

Laboratorium Architektury Komputerów i Systemów Operacyjnych

Ćwiczenie 1

Architektura von Neumanna — model programowy komputera

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 wbudowanym*, 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 zerojedynekowych.

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 przekraczających 4 GB.

Architektury procesorów Intel/AMD

Współcześnie, znaczna większość używanych komputerów osobistych posiada zainstalowane procesory rodziny *x86*, która została zapoczątkowana w toku 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.

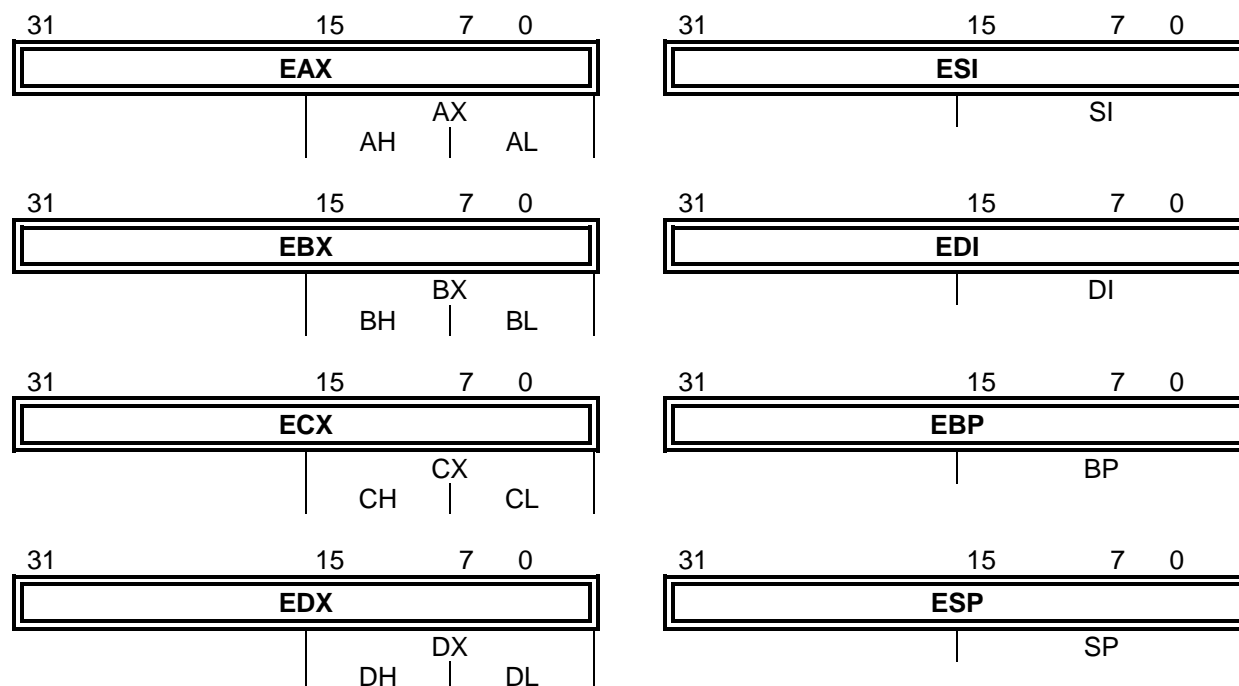
Architektura Intel 64/AMD64 stanowi rozszerzenie stosowanej wcześniej architektury 32-bitowej. Charakterystycznym przykładem są rejestry ogólnego przeznaczenia w procesorach.

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 danych w pamięci głównej 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

	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			

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 — zatem 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.



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 obliczenie na liczbach całkowitych. Jednak procesor rozumie tylko instrukcje zapisane w języku maszynowym w postaci ciągów zer i jedynek. Wprawdzie zapisanie takiego ciągu zerojedynekowego 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 (zerojedynekowy), 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 stanowią 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 12.0. 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 asemblerze dla procesorów architektury Intel32 (lub AMD64/Intel64) 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: w **AL** – jeśli mnożone są liczby 8-bitowe, w **AX** – jeśli mnożone są liczby 16-bitowe, w **EAX** – jeśli mnożone są liczby 32-bitowe. Z tego powodu podaje się tylko jeden argument — mnożnik. Rozmiar mnożnika (8, 16 lub 32 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**, a dla liczb 32-bitowych do **EDX:EAX**.

Samodzielne programy w assemblerze

Zazwyczaj programy napisane w assemblerze stanowią fragmenty dużych aplikacji, w których dominuje kod w języku wysokiego poziomu. Niekiedy jednak tworzymy samodzielne programy w assemblerze. Poniżej podano krótki program przykładowy w assemblerze wraz z opisem sposobu translacji i wykonania. Program wykonuje proste obliczenie na liczbach naturalnych (oblicza sumę wyrazów ciągu $3 + 5 + 7 + 9 + 11$) i wyświetla wynik w postaci liczby dziesiętnej, szesnastkowej i binarnej. Program wykonuje dwukrotnie to samo obliczenie, najpierw bez użycia pętli rozkazowej, następnie za pomocą pętli rozkazowej. Wykonanie tego programu powoduje wyświetlenie na ekranie komputera tekstu:

```

C:\Windows\system32\cmd.exe
EAX = 0000000035 = 00000023H = 0000 0000 0000 0000 0000 0000 0010 0011 B
EAX = 0000000035 = 00000023H = 0000 0000 0000 0000 0000 0000 0010 0011 B
Aby kontynuować, naciśnij dowolny klawisz . . .

```

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

```
.686
```

```
.model flat
```

```
extern wyświetl_EAX : PROC
```

```
extern _ExitProcess@4 : PROC
```

```
public _main
```

```
.code
```

```
_main:
```

```
; 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
```

```
call    wyświetl_EAX ; wyświetlenie wyniku
```

```

; 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 licznika obiegów pętli
    jnz     ptl      ; skok, gdy licznik obiegów różny od 0
    call    wyswietl_EAX ; wyświetlenie wyniku

; zakończenie wykonywania programu
    push    0
    call    _ExitProcess@4
END

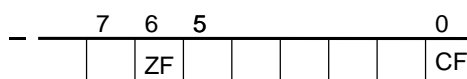
```

Spróbujmy teraz przeanalizować działanie 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.

