

# Ćwiczenie 1

## Asemlacja i konsolidacja programu w assemblerze

---

### Model komputera na poziomie programowania

Komputer jest skomplikowanym urządzeniem cyfrowym, którego opis może być formułowany na różnych poziomach szczegółowości, w zależności od celu któremu ten opis ma służyć. W niniejszym opracowaniu skupimy uwagę na modelu komputera w takim kształcie, w jakim jest widoczny z poziomu programowania. Skupimy więc uwagę na tych aspektach działania komputera, które są ściśle związane ze sposobem wykonywania programu.

Prawie wszystkie współczesne komputery budowane są wg koncepcji, która została podana w roku 1945 przez matematyka amerykańskiego von Neumanna i współpracowników. Oczywiście, pierwotna koncepcja została znacznie rozszerzona i ulepszona, ale podstawowe idee nie zmieniły się. Von Neumann zaproponował ażeby program obliczeń, czyli zestaw czynności potrzebnych do rozwiązania zadania, przechowywać również w pamięci komputera, tak samo jak przechowywane są dane do obliczeń i wyniki pośrednie. W ten sposób ukształtowała się koncepcja komputera z *programem zapisanym w pamięci*, znana w literaturze technicznej jako *architektura von Neumanna*.

Do budowy współczesnych komputerów używane są elementy elektroniczne — inne rodzaje elementów (np. mechaniczne) są znacznie wolniejsze (o kilka rzędów). Ponieważ elementy elektroniczne pracują pewnie i stabilnie jako elementy dwustanowe, informacje przechowywane i przetwarzane przez komputer mają postać ciągów zerojedynkowych.

Zasadniczą i centralną część każdego komputera stanowi procesor — jego własności decydują o pracy całego komputera. Procesor steruje podstawowymi operacjami komputera, wykonuje operacje arytmetyczne i logiczne, przesyła i odbiera sygnały, adresy i dane z jednego podzespołu komputera do drugiego. Procesor pobiera kolejne instrukcje programu i dane z pamięci głównej (operacyjnej) komputera, przetwarza je i ewentualnie odsyła wyniki do pamięci. Komunikacja ze światem zewnętrznym realizowana jest za pomocą urządzeń wejścia/wyjścia.

Pamięć główna (operacyjna, RAM) składa z dużej liczby komórek (np. kilka miliardów), a każda komórka utworzona jest z pewnej liczby bitów (gdy komórkę tworzy 8 bitów, to mówimy, że *pamięć ma organizację bajtową*). Poszczególne komórki mogą zawierać dane, na których wykonywane są obliczenia, jak również mogą zawierać rozkazy (instrukcje) dla procesora.

W większości współczesnych komputerów pamięć ma organizację bajtową. Poszczególne bajty (komórki) pamięci są ponumerowane od 0 — numer komórki pamięci nazywany jest jej *adresem fizycznym*. Adres fizyczny przekazywany jest przez procesor (lub inne urządzenie) do podzespołów pamięci w celu wskazania położenia bajtu, który ma zostać odczytany lub zapisany. Zbiór wszystkich adresów fizycznych nazywa się *fizyczną przestrzenią adresową*.

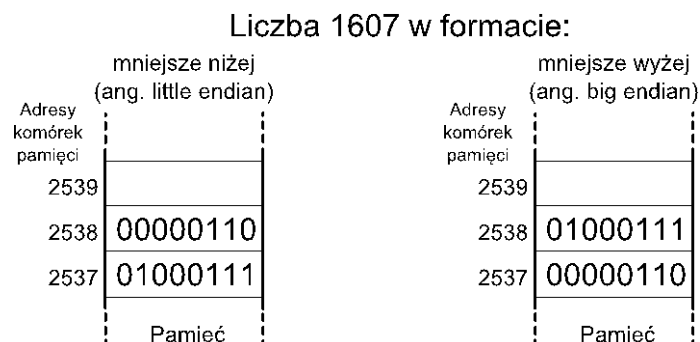
Do niedawna, w wielu współczesnych procesorach używane były najczęściej adresy 32-bitowe, co określa od razu maksymalny rozmiar zainstalowanej pamięci:  $2^{32} = 4\,294\,967\,296$  bajtów (4 GB). Obecnie rozwijane są architektury 64-bitowe, co pozwala na instalowanie pamięci o rozmiarach znacznie przekraczających 4 GB.

## Przechowywanie liczb w pamięci komputera

Liczby w naturalnym kodzie binarnym (NKB) zapisywane na 8 bitach mogą przyjmować wartości z przedziału  $<0, 255>$ . Jednak liczby występujące w programach często przekraczają 255 i muszą być zapisywane na dwóch, czterech lub na większej liczbie bajtów. W systemach komputerowych przyjęto dwa podstawowe schematy określające porządek bajtów:

- *mniej niż* (ang. little endian),<sup>1</sup>
- *mniej niż* (ang. big endian).

Porządek bajtów określany jako *mniej niż* (ang. little endian) oznacza, że bajt zawierający najmłodszą część liczby lokowany jest w pamięci na pozycji o niższym adresie niż pozostałe bajty liczby. Analogicznie, porządek *mniej niż* (ang. big endian) oznacza, że bajt zawierający najmłodszą część liczby lokowany jest w pamięci na pozycji o adresie wyższym niż pozostałe bajty liczby. W procesorach rodziny x86 stosowany jest format *mniej niż* (ang. little endian). Poniższy rysunek pokazuje przykładową reprezentację w pamięci liczby  $(11001000111)_2 = (1607)_{10}$  w obu omawianych formatach.



Podane schematy zapisywania liczb w pamięci odnoszą się także do zapisywania kodów znaków, co będzie m.in. tematem ćwiczenia laboratoryjnego nr 2.

## Architektury procesorów Intel/AMD

Współcześnie, znaczna większość używanych komputerów osobistych posiada zainstalowane procesory rodziny x86, która została zapoczątkowana w roku 1978 przez procesor Intel 8086/8088. Początkowo wytwarzano procesory o architekturze 16-bitowej, później 32-bitowe, a obecnie coraz bardziej rozpowszechniają się procesory 64-bitowe, które zaliczane są do architektury znanej jako Intel 64/AMD64. Wg tej konwencji wcześniejsze procesory 32-bitowe zaliczane do architektury Intel 32.

Architektura Intel 64/AMD64 stanowi rozszerzenie stosowanej wcześniej architektury 32-bitowej. Charakterystycznym przykładem są rejestry ogólnego przeznaczenia w procesorach.

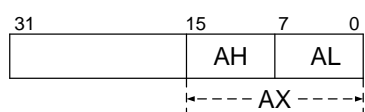
<sup>1</sup> W literaturze technicznej w języku polskim autorzy i tłumacze posługują się zazwyczaj terminami angielskimi *little/big endian*. Terminy *mniej niż*/*mniej niż* są mało rozpowszechnione.

W trakcie wykonywania obliczeń często wyniki pewnych operacji stają się danymi dla kolejnych operacji — w takim przypadku nie warto odsyłać wyników do pamięci operacyjnej, a lepiej przechować te wyniki w komórkach pamięci wewnątrz procesora. Komórki pamięci wewnątrz procesora zbudowane są w postaci rejestrów (ogólnego przeznaczenia), w których mogą być przechowywane dane i wyniki pośrednie. Z punktu widzenia procesora dostęp do

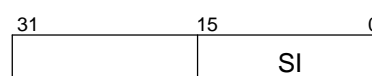
	63	31	0
RAX		EAX	
RBX		EBX	
RCX		ECX	
RDX		EDX	
RBP		EBP	
RSI		ESI	
RDI		EDI	
RSP		ESP	
R8			
R9			
R10			
R11			
R12			
R13			
R14			
R15			

danych w pamięci głównej (operacyjnej) wymaga zawsze pewnego czasu (mierzonego w dziesiątkach nanosekund), natomiast dostęp do danych zawartych w rejestrach jest praktycznie natychmiastowy. Niestety, w większości procesorów jest zaledwie kilka rejestrów ogólnego przeznaczenia, tak że nie mogą one zastępować pamięci głównej. Wprawdzie w procesorach o architekturze RISC liczba rejestrów dochodzi do kilkuset, to jednak jest to ciągle bardzo mało w porównaniu z rozmiarem pamięci głównej.

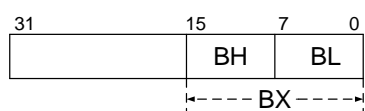
W rodzinie procesorów x86 początkowo wszystkie rejestry ogólnego przeznaczenia były 16-bitowe i oznaczone AX, BX, CX, DX, SI, DI, BP, SP. Wszystkie te rejestry w procesorze 386 i wyższych zostały rozszerzone do 32 bitów i oznaczone dodatkową literą E na początku, np. EAX, EBX, ECX, itd. W ostatnich latach rozwinięto architekturę 64-bitową, wprowadzając rejestry 64-bitowe, np. RAX, RBX, RCX, stanowiące rozszerzone wersje ww. rejestrów. Zatem młodszą część 64-bitowego rejestru RAX stanowi dotychczas używany rejestr EAX. Dodatkowo, w trybie 64-bitowym dostępne są także rejestry 64-bitowe: R8, R9, R10, R11, R12, R13, R14, R15 — tak więc w trybie 64-bitowym programista ma do dyspozycji 16 rejestrów ogólnego przeznaczenia. Na rysunku obok pokazano rejestry dostępne w trybie 64-bitowym, a rysunek na następnej stronie pokazuje strukturę rejestrów 32-bitowych. Ponieważ w komputerach osobistych pracuje nadal spora grupa procesorów 32-bitowych, w niniejszym opracowaniu skupimy się przede wszystkim na przykładach 32-bitowych. Ewentualne przejście na architekturę 64-bitową wymaga niewielkiego wysiłku.



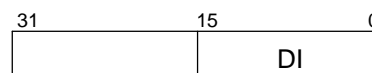
Rejestr EAX



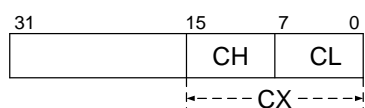
Rejestr ESI



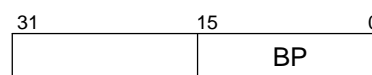
Rejestr EBX



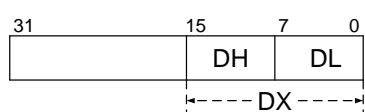
Rejestr EDI



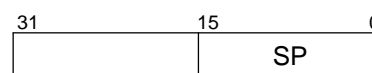
Rejestr ECX



Rejestr EBP



Rejestr EDX



Rejestr ESP

## Wykonywanie programu przez procesor

Podstawowym zadaniem procesora jest wykonywanie programów, które przechowywane są w pamięci głównej (operacyjnej). Program składa się z ciągu elementarnych poleceń, zakodowanych w sposób zrozumiały dla procesora. Poszczególne polecenia nazywane są *rozkazami* lub *instrukcjami*. Rozkazy (instrukcje) wykonują zazwyczaj proste operacje jak działania arytmetyczne (dodawanie, odejmowanie, mnożenie, dzielenie), operacje na pojedynczych bitach, przesłania z pamięci do rejestrów i odwrotnie, i wiele innych. Rozkazy zapisane są w postaci ustalonych ciągów zer i jedynek — każdej czynności odpowiada inny ciąg zer i jedynek. Postać tych ciągów jest określana na etapie projektowania procesora i jest dostępna w dokumentacji technicznej.

Tak więc rozmaite czynności, które może wykonywać procesor, zostały zakodowane w formie ustalonych kombinacji zer i jedynek, składających się na jeden lub kilka bajtów. Zakodowany ciąg bajtów umieszcza się w pamięci operacyjnej komputera, a następnie poleca się procesorowi odczytywać z pamięci i wykonywać kolejne rozkazy (instrukcje). W rezultacie procesor wykonana szereg operacji, w wyniku których uzyskamy wyniki końcowe programu.

Rozpatrzmy teraz dokładniej zasady pobierania rozkazów (instrukcji) z pamięci. Poszczególne rozkazy przekazywane do procesora mają postać jednego lub kilku bajtów o ustalonej zawartości. Przystępując do wykonywania kolejnego rozkazu procesor musi znać jego położenie w pamięci, innymi słowy musi znać adres komórki pamięci głównej (operacyjnej), gdzie znajduje się rozkaz. Często rozkaz składa się z kilku bajtów, zajmujących kolejne komórki pamięci. Jednak do pobrania wystarczy znajomość adresu tylko pierwszego bajtu rozkazu.

W prawie wszystkich współczesnych procesorach znajduje się rejestr, nazywany *wskaźnikiem instrukcji* lub *licznikiem rozkazów*, który określa położenie kolejnego rozkazu, który ma wykonać procesor. Zatem procesor, po zakończeniu wykonywania rozkazu, odczytuje liczbę zawartą we wskaźniku instrukcji i traktuje ją jako położenie w pamięci kolejnego rozkazu, który ma wykonać. Innymi słowy odczytana liczba jest adresem pamięci, pod którym znajduje się rozkaz. W tej sytuacji procesor wysyła do pamięci wyznaczony adres z jednoczesnym żądaniem odczytania jednego lub kilku bajtów pamięci znajdujących się pod wskazanym adresem. W ślad za tym pamięć operacyjna odczytuje wskazane bajty i odsyła je do procesora. Procesor traktuje otrzymane bajty jako kolejny rozkaz, który ma wykonać.

Po wykonaniu rozkazu (instrukcji) procesor powinien pobrać kolejny rozkaz, znajdujący w następnych bajtach pamięci, przylegających do aktualnie wykonywanego rozkazu. Wymaga to zwiększenia zawartości wskaźnika instrukcji, tak by wskazywał położenie następnego rozkazu. Nietrudno zauważyć, że wystarczy tylko zwiększyć zawartość wskaźnika instrukcji o liczbę bajtów aktualnie wykonywanego rozkazu. Tak też postępują prawie wszystkie procesory.

Wskaźnik instrukcji pełni więc bardzo ważną rolę w procesorze, każdorazowo wskazując mu miejsce w pamięci operacyjnej, gdzie znajduje się kolejny rozkaz do wykonania. W niektórych procesorach obliczanie adresu w pamięci jest nieco bardziej skomplikowane, aczkolwiek zasada działania wskaźnika instrukcji jest dokładnie taka sama.

## Rozkazy (instrukcje) sterujące i niesterujące

Omawiany wyżej schemat pobierania rozkazów ma jednak zasadniczą wadę. Rozkazy mogą być pobierane z pamięci w kolejności ich rozmieszczenia. Często jednak sposób wykonywania obliczeń musi być zmieniony w zależności od uzyskanych wyników w trakcie obliczeń. Przykładowo, dalszy sposób rozwiązywania równania kwadratowego zależy od wartości wyróżnika trójmianu (delt). W omawianym wyżej schemacie nie można zmieniać kolejności wykonywania rozkazów, a więc procesor działający ściśle wg tego schematu nie mógłby nawet zostać zastosowany do rozwiązania równania kwadratowego.

Przekładając ten problem na poziom instrukcji procesora można stwierdzić, że w przypadku ujemnego wyróżnika (delt) należy zmienić naturalny porządek ("po kolei") wykonywania rozkazów (instrukcji) i spowodować, by procesor pominął ("przeskoczył") dalsze obliczenia. Można to łatwo zrealizować, jeśli do wskaźnika instrukcji zostanie dodana odpowiednio duża liczba (np. dodanie liczby 143 oznacza, że procesor pominie wykonywanie instrukcji zawartych w kolejnych 143 bajtach pamięci operacyjnej). Oczywiście, takie pominięcie znacznej liczby instrukcji powinno nastąpić tylko w przypadku, gdy obliczony wyróżnik (delta) był ujemny.

Można więc zauważyć, że potrzebne są specjalne instrukcje, które w zależności od własności uzyskanego wyniku (np. czy jest ujemny) zmieniają zawartość wskaźnika instrukcji, dodając lub odejmując jakąś liczbę, albo też zmieniają zawartość wskaźnika instrukcji w konwencjonalny sposób — rozkazy (instrukcje) takie nazywane są *rozkazami sterującymi* (skokowymi).

*Rozkazy sterujące warunkowe* na ogół nie wykonują żadnych obliczeń, ale tylko sprawdzają, czy uzyskane wyniki mają oczekiwane własności. W zależności od rezultatu sprawdzenia wykonywanie programu może być kontynuowane przy zachowaniu naturalnego porządku instrukcji albo też porządek ten może być zignorowany poprzez przejście do wykonywania instrukcji znajdującej się w odległym miejscu pamięci operacyjnej. Istnieją też rozkazy sterujące, zwane *bezwartkowymi*, których jedynym zadaniem jest zmiana porządku wykonywania rozkazów (nie wykonują one żadnego sprawdzenia).

## Obserwacja operacji procesora na poziomie rozkazów

Podany tu opis podstawowych operacji procesora stanie się bardziej czytelny, jeśli uda się zaobserwować działania procesora w trakcie wykonywania pojedynczych rozkazów. W tym celu spróbujemy napisać sekwencję kilku rozkazów (instrukcji) procesora wykonujących proste obliczenia na liczbach całkowitych. Jednak procesor rozumie tylko instrukcje zapisane w języku maszynowym w postaci ciągów zer i jedynek. Wprawdzie zapisanie takiego ciągu zerojedynkowego jest możliwe, ale wymaga to dokładnej znajomości formatów rozkazów, a przy tym jest bardzo żmudne i podatne na błędy.

