dotyczącego podzielności przez 16 zawartości rejestru RSP (zob. pkt. 5). W praktyce programowania, bezpośrednio przed dokonaniem rezerwacji opisanej w pkt. 7, można łatwo sprawdzić czy zawartość RSP jest podzielna przez 16 (wówczas 4 najmłodsze bity zawierają same zera) – jeśli nie, to zawartość RSP trzeba zmniejszyć o 8 (rezerwacja 8 bajtów).

9. Zwolnienie stosu wykonuje program, który umieścił dane na stosie lub zarezerwował obszar.

10. W kodzie asemblerowym nie stosuje się znaków podkreślenia przed nazwami funkcji systemowych i znaków @ (wraz z liczbą) po nazwie funkcji, np. w trybie 64-bitowym wywołanie funkcji `MessageBoxW` będzie miało postać:

```
call  MessageBoxW
```

11. Ponadto podprogramy wykonywane w trybie 64-bitowym w systemie Windows powinny pozostawiać niezmienione zawartości następujących rejestrów: RBX, RSI, RDI, RBP, R12 ÷ R15, XMM6 ÷ XMM15. W praktyce oznacza to, że podane rejestry (jeśli są używane) muszą być zapamiętywane na początku podprogramu i odtwarzane w końcowej części podprogramu. Pełne zestawienie wymagań dotyczących rejestrów w systemach Windows i Linux zawiera tabela na następnej stronie.

	Aplikacje 16-bitoweDOS, Windows	Aplikacje 32-bitoweWindows, Linux	Aplikacje 64-bitoweWindows	Aplikacje 64-bitowe Linux
Rejestry używane bez ograniczeń	AX, BX, CX, DX, ES, ST(0) ÷ ST(7)	EAX, ECX, EDX, ST(0) ÷ ST(7) XMM0 ÷ XMM7	RAX, RCX, RDX, R8 ÷ R11, ST(0) ÷ ST(7) XMM0 ÷ XMM5	RAX, RCX, RDX, RSI, RDI, R8 ÷ R11, ST(0) ÷ ST(7) XMM0 ÷ XMM15
Rejestry, które muszą być zapamiętywane i odtwarzane	SI, DI, BP, DS.	EBX, ESI, EDI, EBP	RBX, RSI, RDI, RBP, R12 ÷ R15, XMM6 ÷ XMM15	RBX, RBP, R12 ÷ R15
Rejestry, które nie mogą być zmieniane		DS, ES, FS, GS, SS		
Rejestry używane do przekazywania parametrów		(ECX)	RCX, RDX, R8, R9, XMM0 ÷ XMM3	RDI, RSI, RDX, RCX, R8, R9, XMM0 ÷ XMM7
Rejestry używane do zwracania wartości	AX, DX, ST(0)	EAX, EDX, ST(0)	RAX, XMM0	RAX, RDX, XMM0, XMM1, ST(0), ST(1)

Program przykładowy w wersji 64_bitowej: szukanie największej liczby w tablicy

Część programu w języku C (plik `szukaj64c.c`*)*

```
/* Poszukiwanie największego elementu w tablicy liczb
   całkowitych za pomoca funkcji (podprogramu)
   szukaj64_max, ktora zostala zakodowana w asemblerze.
   Wersja 64-bitowa
*/

#include <stdio.h>
extern __int64 szukaj64_max (__int64 * tablica, __int64 n);

int main()
{
  __int64   wyniki [12] =
  {-15, 4000000, -345679, 88046592,
   -1, 2297645, 7867023, -19000444, 31,
    456000000000000,
    444444444444444,
   -123456789098765};

  __int64 wartosc_max;

  wartosc_max = szukaj64_max(wyniki, 12);
  printf("\nNajwiekszy element tablicy wynosi %I64d\n",
                                        wartosc_max);
```

```
  return 0;
}
```

Część programu w asemblerze (plik `szukaj64a.asm`*)*

```
public szukaj64_max

.code

szukaj64_max   PROC
     push      rbx       ; przechowanie rejestrów
     push      rsi

     mov       rbx, rcx  ; adres tablicy
     mov       rcx, rdx  ; liczba elementów tablicy
     mov       rsi, 0    ; indeks bieżący w tablicy

; w rejestrze RAX przechowywany będzie największy dotychczas
; znaleziony element tablicy - na razie przyjmujemy, że jest
; to pierwszy element tablicy
     mov       rax, [rbx + rsi*8]

; zmniejszenie o 1 liczby obiegów pętli, bo ilość porównań
; jest mniejsza o 1 od ilości elementów tablicy
     dec       rcx

ptl: inc       rsi       ; inkrementacja indeksu

; porównanie największego, dotychczas znalezionego elementu
; tablicy z elementem bieżącym
     cmp       rax, [rbx + rsi*8]
     jge       dalej; skok, gdy element bieżący jest
                    ; niewiększy od dotychczas znalezionego

; przypadek, gdy element bieżący jest większy
; od dotychczas znalezionego
     mov  rax, [rbx+rsi*8]

dalej:    loop ptl  ; organizacja pętli

; obliczona wartość maksymalna pozostaje w rejestrze RAX
; i będzie wykorzystana przez kod programu napisany w języku C

     pop  rsi
     pop  rbx
     ret
szukaj64_max   ENDP

END
```

W przypadku programów 64-bitowych asemblacja, kompilacja i linkowanie przebiega tak jak wcześniej opisano dla programów 32-bitowych. Poniżej podano zawartość plików pomocniczych potrzebnych do przeprowadzenia asemblacji, kompilacji i konsolidacji. Oczywiście można także korzystać ze środowiska zintegrowanego Microsoft Visual Studio.