Rejestr 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 (`call wyswietl_EAX`). W rezultacie na ekranie pojawi się wynik obliczenia.

Pamiętamy jednak, że procesor wykonuje program zapisany w języku maszynowym w postaci ciągów zerojedynekowych. 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. Asembler (lub kompilator) 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`.

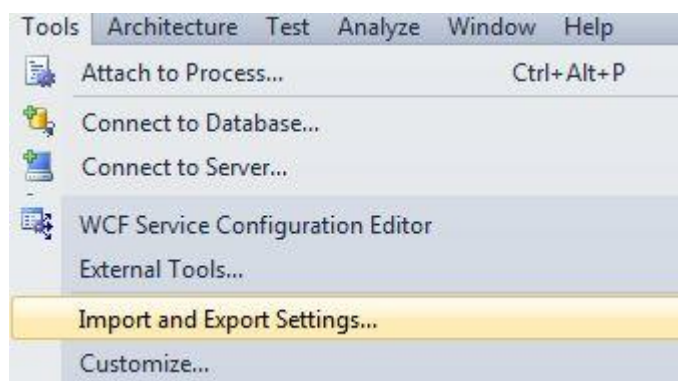
Wywołanie funkcji `ExitProcess` kończy wykonywanie programu i oddaje sterowanie do systemu.

Opisany tu program wprowadzimy teraz do komputera i uruchomimy w środowisku systemu Microsoft Visual Studio 2013.

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

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:**

1. Przy pierwszym uruchomieniu MS Visual Studio może być wymagane podanie typu projektu – wpisujemy wówczas: `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 znacznie 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**.

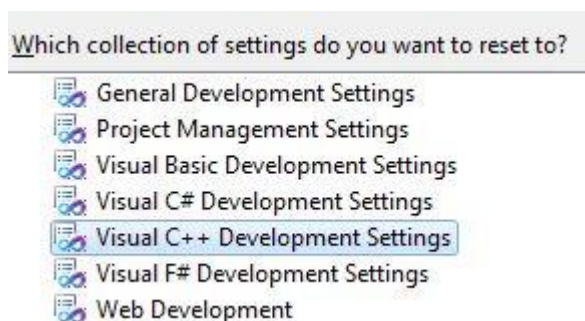
☒ **Reset all settings**

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”.

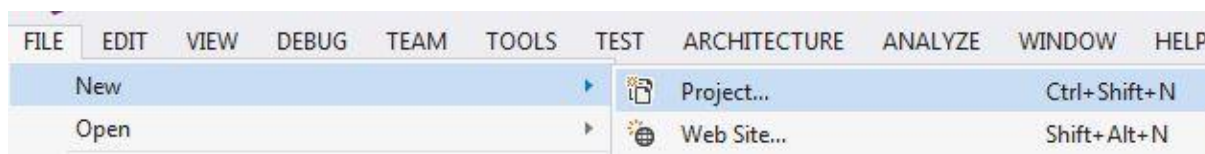
☒ **No, just reset settings, overwriting my current settings**

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

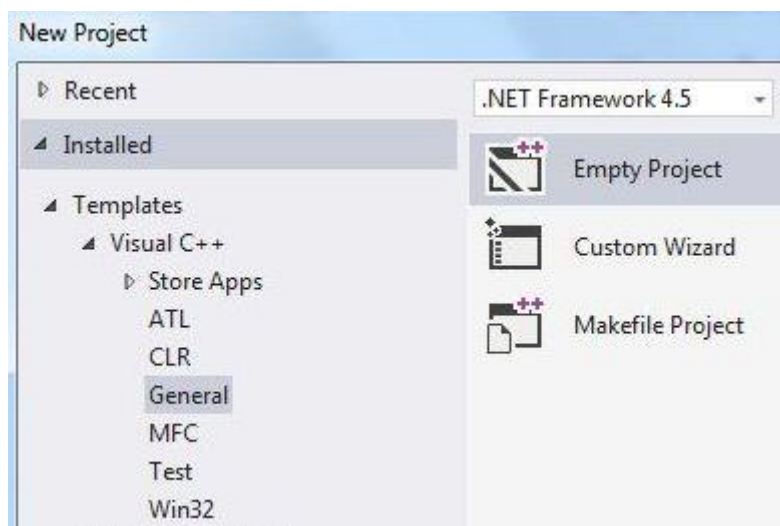


Potem naciskamy kolejno przyciski **Finish** i **Close**.

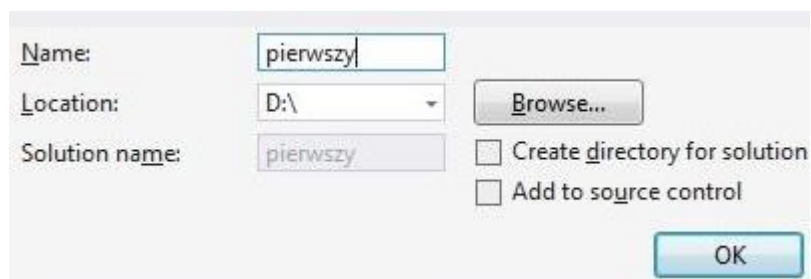
3. Po uruchomieniu MS Visual Studio należy wybrać opcje: **File / New / Project**



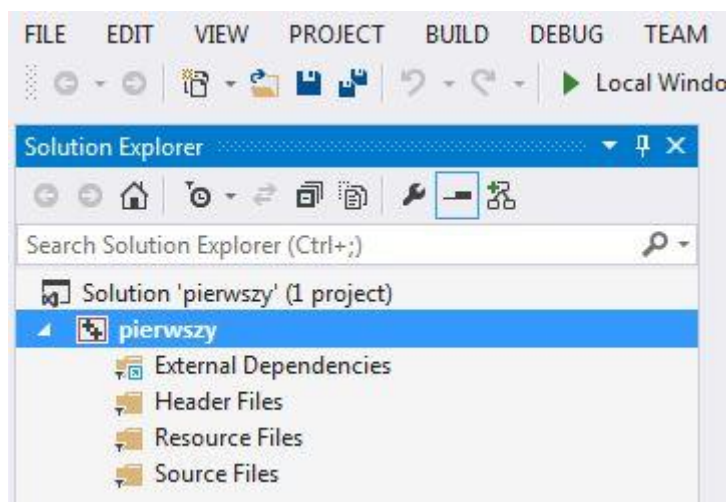
4. W oknie nowego projektu (zob. rys.) określamy najpierw typ projektu poprzez rozwinięcie opcji Visual C++ (z lewej strony okna). Następnie wybieramy opcje General i Empty Project (w środkowym oknie).