Wymienione tu trudności eliminuje się poprzez zapisanie programu (dalej na poziomie pojedynczych rozkazów) w postaci symbolicznej, w której poszczególne rozkazy reprezentowane są przez zrozumiałe skróty literowe, w której występują jawnie podane nazwy rejestrów procesora (a nie w postaci ciągów zer i jedynek) i wreszcie istnieje możliwość podawania wartości liczbowych w postaci liczb dziesiętnych lub szesnastkowych. Oczywiście zapis w języku symbolicznym wymaga przekształcenia na kod maszynowy (zerojedynkowy), zrozumiały przez procesor. Zamiana taka jest zazwyczaj wykonywana przez program nazywany *assemblerem*. Assembler odczytuje kolejne wiersze programu zapisanego w postaci symbolicznej i zamienia je na równoważne ciągi zer i jedynek. Termin

*assembler* oznacza także język programowania, w którym rozkazy i dane zapisywane są w postaci symbolicznej.

Poruszony tu problem kodowania rozkazów można więc dość prosto rozwiązać posługując się assemblerem. Jednak celem naszym działań jest obserwacja wykonywania rozkazów przez procesor. Niestety, nie mamy możliwości bezpośredniej obserwacji zawartości rejestrów procesora czy komórek pamięci (aczkolwiek możliwość taką miały procesory wytwarzane pół wieku temu). I tu także przychodzi z pomocą oprogramowanie. Współczesne systemy programowania oferują m.in. programy narzędziowe pozwalające na wykonywanie programów w sposób kontrolowany, w którym możliwe jest zatrzymywanie programu w dowolnym miejscu i obserwacja uzyskanych dotychczas wyników. Tego rodzaju program, nazywany *debuggerem* omawiany jest na dalszych stronach niniejszego opracowania.

## Kodowanie rozkazów w assemblerze

W początkowym okresie rozwoju informatyki assembly stanowiły często podstawowy język programowania, na bazie którego tworzono nawet złożone systemy informatyczne. Obecnie assembler stosowany jest przede wszystkim do tworzenia modułów oprogramowania, działających jako interfejsy programowe. Należy tu wymienić moduły służące do bezpośredniego sterowania urządzeń i podzespołów komputera. W assemblerze koduje się też te fragmenty oprogramowania, które w decydujący sposób określają szybkość działania programu. Wymienione zastosowania wskazują, że moduły napisane w assemblerze występują zazwyczaj w połączeniu z modułami napisanymi w innych językach programowania.

Dla komputerów PC pracujących w systemie Windows używany jest często assembler MASM firmy Microsoft, którego najnowsza wersja oznaczona jest numerem 14.0. W sieci Internet dostępnych jest wiele innych assemblerów, spośród których najbardziej znany jest assembler NASM, udostępniany w wersjach dla systemu Windows i Linux.

Na poziomie rozkazów procesora, operacja przesłania zawartości komórki pamięci do rejestru procesora realizowana przez rozkaz oznaczony skrótem literowym (mnemonikiem) **MOV**. Rozkaz ten ma dwa argumenty: pierwszy argument określa cel, czyli "*dokąd przesłać*", drugi zaś określa źródło, czyli "*skąd przesłać*" lub "*co przesłać*":

**MOV**    dokąd            skąd (lub co)  
         przesłać        przesłać

W omawianym dalej fragmencie programu mnemonik operacji przesłania zapisywany jest małymi literami (**mov**), podczas w opisach używa się zwykle wielkich liter (**MOV**) — obie formy są równoważne.

Rozkaz (instrukcja) przesłania **MOV** jest jednym z najprostszych w grupie rozkazów niesterujących — jego zadaniem jest skopiowanie zawartości podanej komórki pamięci lub rejestru do innego rejestru. W programach napisanych w assemblerze dla procesorów rodziny x86 rozkaz przesłania **MOV** ma dwa argumenty rozdzielone przecinkami. W wielu rozkazach drugim argumentem może być liczba, która ma zostać przesłana do pierwszego argumentu — tego rodzaju rozkazy określa się jako *przesłania z argumentami bezpośrednimi*., np.

**MOV CX, 7305**

Omawiane tu rozkazy (instrukcje) zaliczane są do klasy rozkazów *niesterujących*, to znaczy takich, które nie zmieniają naturalnego porządku wykonywania rozkazów. Zatem po wykonaniu takiego rozkazu procesor rozpoczyna wykonywanie kolejnego rozkazu, przylegającego w pamięci do rozkazu właśnie zakończonego.

Rozkazy *niesterujące* wykonują podstawowe operacje jak przesłania, działania arytmetyczne na liczbach (dodawanie, odejmowanie, mnożenie, dzielenie), operacje logiczne na bitach (suma logiczna, iloczyn logiczny), operacje przesunięcia bitów w lewo i w prawo, i wiele innych. Argumenty rozkazów wykonujących operacje dodawania ADD i odejmowania SUB zapisuje się podobnie jak argumenty rozkazu MOV

**ADD**    dodajna    ,    dodajnik

**SUB**    odjemna    ,    odjemnik

↑  
wynik wpisywany jest do  
obiektu wskazanego przez  
pierwszy argument

Podane tu rozkazy dodawania i odejmowania mogą być stosowane zarówno do liczb bez znaku, jak i liczb ze znakiem (w kodzie U2). W identyczny sposób podaje się argumenty dla innych rozkazów wykonujących operacje dwuargumentowe, np. XOR. Ogólnie rozkaz taki wykonuje operację na dwóch wartościach wskazanych przez pierwszy i drugi operand, a wynik wpisywany jest do pierwszego operandu. Zatem rozkaz

„operacja”                      cel, źródło

wykonuje działanie

cel ← cel „operacja” źródło

Operandy *cel* i *źródło* mogą wskazywać na rejestry lub lokacje pamięci, jednak tylko jeden operand może wskazywać lokację pamięci. Wyjątkowo spotyka się asemblery (np. asembler w wersji AT&T), w których wynik operacji wpisywany jest do drugiego operandu (przesłania zapisywane są w postaci *skąd, dokąd*).

Nieco inaczej zapisuje się rozkaz mnożenia MUL (dla liczb bez znaku). W przypadku tego rozkazu konstruktorzy procesora przyjęli, że mnożna znajduje się zawsze w ustalonym rejestrze procesora: w AL – jeśli mnożone są liczby 8-bitowe, w AX – jeśli mnożone są liczby 16-bitowe, w EAX – jeśli mnożone są liczby 32-bitowe, w RAX – jeśli mnożone są liczby 64-bitowe. Z tego powodu podaje się tylko jeden argument — mnożnik. Rozmiar mnożnika (8, 16, 32 lub 64 bity) określa jednocześnie rozmiar mnożnej.

**MUL**            mnożnik

Wynik mnożenia wpisywany jest zawsze do ustalonych rejestrów: w przypadku mnożenia dwóch liczb 8-bitowych, 16-bitowy wynik mnożenia wpisywany jest do rejestru AX, analogicznie przy mnożeniu liczb 16-bitowych wynik wpisywany jest do rejestrów DX:AX, dla liczb 32-bitowych do EDX:EAX, dla liczb 64-bitowych do RDX:RAX. Występujące tu pary nazw rejestrów rozdzielone znakiem dwukropka oznaczają złożenie dwóch rejestrów, które w niektórych operacjach traktowane są jako pojedynczy rejestr, np. EDX:EAX oznacza rejestr 64-bitowy, w którym starsze 32 bity są tożsame z rejestrem EDX, a młodsze 32 bity są tożsame z rejestrem EAX.

## Przykładowy program assemblerowy

Autorzy 32-bitowego interfejsu programowego systemu Windows, kierując się zamiarem implementacji systemu operacyjnego na różne typy procesorów, zdefiniowali operacje wejścia/wyjścia na poziomie języka C. Stąd wyprowadzenie znaków na ekran wymaga wywołania funkcji języka C na poziomie assemblera.

Poniżej podano krótki program przykładowy w assemblerze wraz z opisem sposobu translacji i wykonania. W pierwszej chwili, dla mniej zorientowanego czytelnika, podany niżej program może wydawać się dość skomplikowany. Mimo tego, czytelnik powinien dość uważnie go przeanalizować, korzystając z podanych dalej objaśnień. Objasnienia mają charakter wstępny i mają przede wszystkim na celu przedstawienie podstawowych elementów programu. Wykonanie podanego niżej programu powoduje wyświetlenie na ekranie komputera tekstu:

**Nazywam sie ...**

**Moj pierwszy 32-bitowy program assemblerowy dziala juz poprawnie!**

W wyświetlanym tekście używane są wyłącznie litery alfabetu łacińskiego, co powoduje, że niektóre wyrazy są zniekształcone. Problemy kodowania znaków, w tym kodowania liter alfabetu języka polskiego, rozpatrywane będą szczegółowo w ramach ćwiczenia nr 2.

```
; program przykładowy (wersja 32-bitowa)
.686
.model flat
extern _ExitProcess@4 : PROC
extern __write         : PROC ; (dwa znaki podkreślenia)
public _main

.data
tekst      db 10, 'Nazywam sie . . . ' , 10
           db 'Moj pierwszy 32-bitowy program '
           db 'assemblerowy dziala juz poprawnie!', 10

.code
_main PROC
    mov     ecx, 85 ; liczba znaków wyświetlanego tekstu

; wywołanie funkcji "write" z biblioteki języka C
    push    ecx ; liczba znaków wyświetlanego tekstu
    push    dword PTR OFFSET tekst ; położenie obszaru
                                ; ze znakami
    push    dword PTR 1 ; uchwyt urządzenia wyjściowego
    call    __write ; wyświetlenie znaków
                                ; (dwa znaki podkreślenia _ )
    add     esp, 12 ; usunięcie parametrów ze stosu

; zakończenie wykonywania programu
    push    dword PTR 0 ; kod powrotu programu
    call    _ExitProcess@4
_main ENDP
END
```



Zauważmy, że w podanym programie występują dwie charakterystyczne części, z których pierwsza zaczynająca się od dyrektywy `.data` zawiera dane programu, a druga zaczynająca się od dyrektywy `.code` zawiera rozkazy (instrukcje) programu. Cały program kończy dyrektywa `END`.

W każdej części występują różne typy wierszy programu, a wśród nich rozkazy procesora (ściślej: mnemoniki rozkazów), dyrektywy, etykiety i komentarze. Nietrudno zauważyć, że komentarze poprzedzone są średnikiem. Zwróćmy uwagę na wiersz

```
tekst      db      10, 'Nazywam sie . . . ' , 10
```

umieszczony po `.data`. Mamy tutaj dyrektywę `db` (ang. define byte) stanowiącą opis bajtów. Podany wiersz stanowi polecenie zarezerwowania w programie kilkunastu bajtów, przy czym pierwszy z nich będzie zawierał liczbę 10. Liczbę 10 można traktować jako polecenie, by dalszy ciąg tekstu był wyświetlany (drukowany) od nowego wiersza.<sup>2</sup> Następne bajty zawierać będą kody ASCII znaków tworzących wyrazy `Nazywam sie`, itd.

Spróbujmy teraz przeanalizować działanie programu. Jako pierwszy zostanie wykonany rozkaz wskazany przez etykietę `_main`:

```
mov        ecx, 85
```

Rozkaz ten powoduje wpisanie do rejestru `ECX` liczby 85, która w omawianym programie określa liczbę znaków wyświetlanego tekstu. Jeśli podczas redagowania programu liczba znaków tekstu zostanie zwiększona, to trzeba też odpowiednio zwiększyć podaną wartość.

Następne rozkazy przygotowują parametry potrzebne do wywołania funkcji wyświetlającej znaki na ekranie. Można przypuszczać, że wyświetlenie znaków na ekranie wymaga po prostu wpisania odpowiednich kodów do pamięci umieszczonej na karcie graficznej komputera. Rzeczywiście, w pierwszych komputerach PC metoda taka była powszechnie stosowana, jednak rozwój sprzętu i oprogramowania spowodował, że zapis taki jest obecnie dość kłopotliwy, i co najważniejsze może powodować zakłócenia w pracy systemu. Dlatego też współczesne systemy operacyjny nie zezwalają na bezpośrednie sterowanie urządzeniami komputera (np. kartą graficzną), ale wymagają obowiązkowego pośrednictwa systemu operacyjnego. Próba ominięcia tego pośrednictwa jest natychmiast sygnalizowana przez procesor i powoduje zazwyczaj zakończenie wykonywania programu.

Zatem wyświetlenie znaków na ekranie wymaga użycia odpowiedniej funkcji systemowej, która udostępniana jest przez system operacyjny. Funkcje systemu Windows opisane są na poziomie interfejsu języka C, czyli zakłada się (choć nie jest to obowiązkowe), że będą one wywoływane z poziomu języka C. W rezultacie, wywołując funkcję usługową systemu operacyjnego w kodzie assemblerowym musimy przeprowadzić to wywołanie, tak jak gdyby zostało ono wykonane na poziomie języka C.

Musimy teraz odwołać się do znanej z języka C funkcji `write`, która zapisuje dane znajdujące się w obszarze pamięci (określonym przez drugi parametr) do pliku lub urządzenia. W naszym przypadku dane, w postaci znaków w kodzie ASCII, przesyłamy na ekran, czyli do urządzenia, które identyfikowane jest przez pierwszy parametr (uchwyt). Wywołanie funkcji `write` w języku C ma postać (zob. np. opis podany w podręczniku Kernighana i Ritchie „Język C”)

```
write (uchwyt, adres_obszaru_danych, liczba_znaków);
```

<sup>2</sup> W systemie Unix, a później w Linuksie znakowi nowego wiersza (LF – Line Feed) przypisano wartość 10. W systemie Windows (wcześniej w systemie DOS) znak nowego wiersza tworzą dwa bajty o wartościach 13,10.

Funkcja ta ma trzy parametry, i zgodnie ze przyjętymi standardami powinny one zostać zapisane w specjalnym segmencie danych dynamicznych, który nazywany jest *stosem*. Zapisywanie danych na stosie wykonywane jest przez rozkazy `push`. Dodajmy, że parametry funkcji `write` zapisywane są na stos w kolejności od prawej do lewej, a więc najpierw zostanie zapisana liczba znaków, potem adres obszaru danych i wreszcie *uchwyt*, który w tym przypadku ma wartość 1. Po załadowaniu parametrów na stos można wywołać funkcję biblioteczną języka C:

```
call    __write
```

W wyniku realizacji tej funkcji na ekranie zostanie wyświetlony tekst podany w obszarze danych programu. Po wyświetleniu znaków parametry zapisane wcześniej na stosie są już niepotrzebne i muszą zostać usunięte za pomocą rozkazu `add esp, 12`.

Wywołanie funkcji `ExitProcess` kończy wykonywanie programu i oddaje sterowanie do systemu operacyjnego.

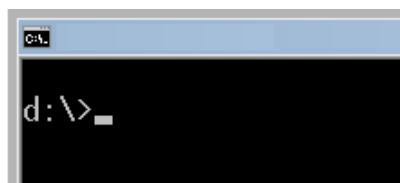
## Edycja i uruchamianie programów z wykorzystaniem asemblera 32-bitowego i konsolidatora zewnętrznego

### Tworzenie pliku źródłowego

Komputery zainstalowane w laboratoriach komputerowych na Wydziale ETI PG pracują w systemie MS Windows. W niektórych laboratoriach zainstalowany jest także system Linux.

W przypadku systemu Windows, po uruchomieniu komputera lub po zakończeniu pracy przez poprzedniego użytkownika należy odczekać aż pojawi się komunikat **Naciśnij Ctrl Alt Del**. Następnie wystarczy tylko kliknąć na ikonę *student*.

W tym podrozdziale pokażemy sposób translacji programu asemblerowego za pomocą asemblera i konsolidatora (linkera), które nie są zintegrowane z edytorem. Z tego względu wygodnie jest przeprowadzać translację w okienku konsoli (dawniej nazywanym oknem DOSowym). Otwarcie okienka konsoli następuje zazwyczaj poprzez dwukrotne kliknięcie na ikonę *Cmd* (*Wiersz poleceń*) umieszczoną na pulpicie. Jeśli ikona nie występuje, to wystarczy tylko wpisać polecenie `cmd` do pola nad przyciskiem startu (lub do pola *Uruchom*). Po otwarciu okna konsoli należy wpisać `d:` i nacisnąć klawisz Enter. Poniższy rysunek przedstawia górny lewy róg okienka konsoli bezpośrednio po wykonaniu opisanych działań.



Wskazane jest by każdy użytkownik utworzył podkatalog roboczy na tym dysku wybierając, np. imię użytkownika *Zofia*. Przypomnijmy, że nowy katalog tworzy się za pomocą polecenia `mkdir`, np. `mkdir Zofia`. Polecenie `cd Zofia` powoduje zmianę bieżącego katalogu na *Zofia*.

```

C:\. Wiersz polecenia
d:\>mkdir Zofia
d:\>cd Zofia
d:\Zofia>_

```

W nazwach katalogów i plików należy unikać stosowania spacji (odstępu) i liter specyficznych dla alfabetu języka polskiego (ą, ć, ę, ...). W praktyce, tworząc nazwy plików i katalogów posługujemy się znakiem podkreślenia `_`, który zastępuje spację.