Plik pomocniczy `VC64.BAT` (*napisać w jednym wierszu*):

```
"C:\Program Files (x86)\Microsoft Visual Studio 12.0\
                                    VC\bin\amd64\VCVARS64.BAT"
```

W systemie Visual Studio 2015 ścieżka ma postać `"C:\Program Files (x86)\Microsoft Visual Studio 14.0\VC\bin\VCVARS64.BAT"`
W systemie Visual Studio 2017 ścieżka ma postać `"C:\Program Files (x86)\Microsoft Visual Studio\2017\Enterprise\VC\Auxiliary\Build\VCVARS.BAT"`
W systemie Visual Studio 2019 ścieżka ma postać `"C:\Program Files (x86)\Microsoft Visual Studio\2019\Enterprise\VC\Auxiliary\Build\VCVARS64.BAT"`
W systemie Visual Studio 2022 ścieżka ma postać `"C:\Program Files\Microsoft Visual Studio\2022\Enterprise\VC\Auxiliary\Build\VCVARS32.BAT"`

Plik pomocniczy `AC64.BAT`:

```
cl -c %2.c
if errorlevel 1 goto koniec
ml64 -c -Cp -Fl %1.asm
if errorlevel 1 goto koniec
link -subsystem:console -out:%2.exe %1.obj %2.obj
:koniec
```

Jeśli w części asemblerowej wywoływane są funkcje systemowe, np. `MessageBox`, to trzeba dołączyć odpowiednią bibliotekę — wówczas plik `AC64.BAT` przyjmie postać:

```
cl -c %2.c
if errorlevel 1 goto koniec
ml64 -c -Cp -Fl %1.asm
if errorlevel 1 goto koniec
link -subsystem:console -out:%2.exe %1.obj %2.obj user32.lib
:koniec
```

Zadanie 4.5. Napisać kod asemblerowy podprogramu, przystosowanego do wywoływania z poziomu języka C w trybie 64-bitowym. Prototyp funkcji (podprogramu) ma postać:

```
__int64 suma_siedmiu_liczb (__int64  v1, __int64  v2, __int64
  v3, __int64  v4, __int64  v5, __int64  v6, __int64  v7);
```

Podana funkcja powinna obliczyć sumę wartości parametrów i zwrócić ją jako wartość funkcji. Napisać także krótki program w języku C ilustrujący sposób wywoływania podanej funkcji.

Uwagi dodatkowe dotyczące kodowania programów 64-bitowych[2]

Kodowanie programów w trybie 64-bitowym jest bardzo podobne do kodowania 32-bitowego, aczkolwiek istnieje kilka przypadków szczególnych, wymagających pewnej uwagi.

1. W trybie 64-bitowym rozkazy mogą wykonywać działania na operandach 8-, 16-, -32 i 64-bitowych. Rozkazy działające na operandach 32-bitowych dodatkowo zerują starszą część rejestru 64-bitowego, np.:

```
mov       rax, 0123456789ABCDEFH
mov       eax, 22222222H
; RAX = 0000000022222222H
```

   Rozkaz `mov eax, 22222222H` wpisał liczbę `22222222H` na 32 młodsze bity rejestru RAX i jednocześnie wyzerował 32 najstarsze bity tego rejestru.
   W przypadku działań na rejestrach 16- i 8-bitowych pozostałe bity rejestru 64-bitowego nie ulegają zmianie, np.:

```
mov       rax, 0123456789ABCDEFH
mov       al, 44H
; RAX =  0123456789ABCD44H
```

   Niejednakowe reguły dla rejestrów o różnych długościach mogą budzić wątpliwości. Obserwacje różnych sytuacji wskazują, że nieużywane bity rejestru powinny być automatycznie zerowane, ponieważ na bitach tych mogą pozostawać wyniki poprzednio wykonywanych operacji, co może stanowić potencjalną przyczynę błędów programistycznych. Zasady te nie mogły jednak wprowadzone w odniesieniu do operacji 8- i 16-bitowych ze względu na wymaganie kompatybilności wstecznej.

2. Jednobajtowe (w kontekście opcode-ów) rozkazy INC i DEC są niedostępne.

[2] Materiał zawarty w tym podrozdziale jest nadobowiązkowy.