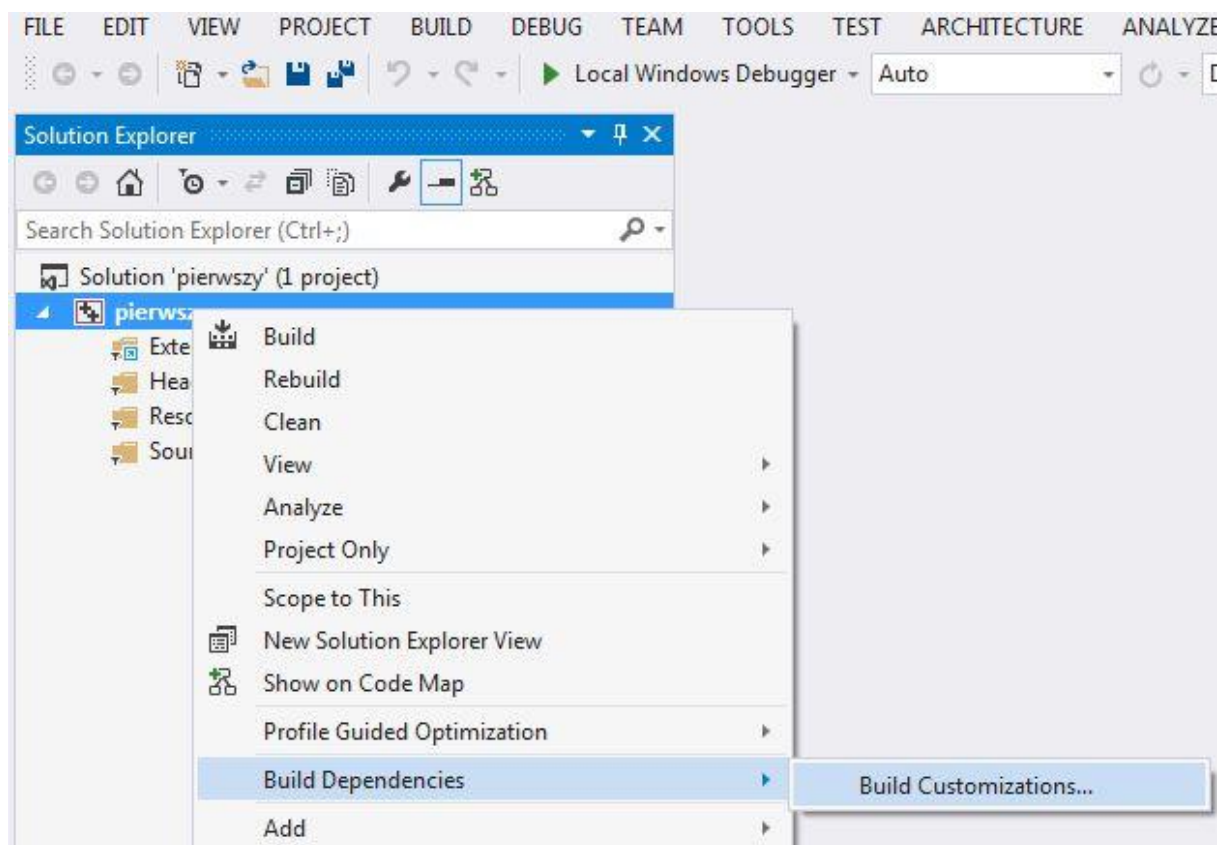
5. Do pola Name wpisujemy nazwę programu (tu: pierwszy) i naciskamy OK. W polu Location powinna znajdować się ścieżka D:\ Znacznik Create directory for solution należy ustawić w stanie nieaktywnym.



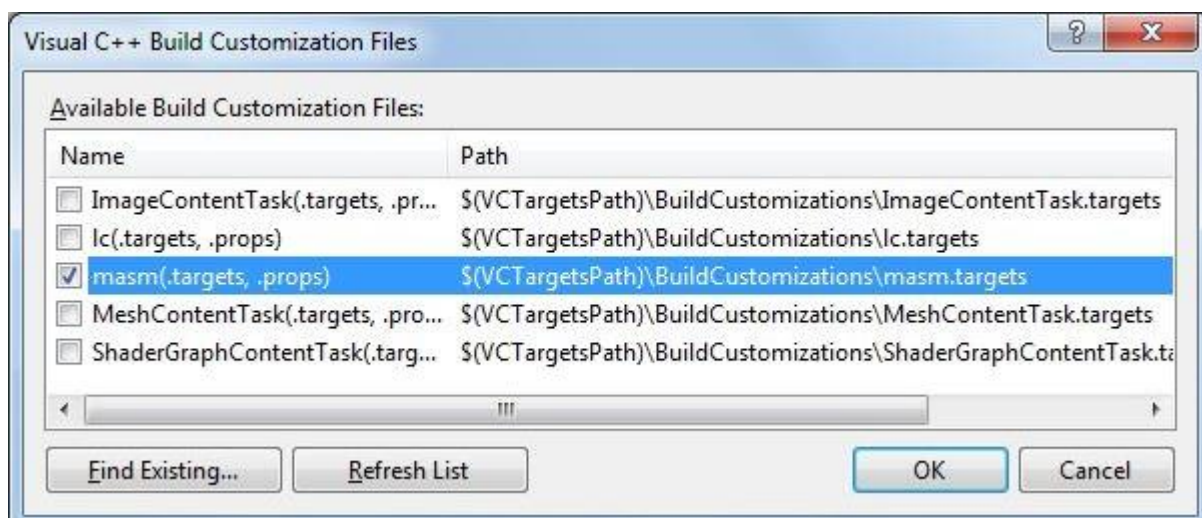
6. W rezultacie wykonania opisanych wyżej operacji pojawi się niżej pokazane okno