Dalsze operacje w okienku konsoli należy realizować w utworzonym katalogu. Tak więc znak zachęty (ang. *prompt*) wyświetlany na ekranie przez cały czas pracy będzie miał postać np.

Wszelkie pliki tworzone w trakcie zajęć laboratoryjnych, w tym pliki źródłowe programów jak i pliki wytwarzane przez asemblery i kompilatory powinny być lokowane w wybranym katalogu na dysku **D**:

d:\Zofia>

Nie należy zmieniać bieżącego napędu dyskowego na inny. Przed zakończeniem zajęć pliki źródłowe należy skopiować na inny nośnik, zaś wcześniej utworzony podkatalog powinien zostać skasowany.

W trakcie uruchamiania programu wielokrotnie wprowadza się te same polecenia — wszystkie wydane wcześniej polecenia można przeglądać posługując się klawiszami strzałek i ↓. Naciśnięcie klawisza F7 powoduje wyświetlenie listy wprowadzonych poleceń, co ułatwia wyszukanie potrzebnego polecenia. Podobny mechanizm dostępny jest w systemie Linux.

```

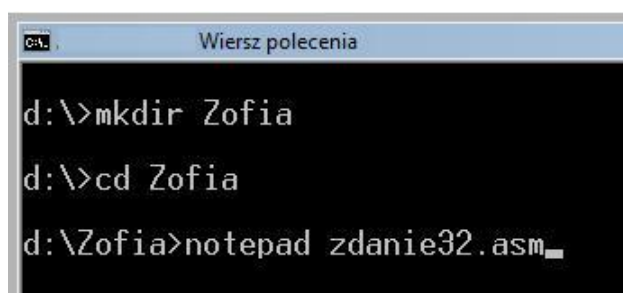
C:\. Wiersz polecenia
d:\>mkdir Zofia
d:\>cd Zofia
d:\Zofia>_

```

1: mkdir Zofia  
 2: cd Zofia

Program źródłowy można napisać korzystając z dowolnego edytora, który nie wprowadza znaków formatujących. Może to być więc Notatnik (ang. Notepad) czy np. Notepad++ (słowa kluczowe języka programowania wyświetla w innym kolorze), ale Word czy Write nie jest odpowiedni. Istotne jest także by edytor wyświetlał numer wiersza — własność tę ma m.in. Notatnik\*. Nazwa pliku zawierającego kod źródłowy powinna posiadać rozszerzenie **ASM**. Praktyczne jest wywoływanie edytora z okienka konsoli z nazwą pliku podaną w linii zlecenia, np.

\* W celu aktywowania wyświetlania numerów wierszy w Notatniku należy wybrać opcję Widok / Pasek stanu. Opcja ta może być niedostępna, jeśli wcześniej ustawiono opcję Format / Zawijanie wierszy.



```

Wiersz polecenia

d:\>mkdir Zofia

d:\>cd Zofia

d:\Zofia>notepad zdanie32.asm

```

Dla wygody dalszego opisu przyjmijmy, że program źródłowy znajduje się w pliku `zdanie32.asm` (kod programu przykładowego podany jest na str. 7–8).

Po utworzeniu pliku źródłowego należy poddać go asemblacji i konsolidacji (linkowaniu). W wyniku asemblacji uzyskuje się plik z rozszerzeniem `.OBJ` (o ile program nie zawierał błędów formalnych). Kod zawarty w pliku `.OBJ` (tzw. kod półskompilowany) zawiera już instrukcje programu zakodowane w języku maszyny, ale nie jest jeszcze całkowicie przygotowany do wykonywania przez procesor. Ostateczne przygotowanie kodu, a także włączenie programów bibliotecznych czy innych programów, jeśli jest to konieczne, następuje w fazie zwanej *konsolidacją* lub *linkowaniem*. Wykonuje to program zwany konsolidatorem lub linkerem (np. `link`), w wyniku czego powstaje plik z rozszerzeniem `.EXE`, zawierający program gotowy do wykonania.

W laboratoriach komputerowych MKZL dostępne są między innymi 32-bitowe asemblery i konsolidatory (linkery) firmy Microsoft: `ML.EXE` (assembler) i `LINK.EXE` (konsolidator).

### Asemblacja i konsolidacja za pomocą oprogramowania firmy Microsoft

Potrzebne oprogramowanie firmy Microsoft zainstalowane jest w laboratoriach MKZL, natomiast studenci mogą je uzyskać w ramach programu *Azure Dev Tools for Teaching*. (bliższe informacje podane są na stronie internetowej Wydziału ETI PG - <https://eti.pg.edu.pl/studenci/dla-studentow/microsoft>).

Najpierw należy skonfigurować ścieżki dostępu do assemblera i konsolidatora. W tym celu w okienku konsoli należy wywołać plik wsadowy `VCVARS32.BAT`. Ponieważ ścieżka dostępu do tego pliku jest dość długa<sup>3 4 5</sup>, a niekiedy trzeba ją kilkakrotnie wpisywać, warto utworzyć plik wsadowy `vc32.bat` zawierający poniższy wiersz (wpisać w jednym wierszu, nie pomijać znaków cudzysłowu):

```
"C:\Program Files\Microsoft Visual
Studio\2022\Enterprise\VC\Auxiliary\Build\VCVARS32.BAT"
```

Najłatwiej zrealizować to wywołując program Notatnik (`notepad`) bezpośrednio z poziomu okienka konsoli:

<sup>3</sup> W systemie Visual Studio 2015 ścieżka ma postać `"C:\Program Files (x86)\Microsoft Visual Studio 14.0\VC\bin\VCVARS32.BAT"`

<sup>4</sup> W systemie Visual Studio 2017 ścieżka ma postać `"C:\Program Files (x86)\Microsoft Visual Studio\2017\Enterprise\VC\Auxiliary\Build\VCVARS32.BAT"`

<sup>5</sup> W systemie Visual Studio 2019 ścieżka ma postać `"C:\Program Files (x86)\Microsoft Visual Studio\2019\Enterprise\VC\Auxiliary\Build\VCVARS32.BAT"`

```
notepad vc32.bat
```

Po przygotowaniu pliku `vc32.bat` wpisanie do okienka konsoli tekstu `vc32` i naciśnięcie klawisza **Enter** spowoduje wykonanie podanego polecenia.

Od tej chwili, w dalszych działaniach, wywołanie asemblera `MASM (ml.exe)` lub konsolidatora `LINK (link.exe)` nie wymaga podawania ścieżek dostępu. Ustalenie to dotyczy tylko okienka konsoli, w którym wywołano plik `VCVARS32.BAT` (lub `vc32.BAT`). Jeśli w trakcie pracy zostanie otwarte inne okienko konsoli, to niezbędne jest ponownym wywołanie pliku `VCVARS32.BAT` (lub `vc32.BAT`) w nowym okienku.

Przyjmujemy, że program znajduje się w pliku `zdanie32.asm`. Polecenie asemblacji ma postać:

```
ml -c -Cp -coff -Fl zdanie32.asm
```

*Uwaga: powyższy wiersz należy wpisać do okienka konsoli z klawiatury, i nie należy kopiować go do okienka konsoli poprzez operacje **Kopiuj / Wklej** (Ctrl C / Ctrl V). Występujący tu znak minus "-" kopiowany jest jako znak specjalny, co powoduje sygnalizowanie błędów przez asembler. Niniejsza uwaga odnosi się także do dalej opisanych poleceń, w których występuje znak minus "-", a także znaki cudzysłowu i apostrofu.*

Jeśli w trakcie asemblacji nie zostały wykryte błędy składniowe, to asembler utworzy plik `zdanie32.obj` zawierający kod programu w języku pośrednim. Ponadto zostanie utworzony plik `zdanie32.lst` zawierający raport z przebiegu asemblacji. W raporcie tym obok przedrukowanego tekstu źródłowego programu podane jest położenie rozkazów i danych w programie i ich reprezentacja szesnastkowa. Na końcu raportu podane jest zestawienie użytych symboli i przyporządkowanych im wartości. Z kolei w celu przeprowadzenia konsolidacji (linkowania) należy napisać polecenie:

```
link -subsystem:console -out:zdanie32.exe zdanie32.obj libcmt.lib
```

W rezultacie powstanie plik `zdanie32.exe` zawierający program gotowy do wykonania.

Niekiedy uruchamiany program zawiera kilka błędów składniowych, co wymaga wielokrotnego powtarzania opisanych operacji. Z tego powodu można napisać prosty plik wsadowy (z rozszerzeniem `.bat`), który automatyzuje wykonywanie ww. czynności. Przykładowy plik `a32.bat` może mieć postać (zob. też podaną wyżej uwagę dot. znaku minus „-“):

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

Korzystając z tego pliku, przetłumaczenie pliku źródłowego `zdanie32.asm` wymaga wprowadzenia polecenia

```
a32 zdanie32
```

Instrukcja `if errorlevel 1 goto koniec` testuje kod powrotu asemblera i jeśli jest on większy od zera (tj. gdy wystąpiły błędy składniowe), to operacja konsolidacji zostaje pominięta.

Jeśli aseblacja i linkowanie zostaną wykonane poprawnie, to powstanie plik `zdanie32.exe` zawierający kod programu gotowy do wykonania. W celu wykonania programu wystarczy wpisać tekst `zdanie32` do okienka konsoli i nacisnąć klawisz Enter (nie trzeba podawać rozszerzenia `.exe`).

## Usuwanie błędów formalnych

Często program poddany aseblacji wykazuje błędy, podawane przez assembler. Usunięcie takich błędów, w przeciwieństwie do opisanych dalej błędów wewnętrznych zakodowanego algorytmu, nie przedstawia na ogół większych trudności. Wystarczy tylko odnaleźć w programie źródłowym błędny wiersz i dokonać odpowiedniej poprawki. Przykładowo, jeśli w trakcie aseblacji sygnalizowane były błędy na ekranie:

```
zdanie.asm(17) : error A2070: invalid instruction operands
zdanie.asm(24) : error A2005: symbol redefinition : ptl
```

to ich odnalezienie nie przedstawia większych trudności. W wierszu 17 występuje instrukcja, w której pojawiła się niezgodność typów operandów — po odszukaniu w pliku źródłowym okazało się, że instrukcja ta ma postać:

```
add    bh, ax
```

Okazało, że autor programu planował pierwotnie dodać do rejestru BH liczbę 8-bitową przechowywaną w rejestrze AL. Po wprowadzeniu zmian wiersz przyjął postać:

```
add    bh, al
```

Podobnie, przyczyną następnego błędu (wiersz 24) było powtórzenie definicji etykiety `ptl`.

W fazie konsolidacji programu (linkowania) pojawia się czasami błąd `unresolved external symbol` — opis tego błędu podany jest na str. 17.

## Typowe błędy kodowania programów w języku aseblera

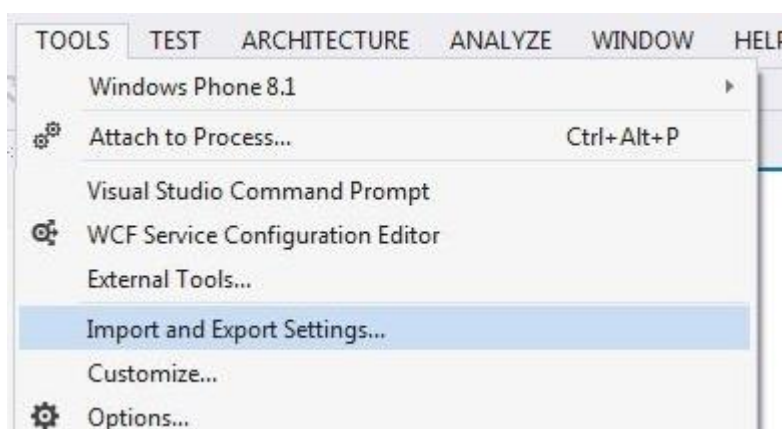
1. Kolejność sekwencji rozkazów **POP** musi być odwrotna w stosunku do sekwencji rozkazów **PUSH**. Liczba wykonanych rozkazów **PUSH** i **POP** musi być jednakowa.
2. Należy przechowywać zawartości rejestrów przy wywoływaniu podprogramów, chyba że rejestry przechowywane są wewnątrz podprogramu. Rejestr **EBP** często pełni rolę pomocniczego wskaźnika stosu i trzeba zachować ostrożność, jeśli stosowany jest jako zwykły rejestr do przechowywania wyników pośrednich.
3. Należy pamiętać o strukturze rejestrów, np. zmiana rejestru **BH** powoduje jednocześnie zmianę rejestru **EBX**, a wyzerowanie rejestru **ESI** powoduje także wyzerowanie rejestru **SI**.
4. W wyrażeniach języka assembler należy precyzyjnie odróżniać adresy i wartości zmiennych.
5. Jeśli do tworzenia programu źródłowego używany jest *Notatnik* (notepad), to utworzony plik z programem powinien być zapisany przy użyciu kodowania ANSI (opcja Plik / Zapisz jako). Assembler firmy Microsoft (`ml.exe`) w wersji 12.0 nie akceptuje

kodu źródłowego w formacie UTF-16 lub UTF-8. Warto dodać, że kompilator języka C/C++ (wersja 18.0) akceptuje wszystkie wymienione formaty (także z bajtami BOM, zob. ćw. 2).

6. Jeśli kod assemblerowy jest łączony z kodem w języku C++, to funkcję zdefiniowaną w assemblerze należy deklarować jako `extern "C"`. Eliminuje to problem wynikający z automatycznych zmian nazw funkcji w celu zakodowania informacji o parametrach (ang. name mangling). Często wymagane jest poprzedzenie nazwy funkcji w kodzie assemblerowym znakiem podkreślenia.
7. W kodzie podprogramu musi wystąpić co najmniej jeden rozkaz `RET`.
8. W systemach 64-bitowych przed wykonaniem rozkazu skoku do podprogramu wskaźnik stosu musi wskazywać adres podzielny przez 16. Należy także pamiętać o rezerwacji obszaru *shadow space*. Konwencje wywoływania funkcji w 64-bitowych wersjach systemu Windows i Linux nie są jednakowe (zob. opis ćw. 4).
9. W operacjach zmiennoprzecinkowych przed zakończeniem podprogramu należy oczyścić rejestry tworzące stos koprocatora, z wyjątkiem ST(0), jeśli służy on do przekazywania wyniku funkcji.

## Edycja i uruchamianie programów w standardzie 32-bitowym w środowisku Microsoft Visual Studio

1. Przy pierwszym uruchomieniu MS Visual Studio należy podać typ projektu C/C++
2. Przed uruchomieniem aplikacji może być konieczne odtworzenie standardowej konfiguracji systemu Visual Studio. Niektórzy użytkownicy Visual Studio dostosowują konfigurację do własnych potrzeb, co powoduje znaczne utrudnienia dla użytkowników, którzy później uruchamiają własne aplikacje. Konfigurację standardową można przywrócić wykonując niżej opisane czynności.  
Najpierw wybieramy opcję Tools / Import and Export Settings,.



W ślad za tym na ekranie zostanie wyświetlone okno dialogowe, w którym należy zaznaczyć opcję **Reset all settings** i nacisnąć przycisk **Next**.

☒ **Reset all settings**

Reset all environment settings to one of the default collections of settings.

W kolejnym oknie zaznaczamy opcję „No, just reset settings, overwriting my current settings”.

☒ **No, just reset settings, overwriting my current settings**

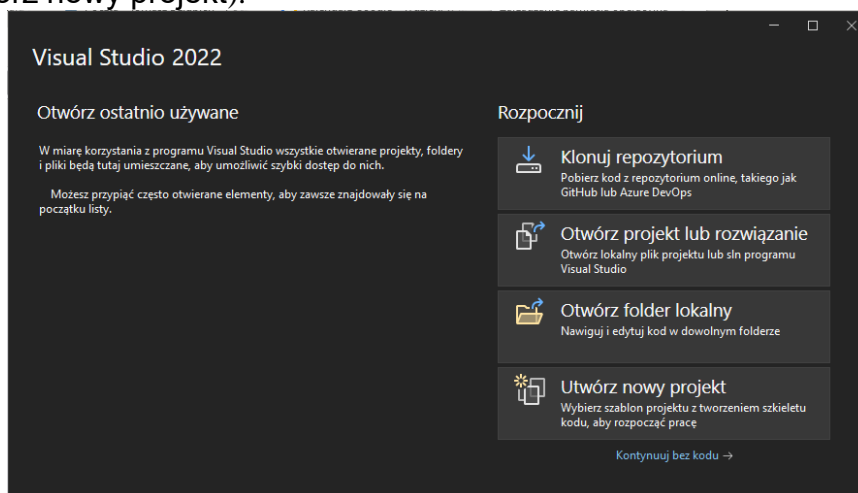
Po ponownym naciśnięciu przycisku Next pojawia niżej pokazane okno, w którym należy zaznaczyć opcję Visual C++ Development Settings.



Potem naciskamy kolejno przyciski Finish i Close.