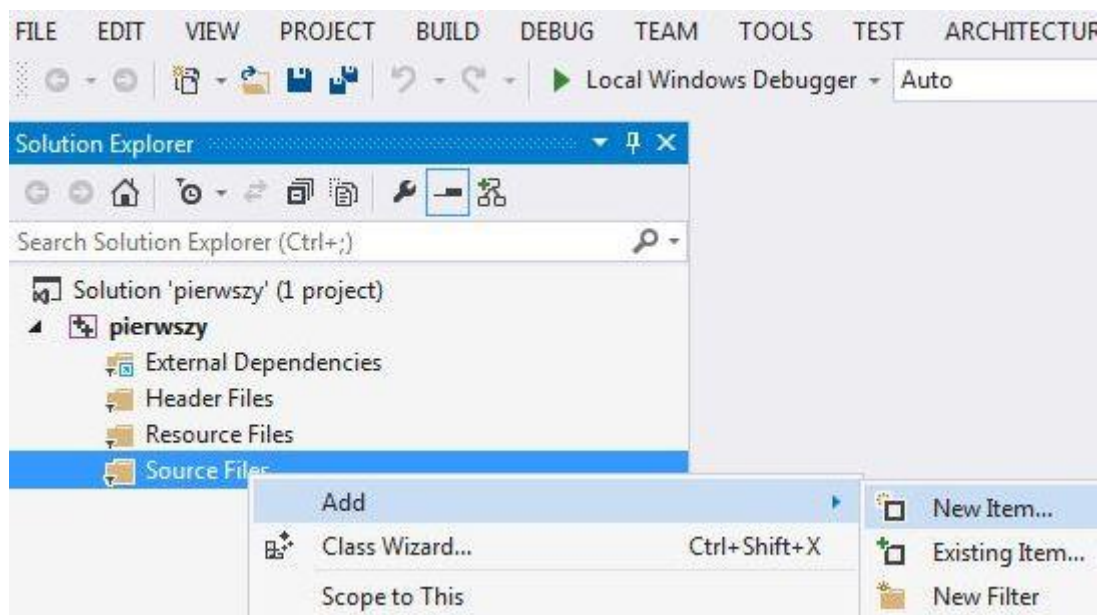
7. Teraz trzeba wybrać odpowiedni asembler. W tym celu należy kliknąć prawym klawiszem myszki na nazwę projektu pierwszy i z rozwijanego menu wybrać opcję Build Dependencies / Build Customizations.



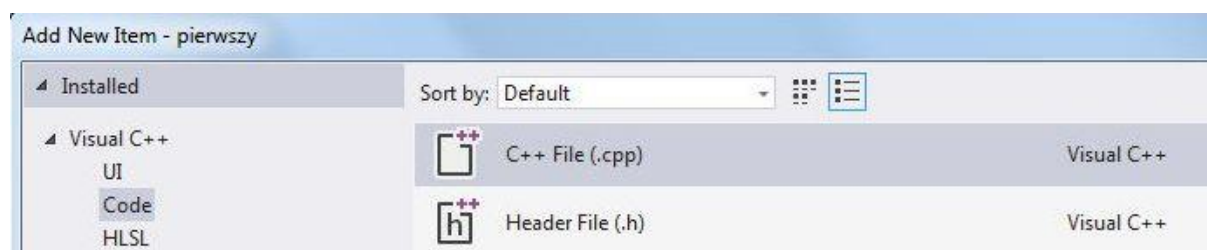
8. W rezultacie na ekranie pojawi się okno (pokazane na poniższym rysunku), w którym należy zaznaczyć pozycję masm i nacisnąć OK.



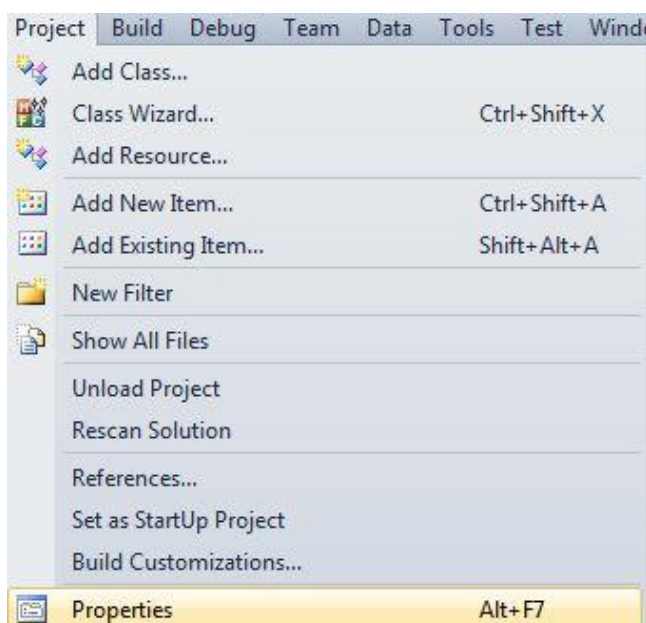
9. Następnie prawym klawiszem myszki należy kliknąć na Source File i wybrać opcję Add / New Item.



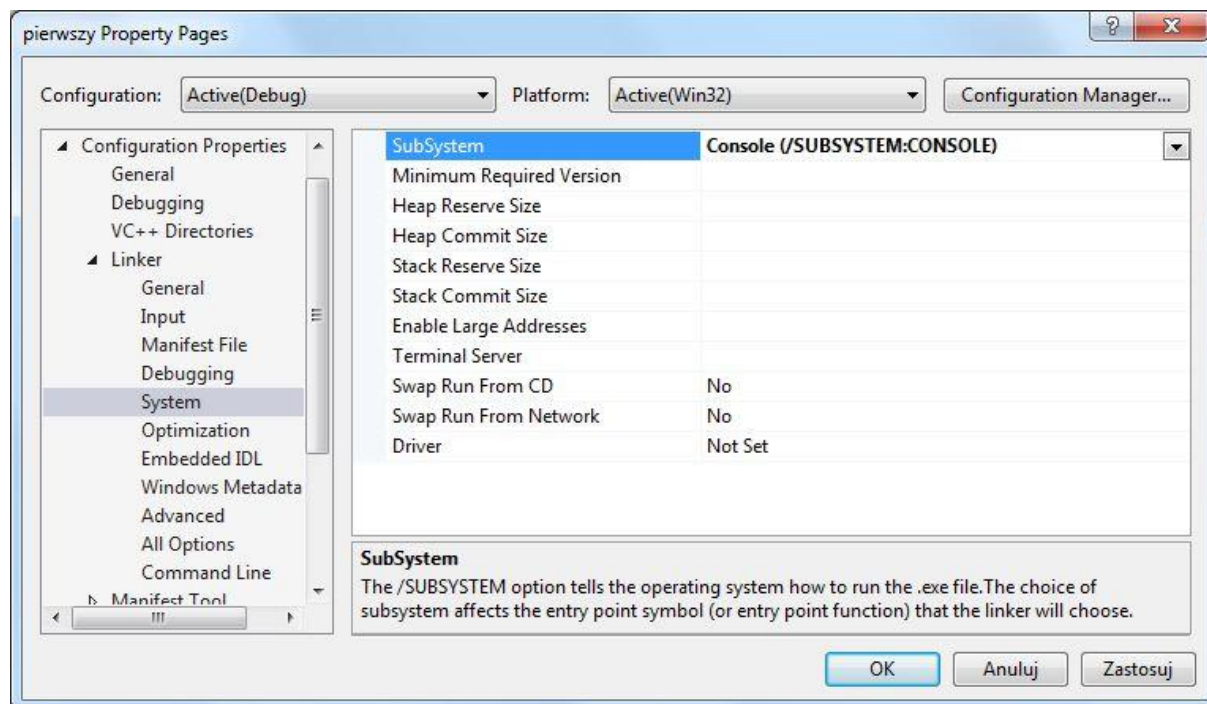
10. W ślad za tym pojawi się kolejne okno, w którym w polu **Installed (Visual C++)** należy zaznaczyć opcję **Code**, a w polu **Name** wpisać nazwę pliku zawierającego programu źródłowy, np. `cw1.asm`. Naciskamy przycisk **Add**.



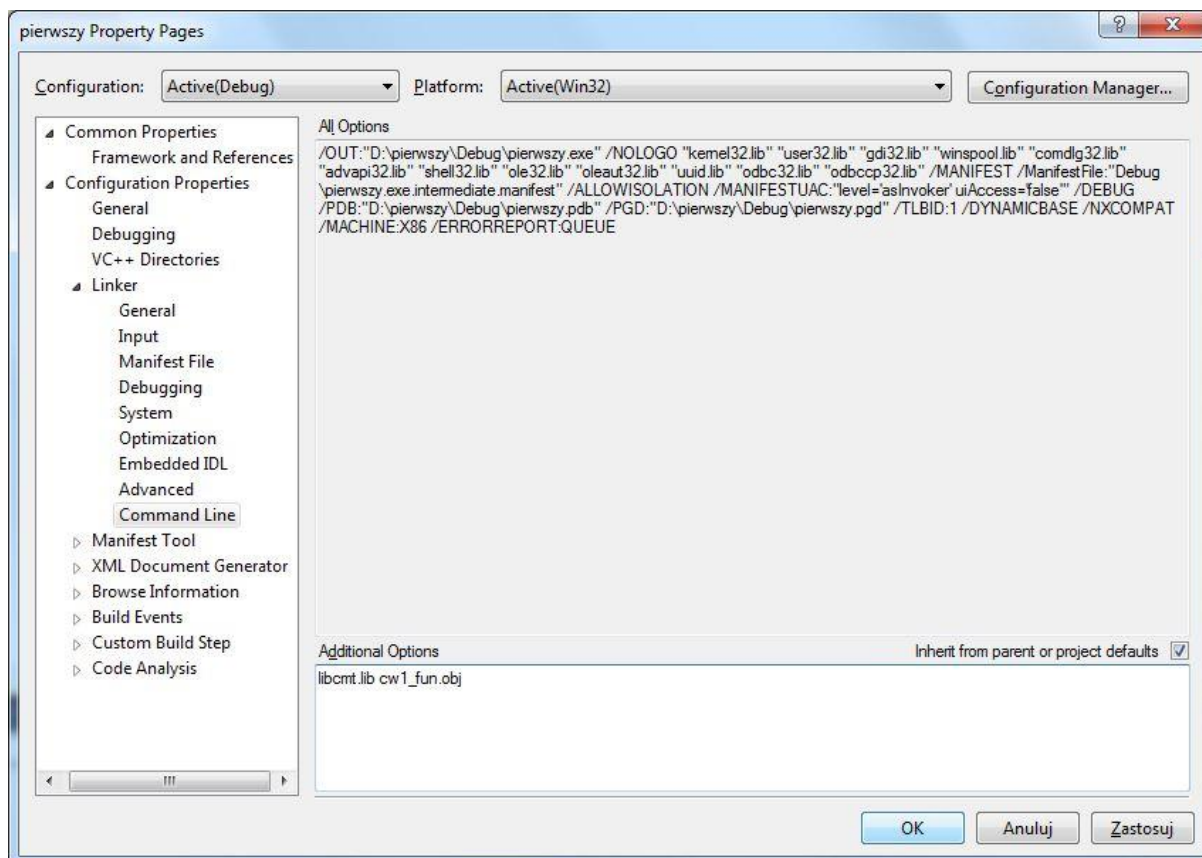
11. W kolejnym kroku należy uzupełnić ustawienia konsolidatora (linkera). W tym celu należy z menu głównego wybrać **Project** i z rozwijanego menu wybrać opcję **Properties**.



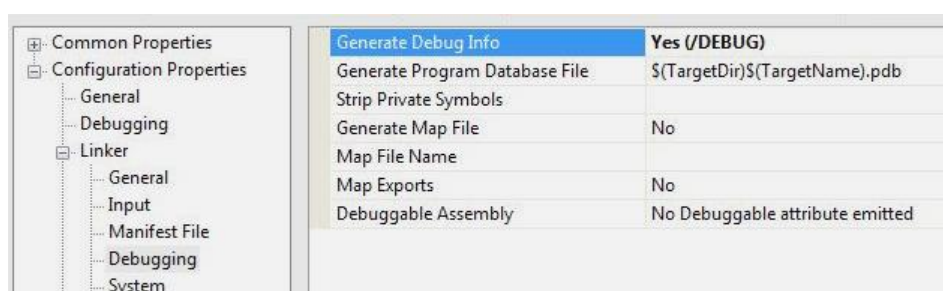
12. W ramach pozycji Linker należy ustawić także typ aplikacji. W tym celu zaznaczamy grupę System (zob. rysunek) i w polu SubSystem wybieramy opcję Console (/SUBSYSTEM:CONSOLE).