3. Po uruchomieniu MS Visual Studio należy wybrać opcje: File / New / Project

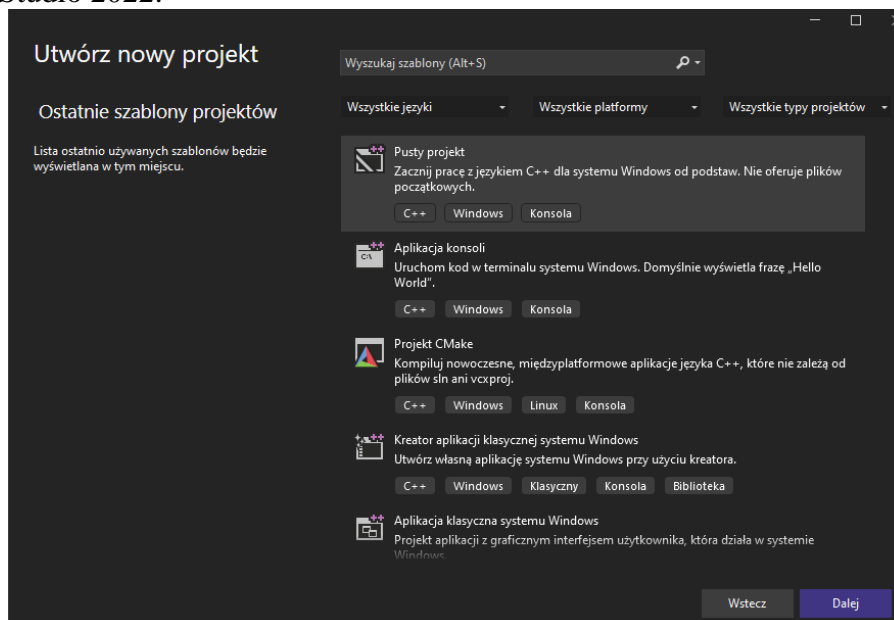
. W Visual Studio 2022 okno Projektu wygląda następująco (opcja Create a new project/Utwórz nowy projekt):





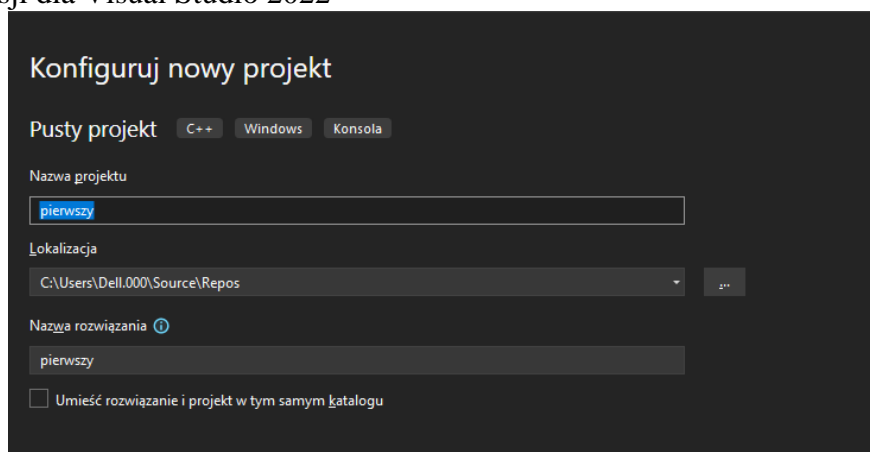
4. W oknie nowego projektu (zob. rys.) określamy najpierw typ projektu poprzez rozwinięcie opcji Visual C++ (z lewej strony okna). Następnie wybieramy opcje General i Empty Project (Pusty projekt) w środkowym oknie).

Dla Visual Studio 2022:

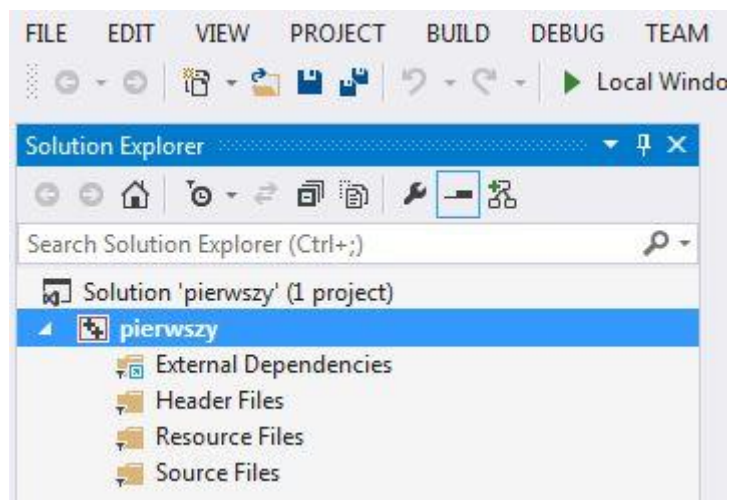


5. Do pola Name (Nazwa) wpisujemy nazwę programu (tu: pierwszy) i naciskamy OK. W polu Location (Lokalizacja) powinna znajdować się ścieżka D:\ Znacznik Create directory for solution (Umieść rozwiązanie i projekt w tym samym katalogu) należy ustawić w stanie nieaktywnym.

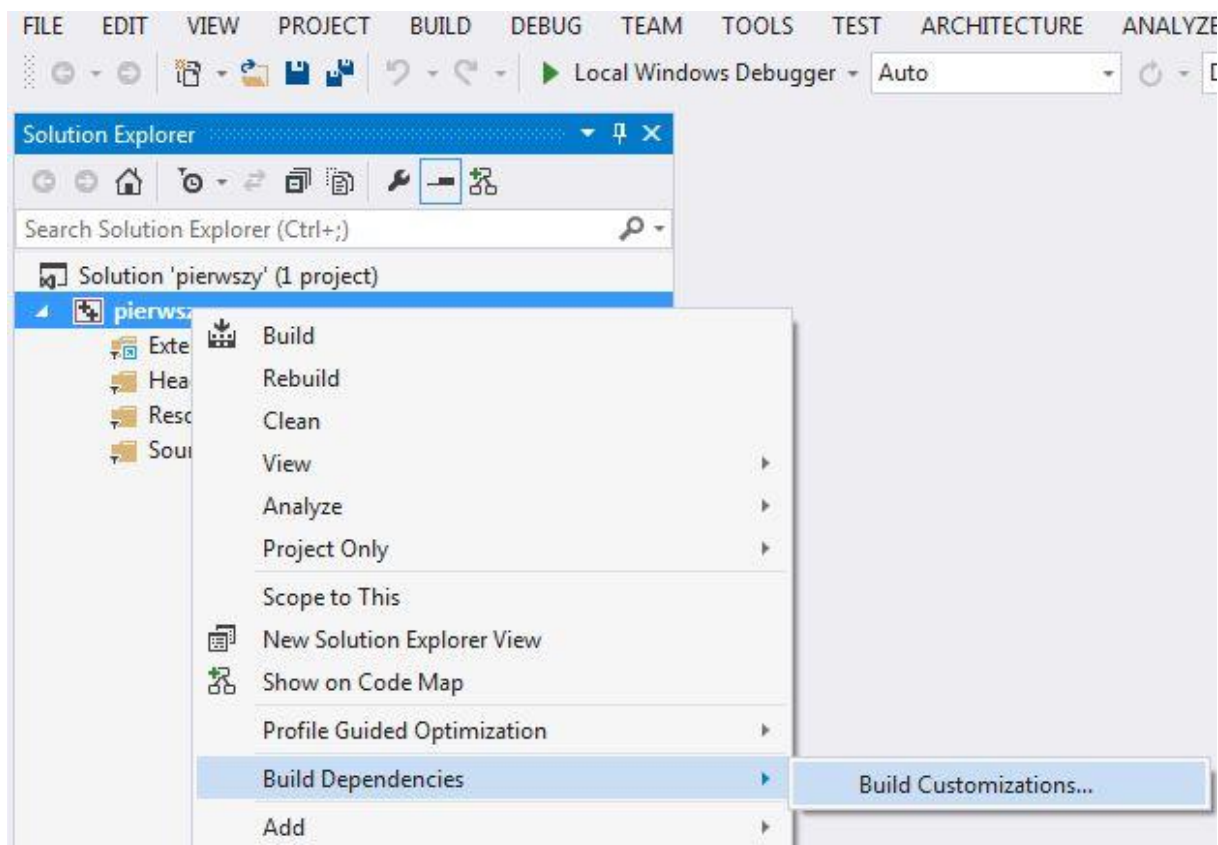
Okno w wersji dla Visual Studio 2022



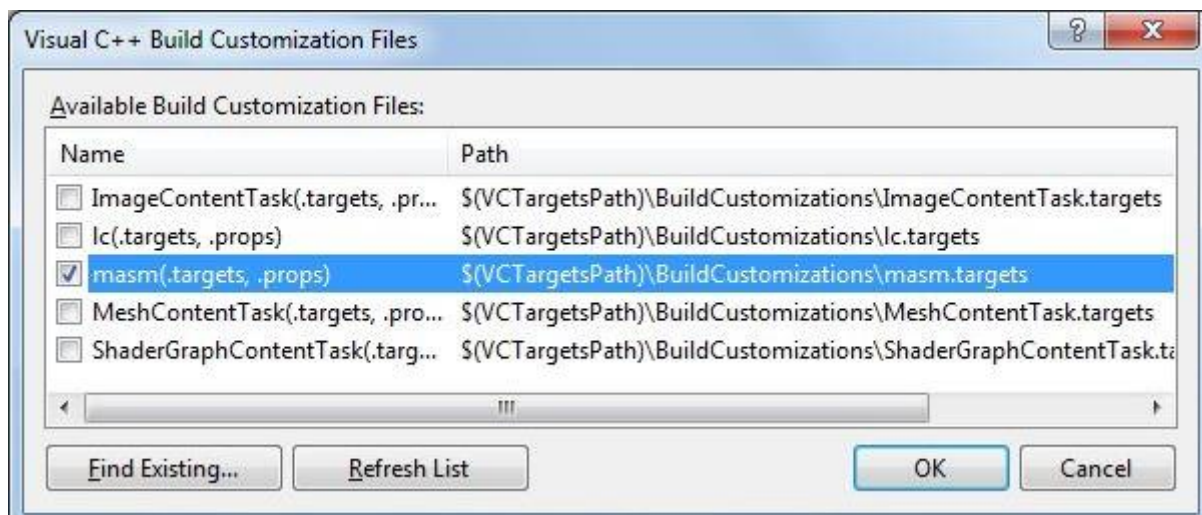
6. W rezultacie wykonania opisanych wyżej operacji pojawi się niżej pokazane okno



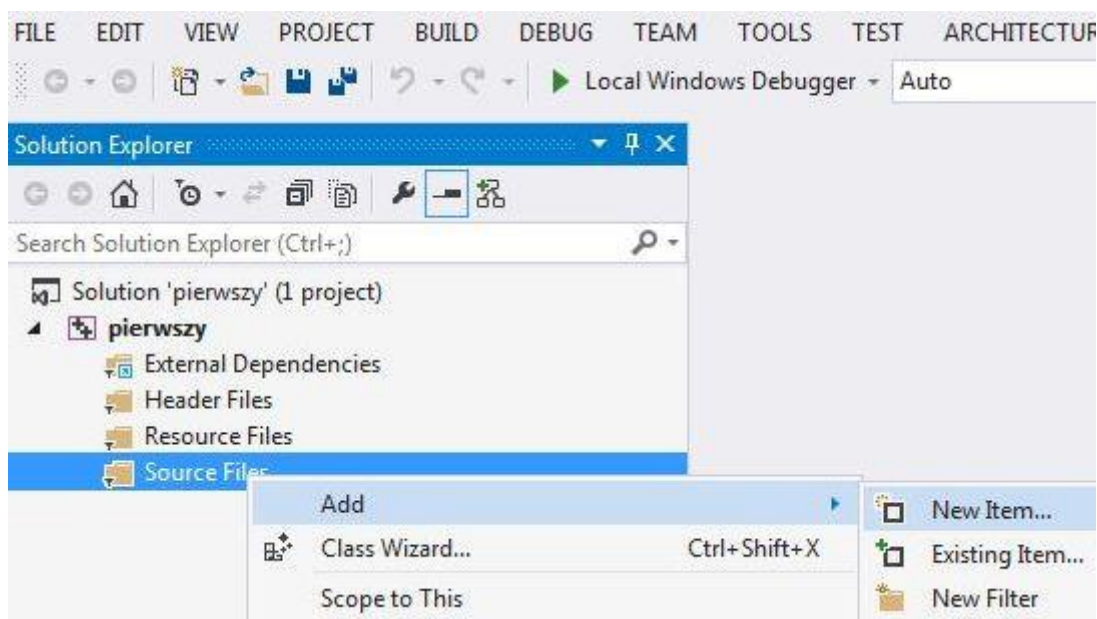
7. Teraz trzeba wybrać odpowiedni asembler. W tym celu należy kliknąć prawym klawiszem myszki na nazwę projektu pierwszy i z rozwijanego menu wybrać opcję Build Dependencies / Build Customizations.



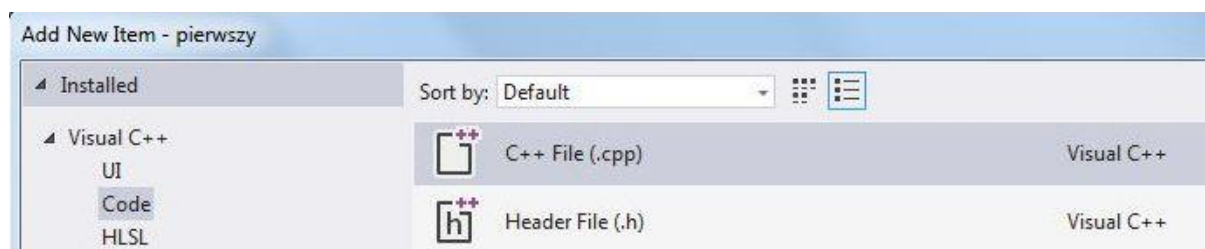
8. W rezultacie na ekranie pojawi się okno (pokazane na poniższym rysunku), w którym należy zaznaczyć pozycję masm i nacisnąć OK.



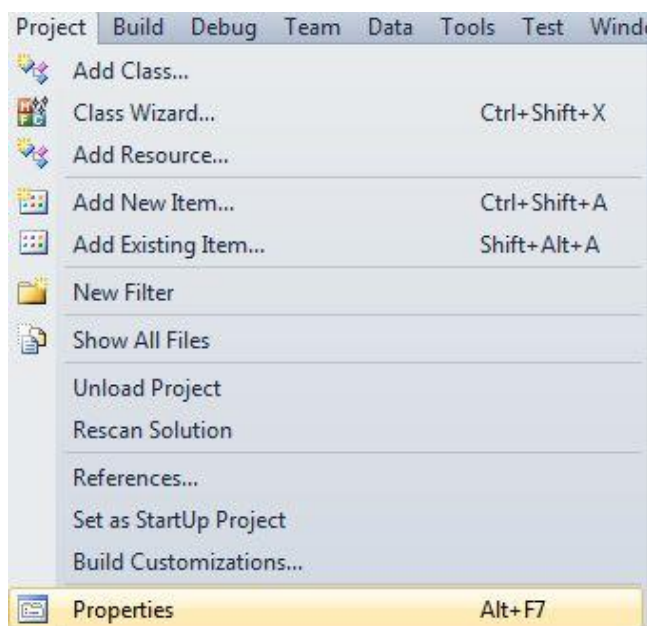
9. Następnie prawym klawiszem myszki należy kliknąć na **Source File** i wybrać opcję **Add / New Item**.



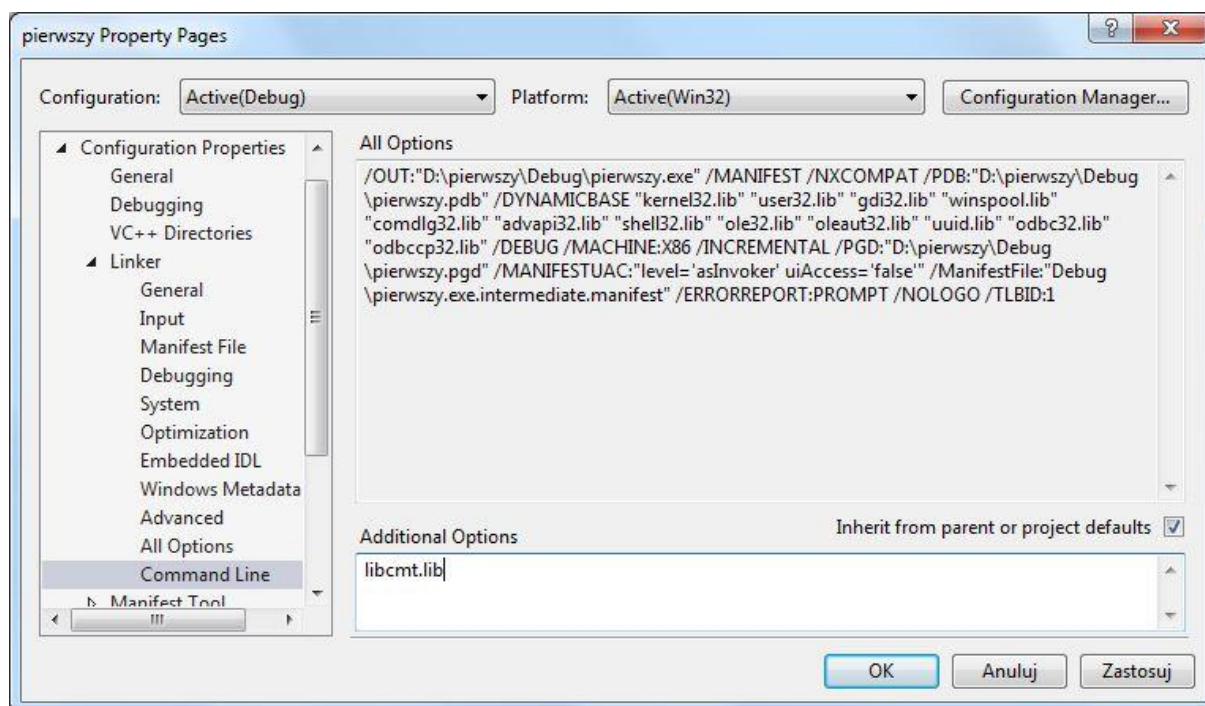
10. W ślad za tym pojawi się kolejne okno, w którym w polu **Installed (Visual C++)** należy zaznaczyć opcję **Code**, a w polu **Name** wpisać nazwę pliku zawierającego programu źródłowy, np. `zdanie32.asm`. Naciskamy przycisk **Add**.