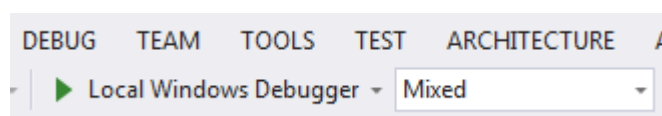
13. Rozwijamy menu konsolidatora (pozycja Linker) i wybieramy opcję Command line. Następnie w polu Additional options podajemy nazwę biblioteki `libcmt.lib` i pliku pomocniczego `cw1_fun.obj` (w pliku tym zawarty jest kod podprogramu `wyswietl_EAX`, za pomocą którego wyświetlamy wyniki podprogramu).



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**.

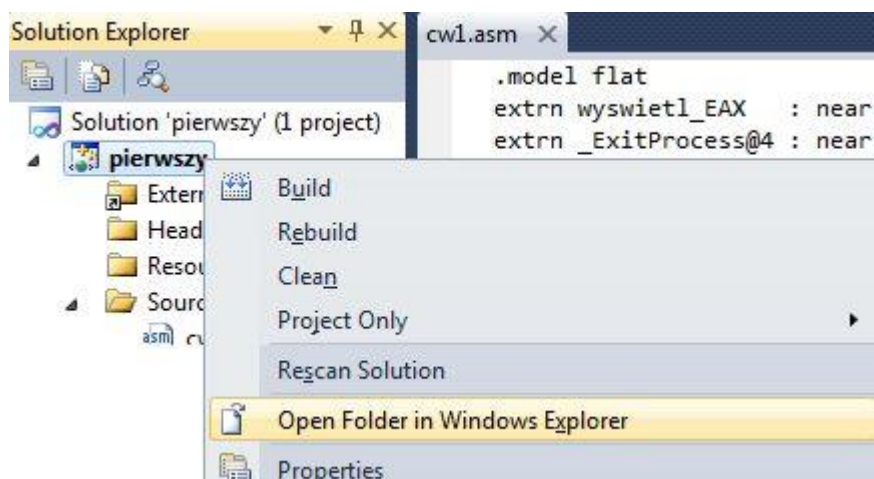


Ponadto w górnej części okna Visual Studio należy ustawić opcję debuggowania „Mixed” tak jak pokazano na poniższym rysunku.

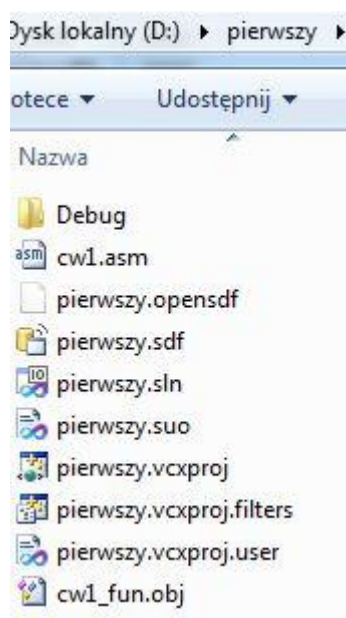


15. Po wykonaniu opisanych czynności przygotowawczych do okna „cw1.asm” należy skopiować kod źródłowy programu (podany na stronie 7/8) i nacisnąć **Ctrl S**.
16. Następnie, do katalogu zawierającego omawiany projekt **pierwszy** należy skopiować plik **cw1_fun.obj** (plik ten udostępniany jest na serwerze `\\mkz14\public\AKiSO`

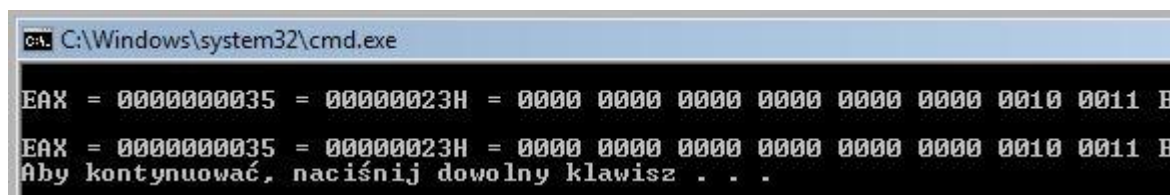
wraz z plikiem zawierającym niniejszą instrukcję). Katalog ten można łatwo otworzyć poprzez kliknięcie prawym klawiszem myszki w oknie Solution Explorer na nazwę projektu pierwszy i wybranie opcji Open Folder in File Explorer, tak pokazuje poniższy rysunek.



17. W rezultacie zostanie wyświetlone okno przedstawiające zawartość katalogu skojarzonego z projektem pierwszy. Do katalogu tego należy skopiować plik cw1_fun.obj — zawartość katalogu po wykonaniu kopiowania pokazana jest na poniższym rysunku.



18. W celu wykonania asemblacji i konsolidacji programu wybrać opcję Build / Build Solution.
19. 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.



```

C:\Windows\system32\cmd.exe
EAX = 0000000035 = 00000023H = 0000 0000 0000 0000 0000 0000 0010 0011 B
EAX = 0000000035 = 00000023H = 0000 0000 0000 0000 0000 0000 0010 0011 B
Aby kontynuować, naciśnij dowolny klawisz . . .

```

Usuwanie błędów formalnych

Często program poddany asemblacji wykazuje błędy, podawane przez asembler. 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 asemblacji sygnalizowany był błąd w postaci:

```
1>cw1.asm(34): error A2070: invalid instruction operands
```

to jego odnalezienie nie przedstawia większych trudności. W wierszu 34 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
```

Śledzenie programów w środowisku MS Visual Studio C++

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). *Debugery* 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.

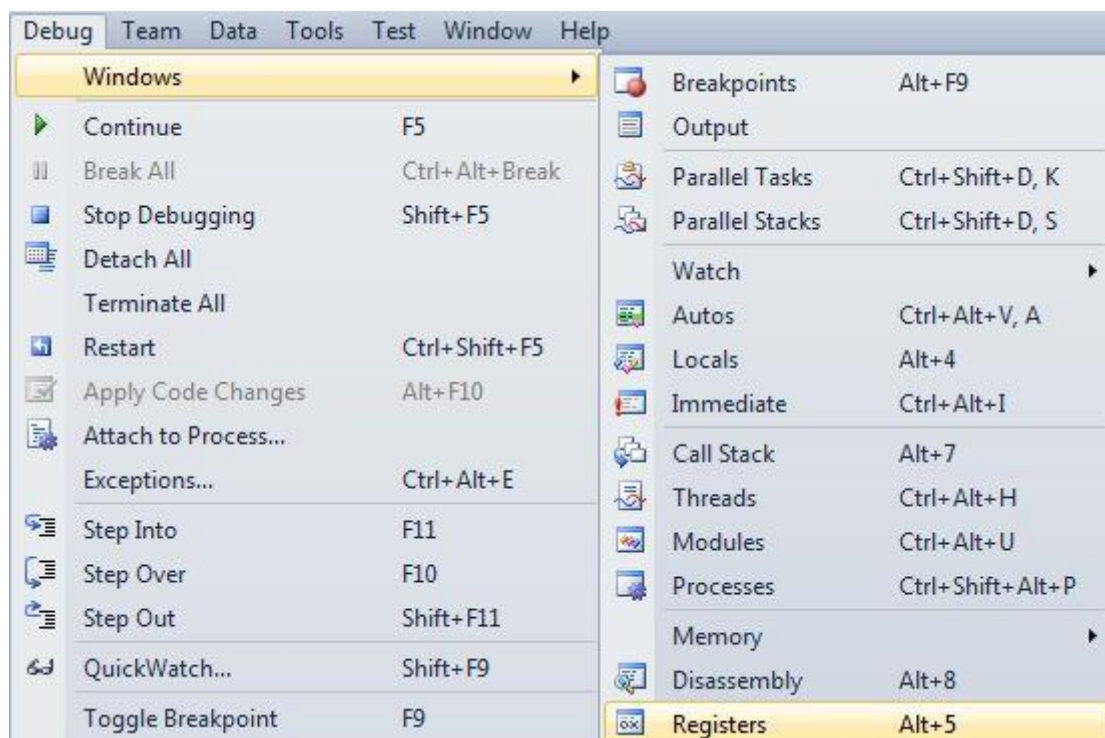
W systemie Microsoft Visual Studio debuggowanie programu jest wykonywane po naciśnięciu klawisza F5. Przedtem należy ustawić punkt zatrzymania (ang. breakpoint) poprzez kliknięcie na obrzeżu ramki obok rozkazu, przed którym ma nastąpić zatrzymanie. Po uruchomieniu *debuggowania*, można otworzyć potrzebne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

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. Podobne znaczenie ma klawisz F11, ale w tym przypadku śledzenie obejmuje także zawartość podprogramów.

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

W opisanych dalej *Zadaniach do wykonania* trzeba wielokrotnie odczytywać zawartość znaczników 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 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).



Wyświetlanie zawartości stosu

Otworzyć okno **Memory** i skopiować zawartość rejestru ESP (z okna **Registers**) do pola adresowego w oknie **Memory**. Przed skopiowaną liczbą dopisać 0x (liczba szesnastkowa w stylu języka C).

Wskazówki praktyczne dotyczące konfiguracji środowiska Microsoft Visual Studio

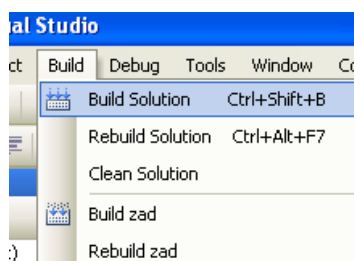
Opisane w poprzedniej części zasady uruchamiania programów w środowisku zintegrowanym Microsoft Visual Studio dotyczą środowiska w konfiguracji standardowej, tj. w postaci bezpośrednio po instalacji systemu. Jednak użytkownicy MS Visual Studio mogą dokonywać zmian konfiguracji, które mają charakter trwały. Przykładowo, może być zmienione znaczenie klawisza F7, który uruchamia kompilację i konsolidację programu. W tej sytuacji podane wcześniej reguły postępowania muszą być częściowo zmodyfikowane. W dalszej rozpatrzymy typowe przypadki postępowania.