11. W kolejnym kroku należy uzupełnić ustawienia konsolidatora (linkera). W tym celu należy z menu głównego wybrać **Project** i z rozwijanego menu wybrać opcję **Properties**.

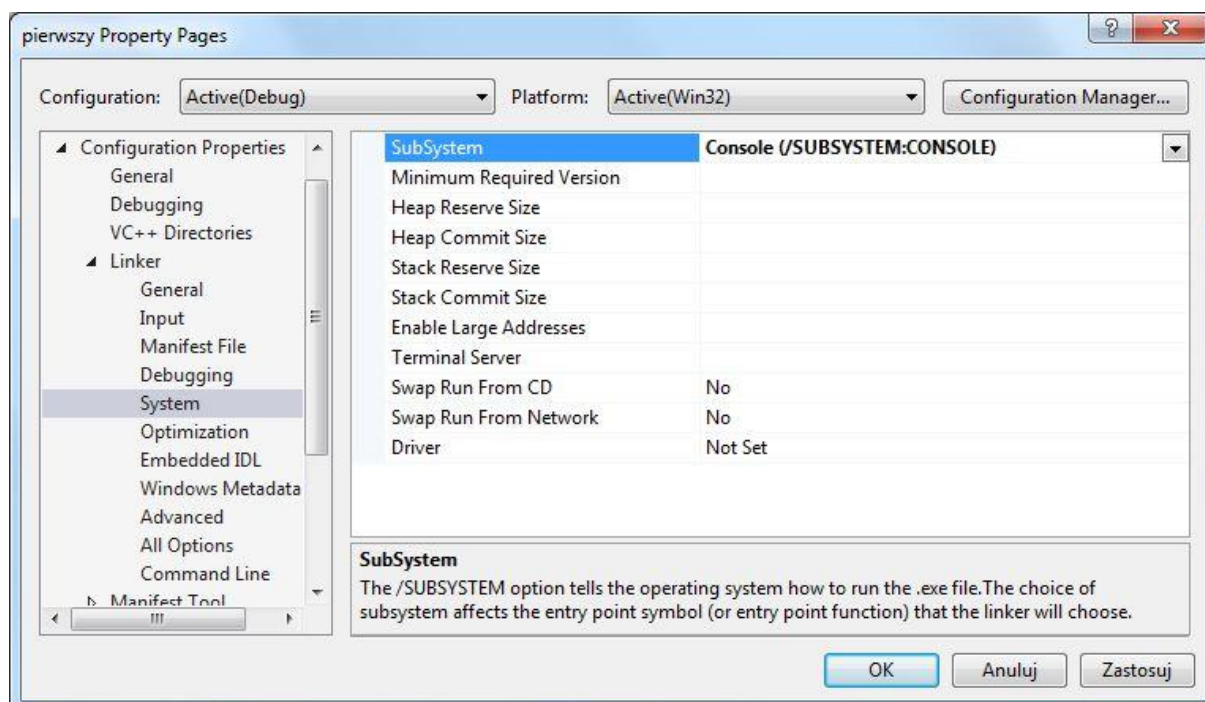


12. W lewym oknie rozwijamy opcję **Configuration Properties**, po czym rozwijamy menu konsolidatora (pozycja **Linker**) i wybieramy opcję **Command line**. Następnie w polu **Additional options** wpisujemy nazwę biblioteki `libcmtd.lib` i naciskamy OK.

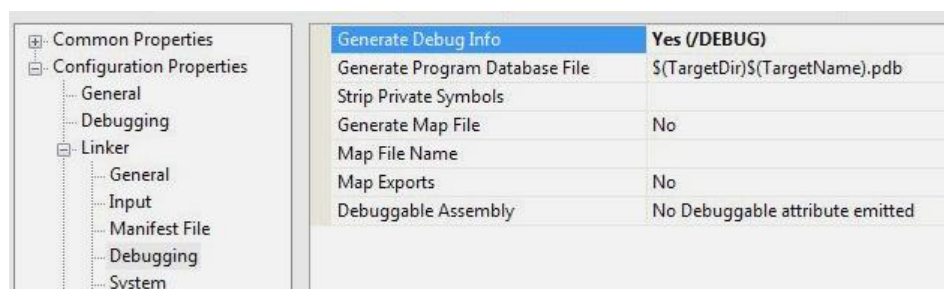


13. W ramach pozycji **Linker** należy ustawić także typ aplikacji. W tym celu zaznaczamy grupę **System** (zob. rysunek) i w polu **SubSystem** wybieramy opcję **Console (/SUBSYSTEM:CONSOLE)**.

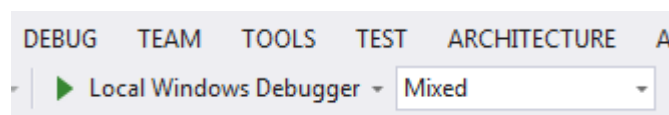




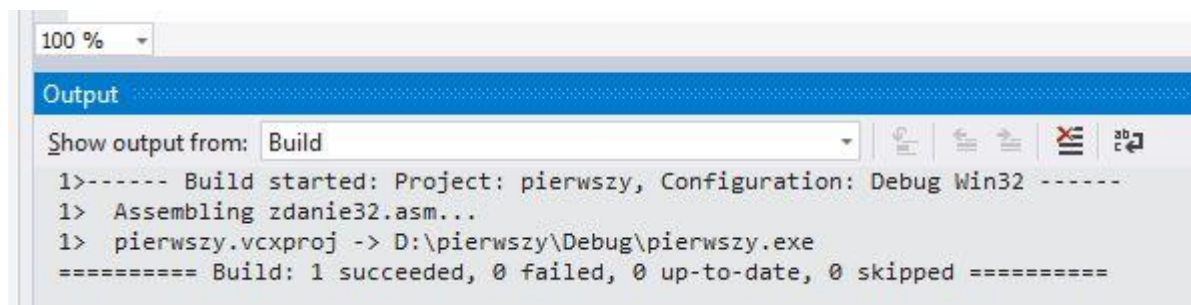
14. Przy okazji warto ustawić opcje aktywizujące debugger. W tym celu wybieramy opcję **Debugging** (stanowiącej rozwinięcie opcji **Linker**), a następnie pole **Generate Debug Info** ustawiamy na **YES**. Następnie naciskamy **OK**.



Ponadto w górnej części okna Visual Studio należy ustawić opcję debuggowania „Mixed” tak jak pokazano na poniższym rysunku.



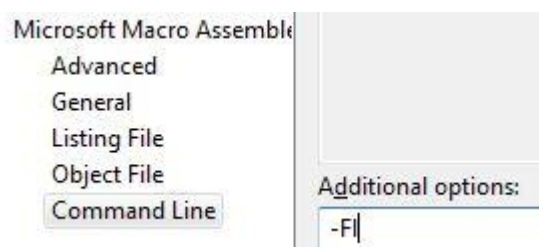
15. Po wykonaniu opisanych czynności przygotowawczych do okna „zdanie32.asm” należy skopiować kod źródłowy programu (podany na stronie 7/8) i nacisnąć **Ctrl S**.
16. W celu wykonania asemblacji i konsolidacji programu wystarczy wybrać opcję **Build / Build Solution**. Opis przebiegu asemblacji i konsolidacji pojawi się w oknie umieszczonym w dolnej części ekranu. Przykładowa postać takiego opisu pokazana jest poniżej.



17. Jeśli nie zidentyfikowano błędów, to można uruchomić program naciskając kombinację klawiszy Ctrl F5. Na ekranie pojawi się okno programu, którego przykładowy fragment pokazany jest poniżej.



18. W trakcie analizy programu w assemblerze może być przydatne sprawozdanie z asemblacji zawarte w pliku z rozszerzeniem .lst (zob. opis podany na str. 12). Do wytworzenia tego pliku w trakcie asemblacji w środowisku MS Visual Studio konieczne jest wprowadzenie dodatkowej opcji asemblacji -F1. W tym celu należy kliknąć prawym klawiszem myszki na nazwę projektu (okno Solution Explorer), wybrać opcję Properties i rozwinąć element Microsoft Macro Assembler. Następnie wybrać pozycję Command Line (zob. rysunek) i w polu Additional options wpisać -F1.
19. Na komputerach domowych wskazane jest wykorzystanie dodatkowych rozszerzeń do Visual Studio takich jak: asmdude (<https://marketplace.visualstudio.com/items?itemName=Henk-JanLebbink.AsmDude>) czy ChASM (<https://ethical.blue/download/ChASM.vsix>), które dokonują wyróżnienia składni. Ze względu na uprawnienia administracyjne nie jest możliwe ich zainstalowanie na wszystkich komputerach w laboratoriach MKZL.



## Uruchamianie programów z wykorzystaniem 64-bitowego assemblera i konsolidatora zewnętrznego

W ciągu ostatnich kilku lat rozszerzono istniejące architektury 32-bitowe stopniowo wprowadzając przetwarzanie 64-bitowe. Najbardziej popularna jest tu architektura oznaczana symbolem AMD64 lub Intel 64. Charakterystycznym elementem tej architektury są 64-bitowe rejestry ogólnego przeznaczenia (zob. rys. na str. 2), które stanowią rozszerzenia stosowanych dotychczas rejestrów 32-bitowych. Między innymi młodsza część 64-bitowego rejestru RAX jest tożsama z nadal używanym 32-bitowym rejestrem EAX, tak samo młodsza część rejestru RBX to po prostu rejestr EBX, itd.

Programy w asemblerze przewidziane do wykonywania w trybie 64-bitowym mają podobną strukturę do stosowanej w trybie 32-bitowym. Istotne różnice występują jedynie w technice przekazywania parametrów do podprogramów (funkcji). W szczególności parametry przekazywane są przez rejestry, przy czym

- pierwszy parametr przekazywany jest przez rejestr RCX ,
- drugi parametr przekazywany jest przez rejestr RDX .
- trzeci parametr przekazywany jest przez rejestr R8 ;
- czwarty parametr przekazywany jest przez rejestr R9 ,
- dalsze parametry, jeśli występują, przesyłane są przez stos.

Poniżej podano program przykładowy `zdanie64.asm`, przewidziany do wykonywania w trybie 64-bitowym.

```
; program przykładowy (wersja 64-bitowa)

extern _write      : PROC
extern ExitProcess : PROC
public main

.data
tekst          db 10, 'Nazywam sie . . . ' , 10
               db  'Moj pierwszy 64-bitowy program asemblerowy '
               db  'działa już poprawnie!', 10

.code
main PROC
    mov rcx, 1 ; uchwyt urządzenia wyjściowego
    mov rdx, OFFSET tekst ; położenie obszaru ze znakami

; liczba znaków wyświetlanego tekstu
    mov r8, 85

; przygotowanie obszaru na stosie dla funkcji _write
    sub rsp, 40

; wywołanie funkcji "_write" z biblioteki języka C
    call _write

; usunięcie ze stosu wcześniej zarezerwowanego obszaru
    add rsp, 40

; wyrównanie zawartości RSP, tak by była podzielna przez 16
    sub rsp, 8

; zakończenie wykonywania programu
    mov rcx, 0 ; kod powrotu programu
    call ExitProcess
```

```
main ENDP
```

```
END
```

Łatwo zauważyć, że struktura tego programu jest prawie identyczna z podanym wcześniej programem 32-bitowym. Główne różnice dotyczą przekazywania parametrów do funkcji `write`, które w tym przypadku przekazywane są przez rejestry `RCX`, `RDX` i `R8`.

W celu przeprowadzenia asemblacji i konsolidacji (linkowania) podanego programu warto przygotować najpierw pliki pomocnicze (analogicznie jak dla przykładu 32-bitowego). Plik `VC64.BAT` ułatwia wywołanie pliku `VCVARS64.BAT`, który konfiguruje ścieżki dostępu (*napisać w jednym wierszu*):

```
"C:\Program Files\Microsoft Visual
Studio\2022\Enterprise\VC\Auxiliary\Build\VCVARS64.bat"
```

Tak jak poprzednio wpisanie do okienka konsoli tekstu `VC64` i naciśnięcie klawisza `Enter` spowoduje wykonanie podanego wyżej polecenia.

Drugi plik pomocniczy `a64.bat` zawiera polecenia asemblacji i konsolidacji programu 64-bitowego:

```
@echo Asemblacja i linkowanie programu 64-bitowego
ml64 -c -Cp -Fl %1.asm
if errorlevel 1 goto koniec
link -subsystem:console -out:%1.exe %1.obj libcmtd.lib
:koniec
```

Korzystając z tego pliku, przetłumaczenie pliku źródłowego `zdanie64.asm` wymaga wprowadzenia polecenia

```
a64  zdanie64
```

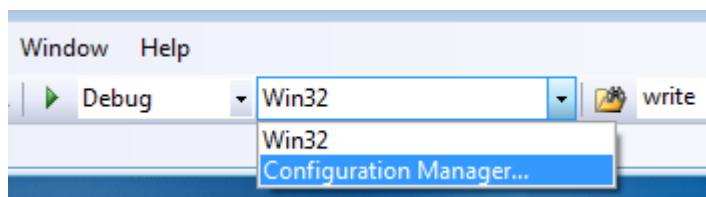
Jeśli asemblacja i linkowanie zostaną wykonane poprawnie, to powstanie plik `zdanie64.exe` zawierający kod programu gotowy do wykonania. W celu wykonania programu wystarczy wpisać tekst `zdanie64` do okienka konsoli i nacisnąć klawisz `Enter` (nie trzeba podawać rozszerzenia `.exe`).

## Uruchamianie programów w standardzie 64-bitowym w środowisku zintegrowanym Microsoft Visual Studio

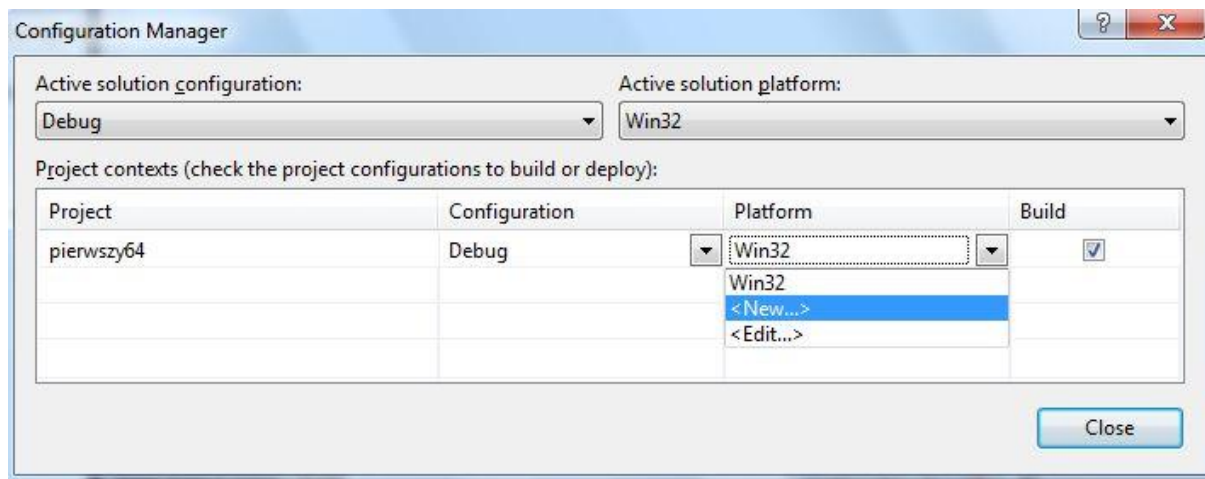
Tworzenie programu 64-bitowego polega, z nielicznymi wyjątkami, na wykonaniu tych samych czynności, które opisano w poprzedniej części instrukcji dla aplikacji 32-bitowych. W tym celu wykonujemy działania opisane w punktach 1 ÷ 14 na str. 15–21, przy czym jako nazwę projektu przyjmujemy `pierwszy64`, a nazwę pliku zawierającego kod źródłowy w asemblerze określamy jako `zdanie64.asm`.

Po wykonaniu podanych czynności trzeba jeszcze zmienić tryb na 64-bitowy. W tym celu w górnej części ekranu trzeba wybrać opcję `Configuration Manager` tak jak pokazano na poniższym rysunku.

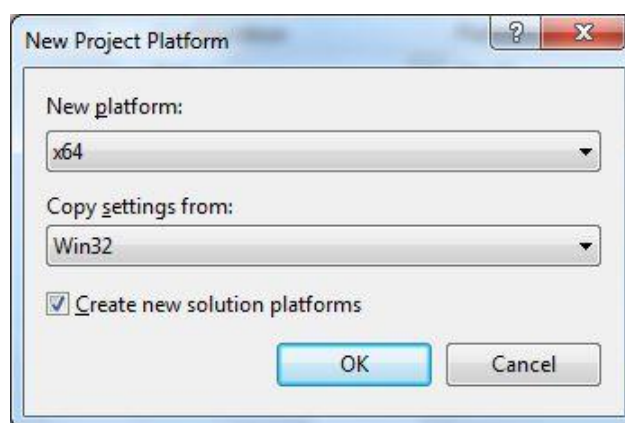




W rezultacie zostanie otwarte pokazane niżej okno — w oknie tym w kolumnie Platform należy wybrać opcję New.



Z kolei pojawi się kolejne okno dialogowe (zob. rys. na następnej stronie), w którym należy wybrać opcję x64 i nacisnąć OK.



Po naciśnięciu Close, w górnej części ekranu pojawi się napis x64 w (zob. rysunek).



W celu wykonania asemblacji i konsolidacji programu wystarczy nacisnąć klawisz wybrać opcję Build / Build Solution. Tak jak poprzednio, opis przebiegu asemblacji i konsolidacji pojawi się w oknie umieszczonym w dolnej części ekranu. Jeśli program był bezbłędny, to

można go uruchomić naciskając kombinację klawiszy Ctrl F5. Poniżej podano fragment okna z wynikami programu.



```

C:\Windows\system32\cmd.exe
Nazywam sie . . .
Moj pierwszy 64-bitowy program asemblerowy dziala juz poprawnie!
Aby kontynuowac, naciśnij dowolny klawisz . . . _
  
```

## Śledzenie programów w środowisku MS Visual Studio

Opisane dotychczas działania służyły do przetłumaczenia programu źródłowego w celu uzyskania wersji w języku maszynowym, zrozumiałym dla procesora. W środowisku systemu Windows kod maszynowy programu przechowywany jest w plikach z rozszerzeniem .EXE. Obok kodu i danych programu w plikach tych zawarte są informacje pomocnicze dla systemu operacyjnego informujące o wymaganiach programu, przewidywanym jego położeniu w pamięci i wiele innych.

Program w języku maszynowym można wykonywać przy użyciu *debuggera*. Wówczas istnieje możliwość zatrzymywania programu w dowolnym miejscu, jak również wykonywania krok po kroku (po jednej instrukcji). *Debuggery* używane są przede wszystkim do wykrywania błędów i testowania programów. W ramach niniejszego ćwiczenia *debugger* traktowany jest jako narzędzie pozwalające na obserwowanie działania procesora na poziomie pojedynczych rozkazów.

W systemie Microsoft Visual Studio *debuggowanie* programu jest wykonywane po naciśnięciu klawisza F5. Przedtem należy ustawić punkt zatrzymania (ang. breakpoint) poprzez kliknięcie na obrzeżu ramki obok rozkazu, przed którym ma nastąpić zatrzymanie. Po uruchomieniu debuggowania, można otworzyć potrzebne okna, wśród których najbardziej przydatne jest okno prezentujące zawartości rejestrów procesora. W tym celu wybieramy opcje Debug / Windows / Registers. (Uwaga: zawartości rejestrów wyświetlane są w postaci liczb w zapisie szesnastkowym). W analogiczny sposób można otworzyć inne okna. Ilustruje to poniższy rysunek.

Wśród wyświetlanych rejestrów procesora szczególnie ważny jest *wskaźnik instrukcji* EIP, nazywany czasami licznikiem rozkazów lub licznikiem programu. Rejestr ten zawiera adres komórki pamięci, z której procesor pobierze kolejny rozkaz (instrukcję) do wykonania. W trakcie wykonywania rozkazu omawiany rejestr zostaje zwiększony o liczbę bajtów, na których zapisany jest aktualnie wykonywany rozkaz. W przypadku opisanych dalej rozkazów sterujących (skoków) zawartość rejestru EIP zmieniana jest wg innych reguł.

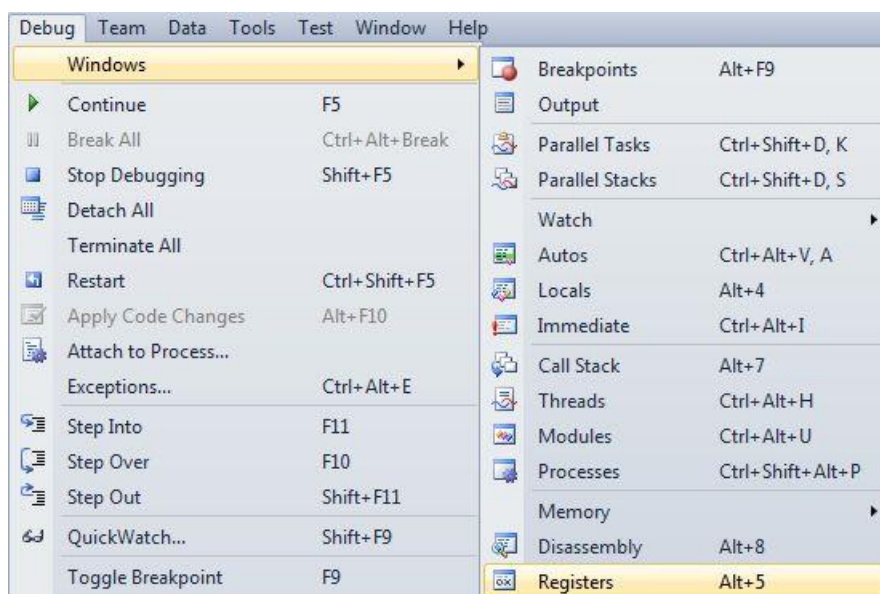
Po naciśnięciu klawisza F5 program jest wykonywany aż do napotkania (zaznaczonego wcześniej) punktu zatrzymania. Można wówczas wykonywać pojedyncze rozkazy programu poprzez wielokrotne naciskanie klawisza F10 (*step over*), obserwując jednocześnie zawartości niektórych rejestrów lub komórek pamięci. Podobne znaczenie ma klawisz F11 (*step into*), ale w tym przypadku śledzenie obejmuje także zawartość podprogramów. Warto dodać, że po naciśnięciu klawisza F10 procesor pobiera i wykonuje pojedynczy rozkaz, który znajduje się w pamięci głównej (operacyjnej) pod adresem podanym w rejestrze EIP.

W pewnych sytuacjach używa się także kombinacji klawiszy Shift F11 (*step out*) – naciśnięcie tej kombinacji powoduje wyjście z podprogramu i przejście do programu wywołującego.

Wybierając opcję **Debug / Stop debugging** można zatrzymać debuggowanie programu. Prócz podanych, dostępnych jest jeszcze wiele innych opcji, które można wywołać w analogiczny sposób.

Niekiedy trzeba wielokrotnie odczytywać zawartość znaczników procesora ZF i CF. W celu odczytania zawartości tych znaczników należy kliknąć prawym klawiszem myszki w oknie **Registers** i wybrać opcję **Flags**. Znaczniki te oznaczone są nieco inaczej: znacznik ZF oznaczony jest symbolem ZR, a znacznik CF – symbolem CY.

Ponadto zawartość całego rejestru znaczników (32 bity) wyświetlana jest w oknie **Registers** w postaci liczby szesnastkowej oznaczonej symbolem EFL. Na tej podstawie można także określić stan znaczników ZF i CF. Przedtem jednak trzeba dwie ostatnie cyfry szesnastkowe zamienić na binarne i w uzyskanym ciągu 8-bitowym wyszukać pozycje ZF i CF. Nie stanowi to problemu, jeśli wiadomo, że znacznik ZF zajmuje bit nr 6, a znacznik CF zajmuje bit nr 0. Przykładowo, jeśli w oknie debuggera wyświetlana jest liczba  $EFL = 00000246$ , to konwersja dwóch ostatnich cyfr (46) na postać binarną daje wynik 0100 0110. Bity numerowane są od prawej do lewej, zatem bit numer 0 zawiera 0 (czyli  $CF = 0$ ), a bit numer 6 zawiera 1 (czyli  $ZF = 1$ ).



Debugery oferują jeszcze wiele innych opcji wspomagających uruchamianie programów. Między innymi możliwa jest obserwacja zmian zawartości wskazanych zmiennych lub obszarów pamięci, obserwacja zawartości stosu, disasemblacja kodu programu, itd. Niektóre z wymienionych opcji omawiane są na końcu niniejszej instrukcji w formie dodatku.

### Przykład wykorzystania debuggera do śledzenia fragmentu programu, w którym wykonywane są operacje arytmetyczne

Do opracowanego wcześniej programu wprowadzimy teraz krótki fragment, w którym obliczana jest suma wyrazów ciągu  $3 + 5 + 7 + 9 + 11$ . Podany fragment należy wprowadzić do programu przykładowego (32-bitowego) bezpośrednio przed rozkazem:

```
mov     ecx, 85
```

W poniższym fragmencie to samo obliczenie wykonywane jest dwukrotnie: najpierw bez użycia pętli rozkazowej, następnie za pomocą pętli rozkazowej.

```
; obliczenie sumy wyrazów ciągu 3 + 5 + 7 + 9 + 11

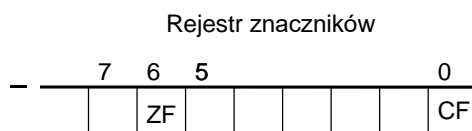
; obliczenie bez użycia pętli rozkazowej
    mov     eax, 3    ; pierwszy element ciągu
    add     eax, 5    ; dodanie drugiego elementu
    add     eax, 7    ; dodanie trzeciego elementu
    add     eax, 9    ; dodanie czwartego elementu
    add     eax, 11   ; dodanie piątego elementu

; obliczenie z użyciem pętli rozkazowej
    mov     eax, 0    ; początkowa wartość sumy
    mov     ebx, 3    ; pierwszy element ciągu
    mov     ecx, 5    ; liczba obiegów pętli
ptl:   add     eax, ebx ; dodanie kolejnego elementu
       add     ebx, 2   ; obliczenie następnego elementu
       sub     ecx, 1   ; zmniejszenie licznka obiegów pętli
       jnz    ptl      ; skok, gdy licznik obiegów różny od 0
```

Spróbujmy teraz przeanalizować działanie fragmentu programu. W pierwszej części programu obliczenia wykonywane są bez użycia pętli rozkazowej. Najpierw do rejestru EAX wpisywany jest pierwszy element ciągu (rozkaz `mov eax, 3`), a następnie do rejestru EAX dodawane są następne elementy (rozkaz `add eax, 5` i dalsze).

W drugiej części programu wykonywane są te same obliczenia, ale z użyciem pętli rozkazowej. Przed pętlą zerowany jest rejestr EAX, w którym obliczana będzie suma, ustawiany jest licznik obiegów pętli w rejestrze ECX (rozkaz `mov ecx, 5`), zaś do rejestru EBX wpisywana jest wartość pierwszego elementu ciągu (rozkaz `mov ebx, 3`). Następnie wykonywane są rozkazy wchodzące w skład pętli rozkazowej.

Istotnym elementem tego fragmentu jest sterowanie pętlą w taki sposób, by rozkazy wchodzące w skład pętli zostały wykonaneadaną liczbę razy. W tym celu przed pętlą ustawiany jest licznik obiegów, który realizowany jest zazwyczaj za pomocą rejestru ECX (rozkaz `mov ecx, 5`). W każdym obiegu pętli zawartość rejestru ECX jest zmniejszana o 1 (rozkaz odejmowania `sub ecx, 1`). Rozkaz odejmowania `sub` (podobnie jak rozkaz dodawania `add` i wiele innych rozkazów) ustawia między innymi znacznik zera ZF w rejestrze znaczników.



Jeśli wynikiem odejmowania jest zero, to do znacznika ZF wpisywana jest 1, w przeciwnym razie znacznik ZF jest zerowany. Kolejny rozkaz (`jnz ptl`) jest rozkazem sterującym (skoku), który testuje stan znacznika ZF. Z punktu widzenia tego rozkazu testowany warunek jest spełniony, jeśli znacznik ZF zawiera wartość 0. Jeśli warunek jest spełniony to nastąpi

skok do miejsca w programie opatrzonego etykietą, a jeśli nie jest spełniony, to procesor będzie wykonywał rozkazy w naturalnym porządku.

Zatem w każdym obiegu pętli zawartość rejestru **ECX** będzie zmniejszana o 1, ale dopiero w ostatnim obiegu zawartość rejestru **ECX** przyjmie wartość 0. Wówczas warunek testowany przez rozkaz **jnz** nie będzie spełniony, skok nie zostanie wykonany i procesor wykona rozkaz podany w programie jako następny.

Pamiętamy jednak, że procesor wykonuje program zapisany w języku maszynowym w postaci ciągów zerojedynkowych. W szczególności rozkaz **jnz** jest dwubajtowy, a jego struktura pokazana jest na poniższym rysunku.

01110101	Zakres skoku
----------	--------------

W kodzie maszynowym nie występuje więc pole etykiety, ale bajt określający *zakres skoku*. Jeśli warunek testowany przez rozkaz skoku jest spełniony, to liczba (dodatnia lub ujemna) umieszczona w polu *zakres skoku* jest dodawana do zawartości wskaźnika instrukcji **EIP** (do **EIP** dodawana jest także liczba bajtów samego rozkazu skoku, tu: 2). Asembler w trakcie tłumaczenia programu źródłowego na kod maszynowy dobiera liczbę w polu *zakres skoku* w taki sposób, by liczba ta (tu: ujemna) dodana do rejestru **EIP** zmniejszyła jego zawartość o tyle, by jako kolejny rozkaz procesor pobrał z pamięci rozkaz `add eax, ebx`), tj. rozkaz opatrzonej etykietą `pt1`.

Opisany tu fragment programu wprowadzimy teraz do komputera i uruchomimy w środowisku systemu Microsoft Visual Studio. Przedtem należy ustawić punkt zatrzymania (ang. breakpoint) poprzez kliknięcie na obrzeżu ramki obok rozkazu:

```
mov     eax, 3 ; pierwszy element ciągu
```

Po naciśnięciu klawisza **F5** rozpocznie się wykonywanie programu i niezwłoczne jego zatrzymanie, tak że podany wyżej rozkaz na razie nie zostanie wykonany. W celu wykonania podanego rozkazu i następnych należy wielokrotnie naciskać klawisz **F10**. Jednocześnie warto obserwować jak zmieniają się zawartości rejestrów procesora w trakcie wykonywania kolejnych rozkazów. Szczególnie ważna jest obserwacja zawartości wskaźnika instrukcji **EIP**.

## Wskazówki praktyczne dotyczące konfiguracji środowiska Microsoft Visual Studio

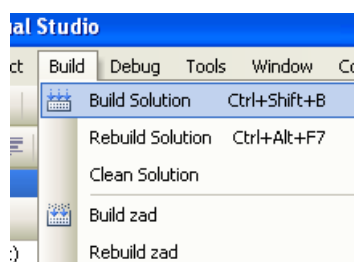
Opisane w poprzedniej części zasady uruchamiania programów w środowisku zintegrowanym Microsoft Visual Studio dotyczą środowiska w konfiguracji standardowej, tj. w postaci bezpośrednio po instalacji systemu. Jednak użytkownicy MS Visual Studio mogą dokonywać zmian konfiguracji, które mają charakter trwały. Przykładowo, może być zmienione znaczenie kombinacji klawiszy, które uruchamiają kompilację i konsolidację programu. W tej sytuacji podane wcześniej reguły postępowania muszą być częściowo zmodyfikowane. W dalszej rozpatrzmy typowe przypadki postępowania.

### Zmiana układu klawiatury

W trakcie uruchamiania programów korzystamy często z różnych klawiszy funkcyjnych a także kombinacji klawiszy Ctrl, Shift i Alt z innymi znakami. Wśród tych kombinacji występuje także kombinacja Ctrl Shift, która w systemie Windows jest interpretowana jako zmiana układu klawiatury: *klawiatura programisty* ↔ *klawiatura maszynistki*. W rezultacie zostaje zmienione znaczenie wielu klawiszy, np. klawisz znaku średnika jest interpretowany jako litera *l*. Jeśli zauważymy, że klawiatura pracuje w *układzie maszynistki*, to należy niezwłocznie nacisnąć kombinację klawiszy Ctrl Shift w celu ponownego włączenia *układu programisty*.

### Kompilacja i konsolidacja programu

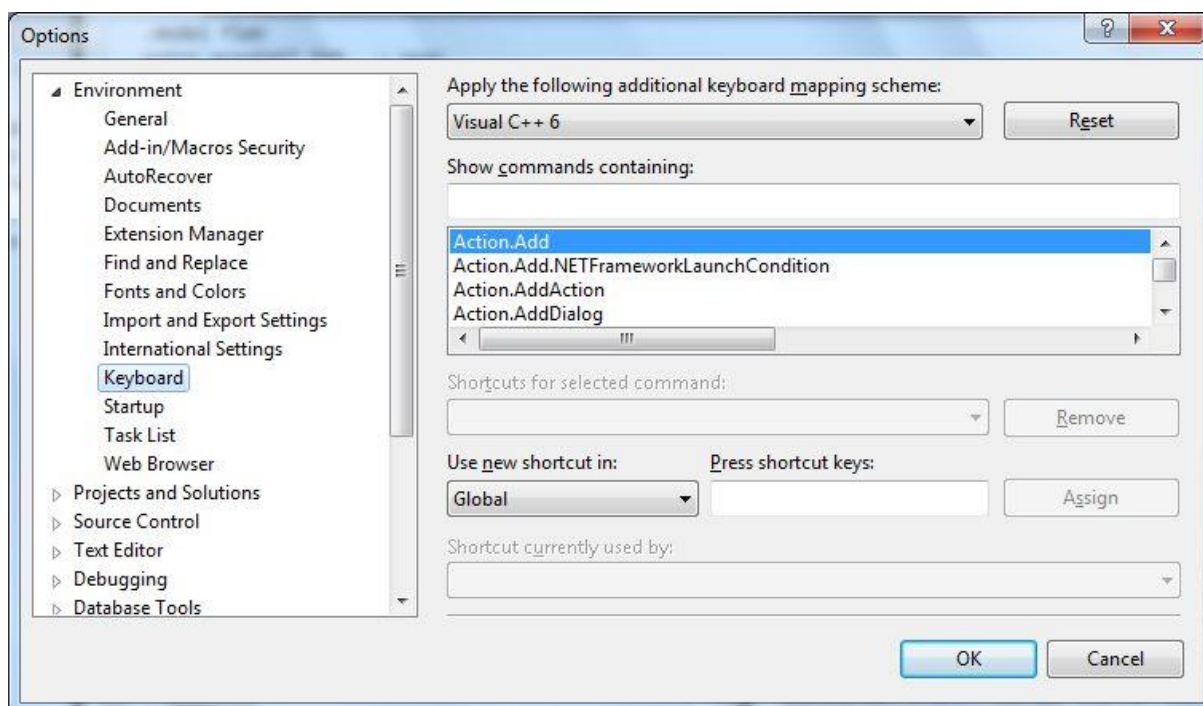
Kompilacja i konsolidacja programu w wersji standardowej następuje po naciśnięciu klawisza F7. W niektórych konfiguracjach zamiast klawisza F7 używa się kombinacji klawiszy Ctrl Shift B (zob. rys.). W takim przypadku w celu wykonania kompilacji (asemblacji) programu i konsolidacji należy wybrać z menu opcję Build / Build Solution i kliknąć myszką na wybraną opcję albo nacisnąć kombinację klawiszy Ctrl Shift B.



### Zmiana kombinacji klawiszy (ang. shortcut key)

Istnieje też możliwość zmiany skrótu klawiszy przypisanego wybranej operacji. W tym celu należy kliknąć myszką na opcję menu Tools / Options, wskutek czego na ekranie pojawi się okno dialogowe Options. W lewej części tego okna należy rozwinąć pozycję Environment i zaznaczyć Keyboard (zob. rys.).





W oknie po prawej stronie należy odszukać operację, której chcemy przypisać inny skrót klawiszy, np. `Build.BuildSolution`. Poszukiwanie będzie łatwiejsze jeśli do okienka `Show commands containing` wprowadzimy początkową część nazwy szukanej operacji, np. `Build`. Po odnalezieniu potrzebnej pozycji należy kliknąć myszką w okienku `Press shortcut keys` i nacisnąć kombinację klawiszy, która ma zostać przypisana wybranej operacji. W rezultacie w omawianym okienku pojawi się opis wybranej kombinacji klawiszy, a naciśnięcie przycisku `Assign` powoduje dopisanie tej kombinacji do listy w okienku `Shortcuts for selected commands`. Uwaga: jeśli zachowano dotychczas używaną kombinację klawiszy, to będzie ona nadal dostępna (i wyświetlana w menu). Nowa kombinacja klawiszy będzie także dostępna, ale nie pojawi się w menu.

## Uaktywnienie asemblera

W praktyce uruchamiania programów w asemblerze można często zaobserwować pozorną kompilację, która kończy się pomyślnie (komunikat: `1 succeeded`), ale program wynikowy nie daje się uruchomić. Przyczyną takiego zachowania jest odłączenie asemblera. W celu uaktywnienia asemblera w oknie `Solution Explorer` (zob. rys.) należy prawym klawiszem myszki zaznaczyć nazwę projektu (tu: pierwszy) i wybrać opcję `Build Dependencies / Build Customizations`. Dalej należy postępować tak jak opisano w pkt. 6 i 7 na str. 16-17. Po wykonaniu tych czynności należy usunąć plik `.asm` z grupy `Source Files` (opcja `Remove`) i ponownie go dołączyć (`Add Existing Item`) do grupy `Source Files`.

## Zmiana opcji asemblera

Asemlacja programu przeprowadzona jest za pomocą asemblera `ml.exe`, który wywołany jest z zestawem standardowych opcji dla trybu konsoli: `-c -Cp -coff -Fl`.

W szczególnych przypadkach może pojawić się konieczność dodania innych opcji czy też usunięcia istniejących.

W tym celu w oknie **Solution Explorer** należy prawym klawiszem myszki zaznaczyć nazwę projektu (tu: pierwszy) i wybrać opcję **Properties**. W rezultacie na ekranie pojawi się okno dialogowe pierwszy **Property Pages**. W lewej części tego okna należy rozwinąć pozycję **Microsoft Macro Assembler** (zob. rys.).



Z kolei, po zaznaczeniu opcji **Command Line** w prawej części okna, na ekranie pojawi się aktualna lista opcji (**All options:**) asemblera `ml.exe` (znak `/` przed opcją ma takie samo znaczenie jak znak `-`). Lista ta wyświetlana jest na szarym tle i może być zmieniona tylko poprzez zmiany opcji asemblacji dostępne poprzez wybranie jednej z pozycji w lewej części okna: **General**, **Listing File**, itd.

```
ml.exe /c /nologo /Zi /Fo"Debug\%(FileName).obj" /Fl"" /W3
/errorReport:prompt /Ta
```

Do podanej listy można dołączyć dodatkowe opcje poprzez wpisanie ich do okna **Additional options** (poniżej okna **All options**).

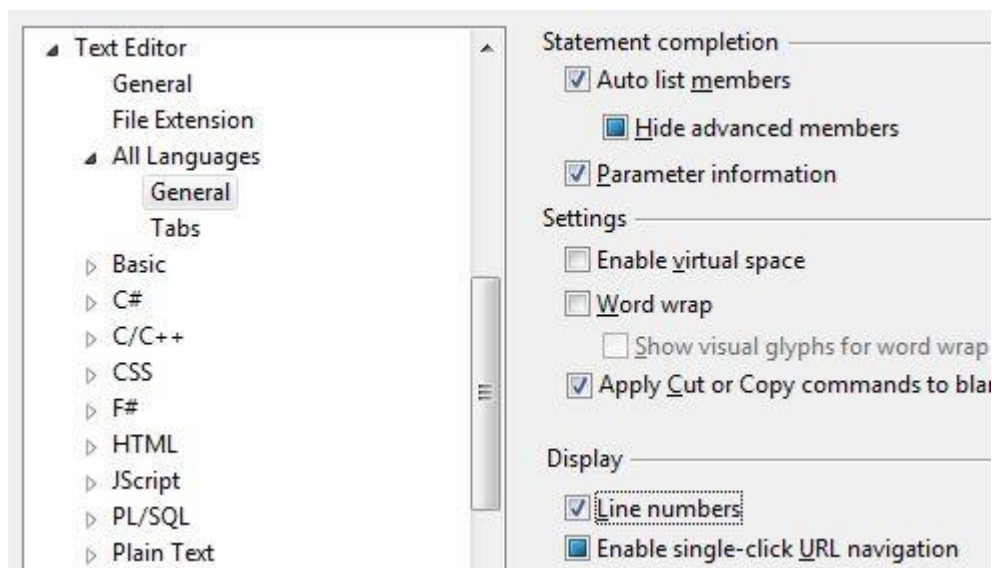
### Kompilacja (asemblacja) i konsolidacja w przypadku znacznych zmian w konfiguracji

Niekiedy zmiany konfiguracyjne dokonane przez poprzednich użytkowników **MS Visual Studio** są tak głębokie, że nawet trudno rozpocząć pracę w systemie. W takim przypadku można wybrać opcję **Window / Reset Window Layout** i potwierdzić wybór za pomocą przycisku **Tak**. W rezultacie zostanie przywrócony domyślny układ okien, który jest dostosowany do typowych zadań. Czasami wystarcza wybranie opcji **View / Solution Explorer**.

### Wyświetlanie numerów wierszy programu źródłowego

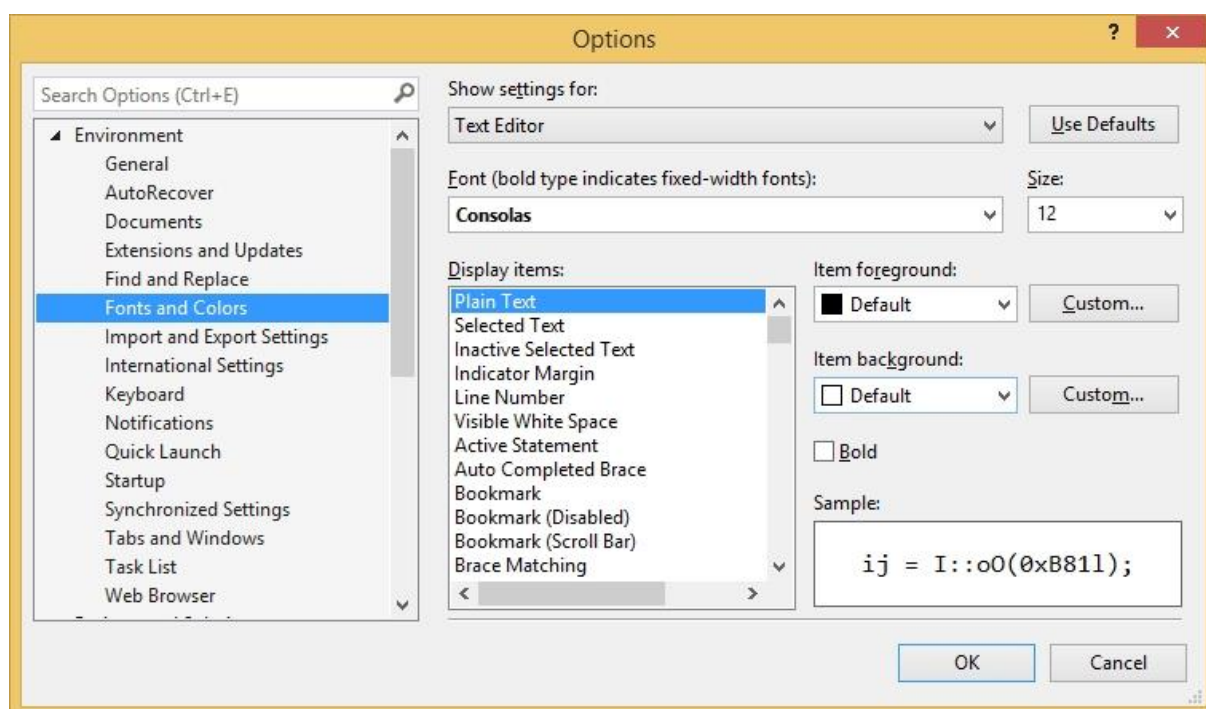
Opcjonalnie można wyświetlać numery wierszy programu źródłowego. W tym celu należy wybrać opcję **Tools / Options**. W oknie dialogowym z lewej strony należy wybrać opcję **Text Editor / All Languages / General**, a w oknie po prawej stronie zaznaczyć kwadracik **Line numbers**, tak jak pokazano na poniższym rysunku.





### Zmiana wielkości i koloru czcionki

W celu zmiany wielkości czcionki i ewentualnie koloru tekstu programu należy wybrać opcję Tools / Options / Environment / Fonts and Colors. Pojawi się wówczas niżej pokazane okno dialogowe, w którym należy ustawić wymagane parametry.



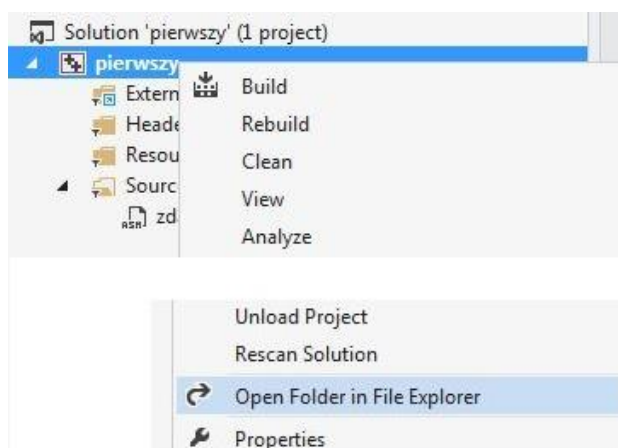
### Zmiana koloru tła

W celu zmiany koloru tła programu należy wybrać opcję Tools / Options / Environment / General. Pojawi się wówczas okno dialogowe, w którym należy wybrać tło

spośród podanych w „Color Theme” (w górnej części okna) ustawić wymagane parametry. Oferowane są trzy kolory tła: light, blue, dark (ciemne tło).

## Wyświetlanie listy plików programu

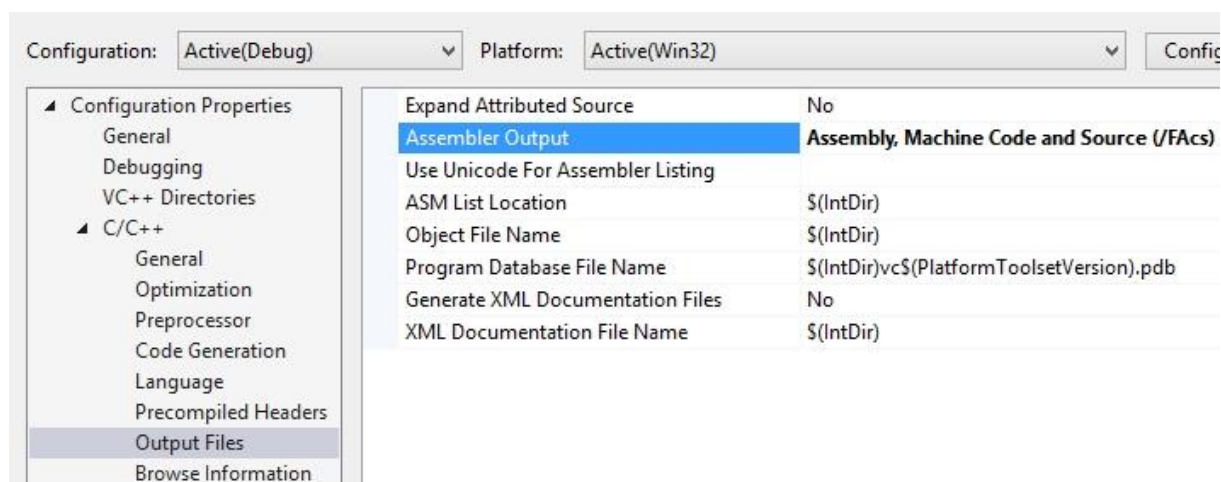
W niektórych przypadkach potrzebne są informacje o położeniu plików składających się na cały program. Można wówczas w oknie **Solution Explorer** kliknąć prawym klawiszem myszki na nazwę projektu (tu: pierwszy) i wybrać opcję **Open Folder in File Explorer** (tak jak pokazano na rysunku). W rezultacie zostanie wyświetlona lista plików uruchamianej aplikacji — przykładowa postać podana jest poniżej.



## Tworzenie kodu assemblerowego dla programu języku C

Niekiedy może być potrzebna znajomość kodu w assemblerze dla programu w języku C. Uzyskanie takiego kodu wymaga podjęcia niżej opisanych działań.

1. Kliknąć prawym klawiszem myszki na nazwę projektu i wybrać opcję **Properties**. W rezultacie zostanie otwarte niżej pokazane okno dialogowe, w którym po lewej stronie należy rozwinąć opcję **C/C++**, a następnie wybrać pozycję **Output Files**. Opcja **C/C++** nie występuje, jeśli w projekcie nie zadeklarowano pliku z rozszerzeniem **C** lub **C++**.



2. W prawej części okna należy wybrać pozycję **Assembler Output** i zaznaczyć w niej opcję **Assembly, Machine Code and Source**.
3. Po przeprowadzeniu kompilacji programu źródłowego powstanie między innymi plik (w katalogu `..\Debug`) z rozszerzeniem `.cod`, w którym zawarty będzie kod assemblerowy skompilowanego programu.
4. Przykład: poniższy fragment programu w języku C

```
int p, q;
p = -4;
q = p * p + 1;
```

zostanie przekształcony do podanej postaci.

```
; 6      : int p, q;
; 7      : p = -4;

0001e    c7 45 f8 fc ff
         ff ff          mov  DWORD PTR _p$[ebp], -4 ; ffffffffch

; 8      : q = p * p + 1;

00025    8b 45 f8      mov  eax, DWORD PTR _p$[ebp]
00028    0f af 45 f8    imul     eax, DWORD PTR _p$[ebp]
0002c    83 c0 01      add  eax, 1
0002f    89 45 ec      mov  DWORD PTR _q$[ebp], eax
```

## Informacje dodatkowe dotyczące debuggowania programów

Podane tu informacje dodatkowe dotyczące debuggowania programów mają charakter uzupełniający i nie są bezpośrednio związane z niniejszą instrukcją. Mogą być natomiast przydatne w trakcie realizacji kolejnych ćwiczeń laboratoryjnych.

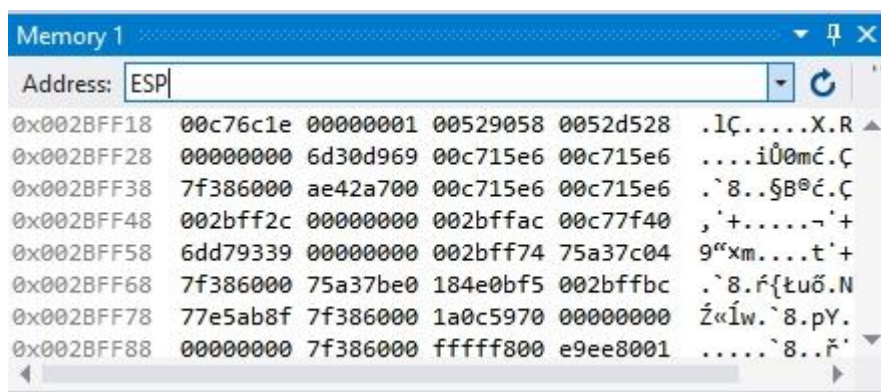
### Rozwiązanie braku możliwości debuggowania pomimo ustawienia właściwych opcji

W wersjach Visual Studio 2017 i 2019 podczas uruchomienia debuggowania może występować zjawisko braku zatrzymania na breakpointcie. Możliwą przyczyną tego zjawiska jest oznaczenie początku programu etykietą `main:`, a nie poprzez nagłówek funkcji `_main` PROC. Należy zastąpić etykietę blokiem

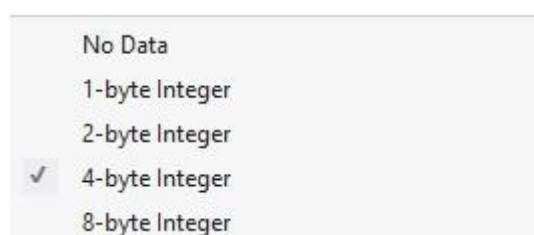
```
_main PROC
; ..... tutaj kod programu
_main ENDP
```

### Obserwacja zawartości stosu

Po uruchomieniu debuggowania programu, tak jak opisano wcześniej należy wybrać opcje **Debug / Windows / Memory**. W rezultacie pojawi się niżej pokazane okno, w którym w polu **Address** należy wpisać **ESP** (wskaźnik stosu).



Ponieważ elementy stosu są liczbami 32-bitowymi (w trybie 32-bitowym), więc wygodnie będzie wprowadzenie formatu 32-bitowego dla wyświetlanych liczb. W tym celu należy kliknąć prawym klawiszem myszki i wybrać opcję „4-byte Integer” (zob. rys.). Oczywiście, w zależności od typu danych przechowywanych na stosie można wybrać inny format.



W prawie identyczny sposób można wyświetlić zawartość zmiennej deklarowanej w sekcji danych: w polu Address należy wpisać nazwę zmiennej poprzedzoną symbolem &, np. &liczba.

### Przejsięcie do odległego rozkazu w programie podczas debuggowania

– zaznaczyć ten rozkaz poprzez wstawienie punktu zatrzymania (break point) i nacisnąć klawisz F5.

### Podawanie parametrów w linii zlecenia (wywołania) programu

Niektóre programy, zwłaszcza wywoływane z poziomu konsoli, wymagają podania parametrów w linii zlecenia programu. Jeśli program wywoływany jest z poziomu Visual Studio, to wymagane parametry ustawia się poprzez wybranie opcji:

Properties / Configuration Properties / Debugging / Command Arguments

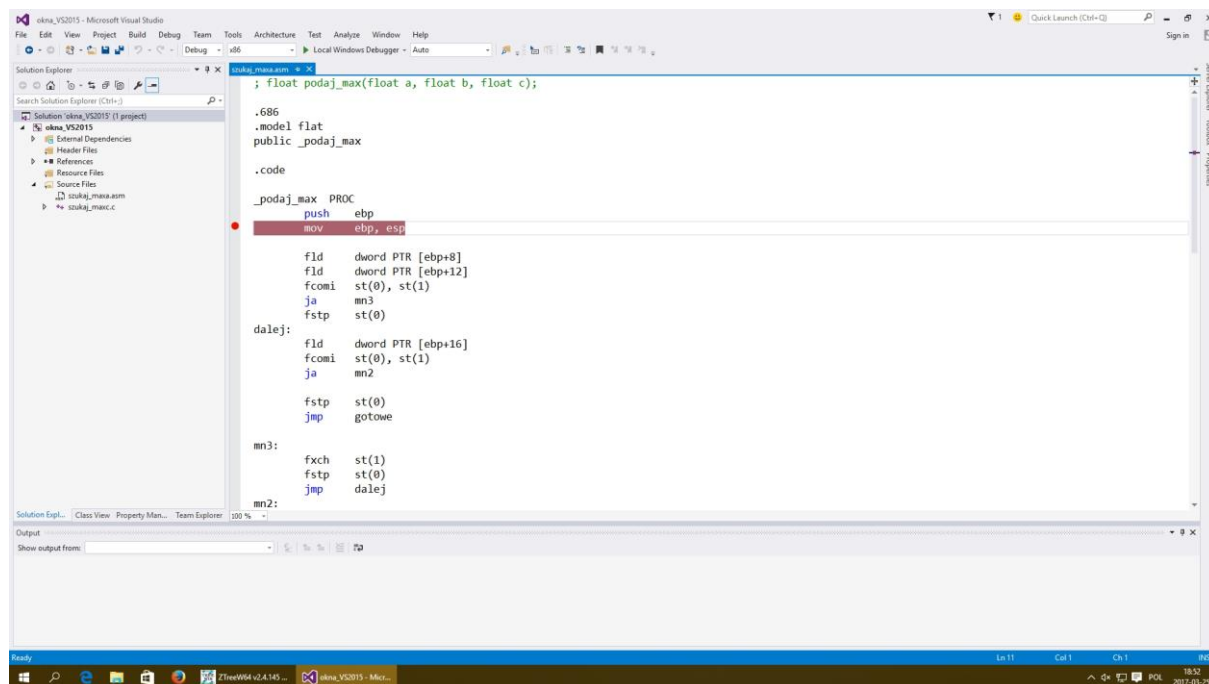
i następnie wpisanie odpowiednich wartości argumentów.

### Usytuowanie okien na ekranie

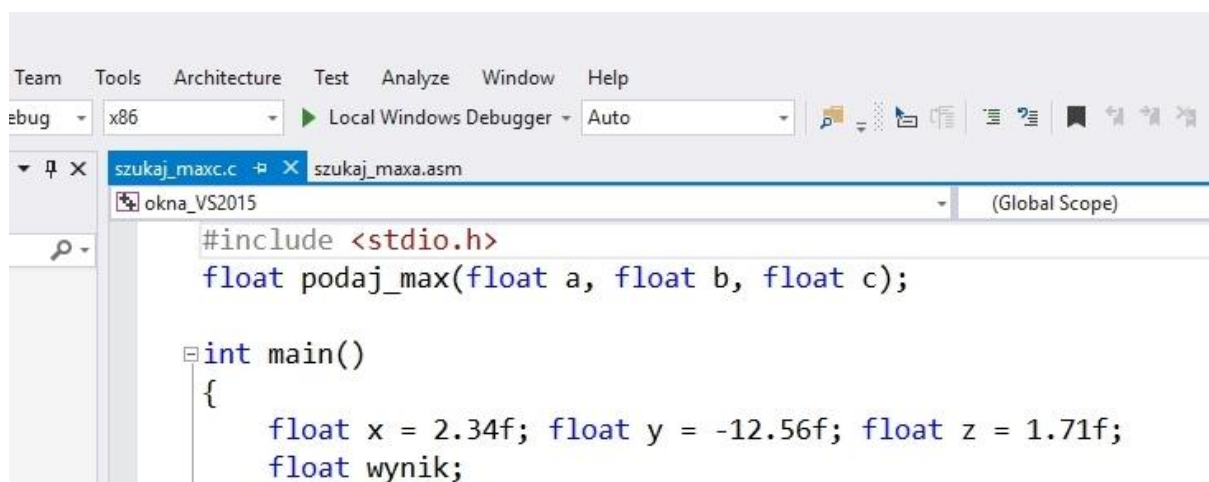
Instrukcja do ćwiczenia laboratoryjnego nr 4 omawia zasady *programowania mieszanego*, czyli tworzenia programów, których fragmenty kodowane są w różnych językach programowania. W praktyce często mamy do czynienia z programami w języku C (lub C++), do których dołączane są podprogramy kodowane w assemblerze. Poniżej pokazano w jaki

sposób w środowisku Visual Studio można jednocześnie wyświetlać na ekranie kody źródłowe dwóch (lub więcej) fragmentów (plików) tego samego programu.

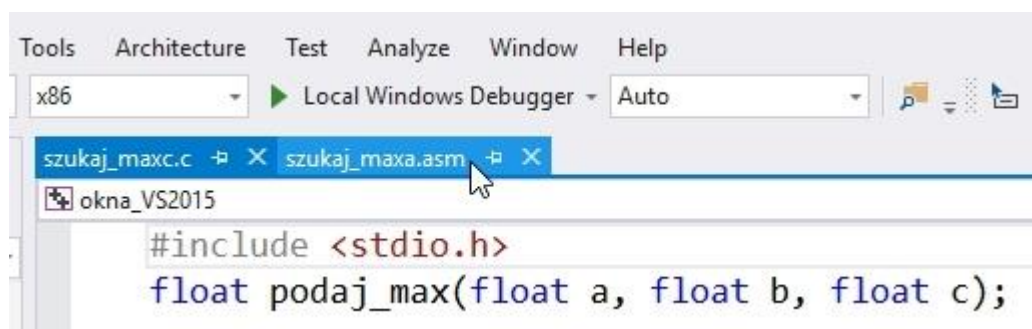
Omawiany tu program przykładowy zawiera dwa pliki źródłowe, z których jeden zawiera kod w języku C (szukaj\_maxc.c), a drugi w asemblerze (szukaj\_maxa.asm). W celu uruchomienia tego programu w środowisku *Visual Studio* tworzymy projekt (tak jak opisano na str. 15 i dalszych). Po utworzeniu projektu zazwyczaj otwieramy najpierw okno zawierające kod w asemblerze – przykładowy wygląd ekranu pokazano na poniższym rysunku.



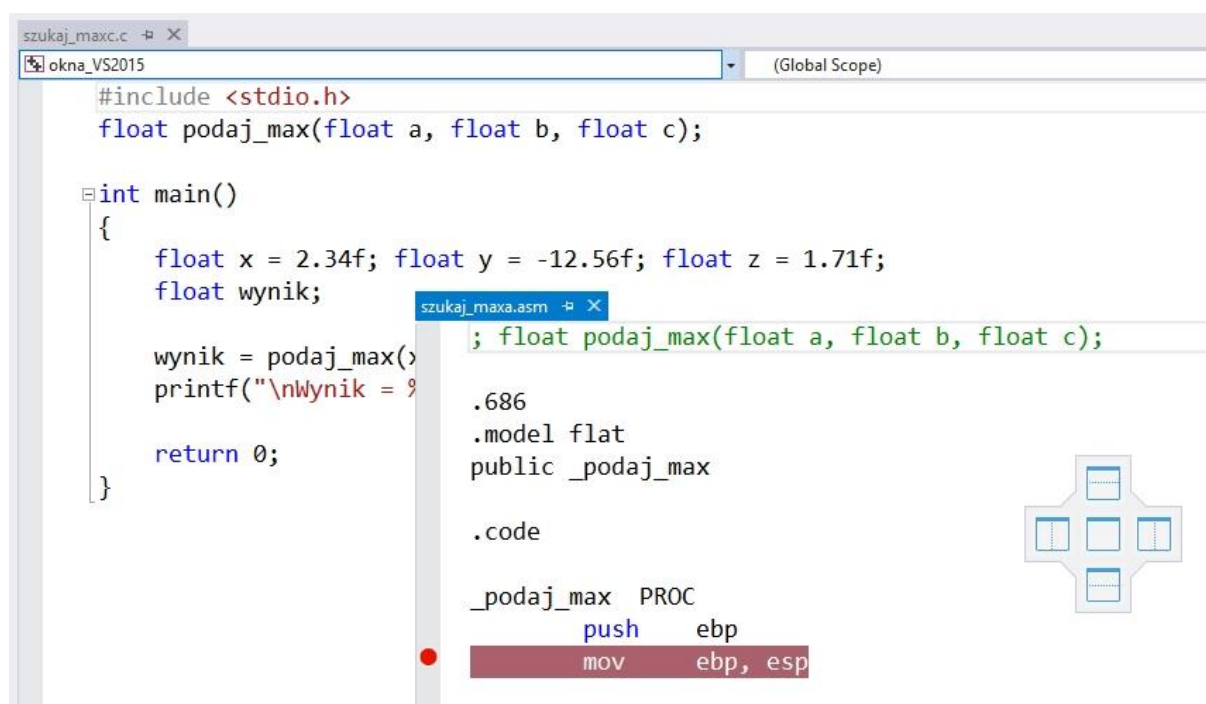
Uruchomienie programu będzie łatwiejsze, jeśli na ekranie, w sąsiednim oknie będzie również wyświetlany kod w języku C. W tym celu w oknie Solution Explorer trzeba dwukrotnie kliknąć na nazwę pliku `szukaj_maxc.c`. W rezultacie zamiast fragmentu kodu w asemblerze na ekranie będzie wyświetlany kod w języku C – poniżej pokazano przykładowy wygląd górnej części ekranu.



W kolejnym kroku trzeba przesunąć kursor myszki nad nazwę drugiego pliku, tj. `szukaj_maxa.asm`, tak jak pokazano na rysunku.

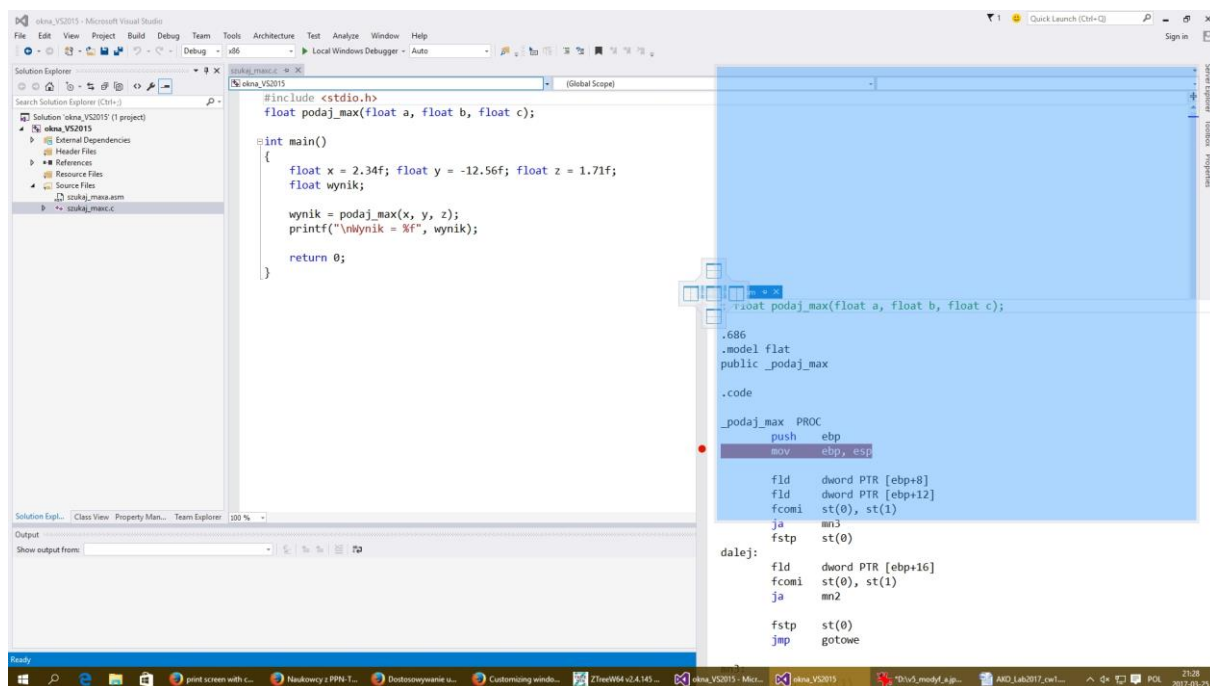


Następnie, naciskając lewy klawisz myszy, rozpoczynamy przeciąganie okna. W tym momencie na ekranie pojawi się charakterystyczny romb (ang. guide diamond = romb przewodnik).



Dalej trzeba przesunąć kursor myszki w taki sposób, by kursor myszki znalazł się w jednym z wyróżnionych pól wewnątrz rombu. Sytuację, gdy kursor został przesunięty wewnątrz rombu na pole po prawej stronie ilustruje poniższy rysunek.





W rezultacie na ekranie w oddzielnych oknach pojawiają się zawartości obu plików źródłowych,