1. 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*.

2. Kompilacja i konsolidacja programu

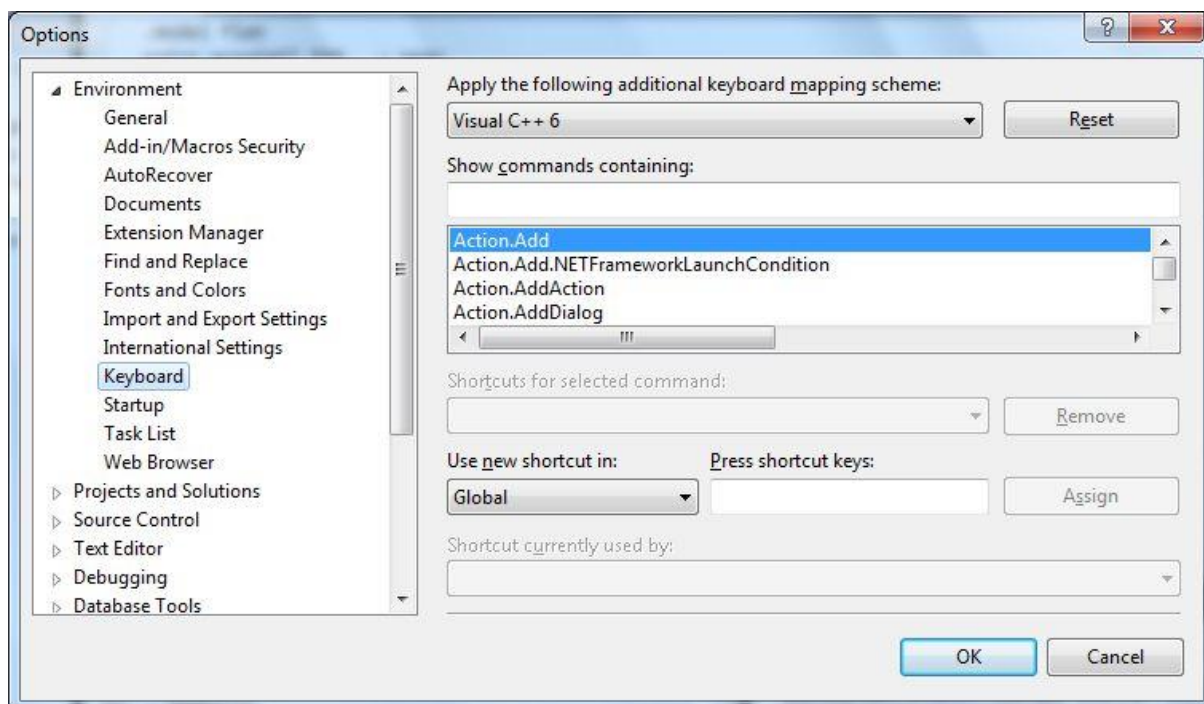
Kompilacja i konsolidacja programu w wersji standardowej następuje po naciśnięciu klawisza F7. W niektórych konfiguracjach zamiast klawisza F7 używa się kombinacji klawiszy Ctrl Shift B (zob. rys.). W takim przypadku w celu wykonania kompilacji (asemblacji) programu i konsolidacji należy wybrać z menu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



3. 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.

4. 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 zachowanie 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 Customization. Dalej należy postępować tak jak opisano w pkt. 6 i 7 na str. 12. 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.

5. 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.).



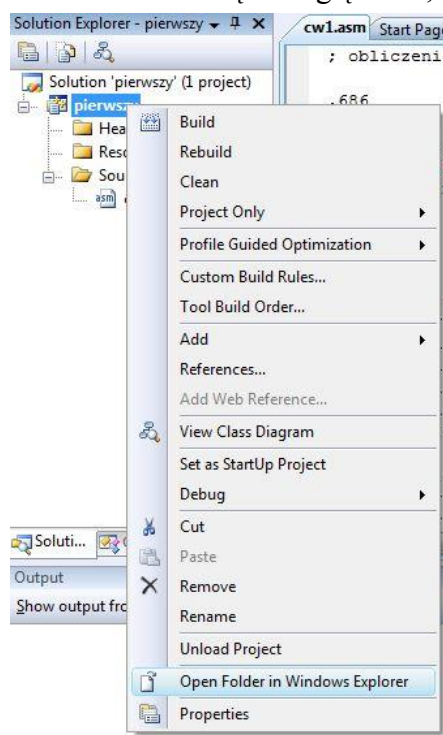
Z kolei, po zaznaczeniu opcji **Command Line** w prawej części okna, na ekranie pojawi się aktualna lista opcji (**All options:**) asemblera `ml.exe` (znak `/` przed opcją ma takie samo znaczenie jak znak `-`). Lista ta wyświetlana jest na szarym tle i może być zmieniona tylko poprzez zmiany opcji asemlacji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

6. Kompilacja (asemlacja) i konsolidacja w przypadku znacznych zmian w konfiguracji

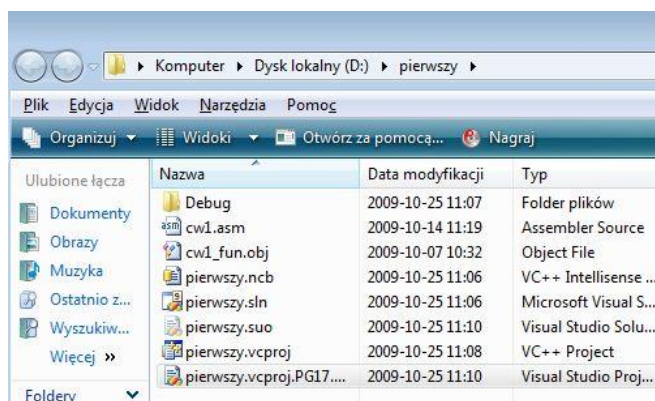
Niekiedy zmiany konfiguracyjne dokonane przez poprzednich użytkowników MS Visual Studio są tak głębokie, że nawet trudno rozpocząć pracę w systemie. W takim przypadku można wybrać opcję **Window / Reset Window Layout** i potwierdzić wybór za pomocą przycisku **Tak**. W rezultacie zostanie przywrócony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



7. Wyświetlanie listy plików programu

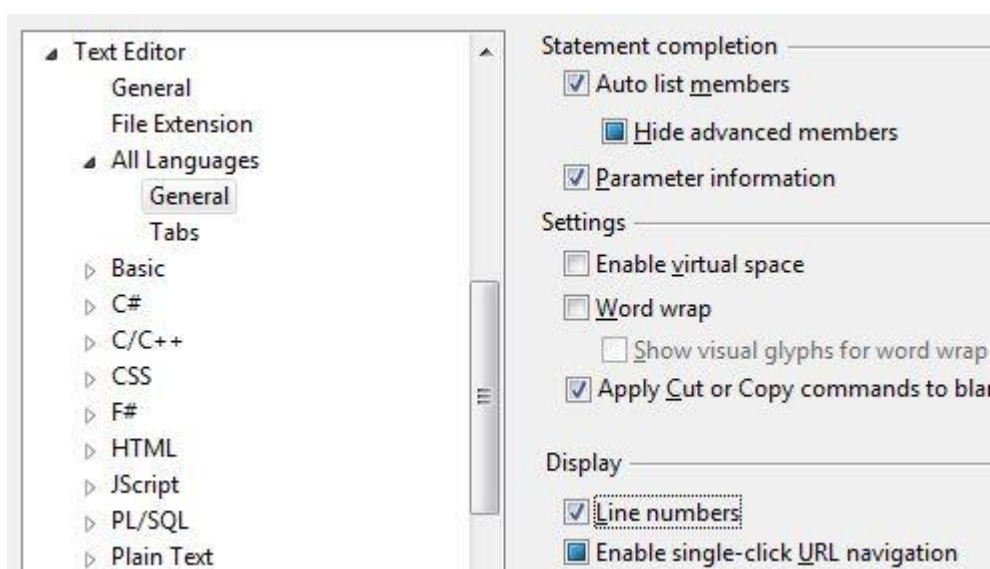
W niektórych przypadkach potrzebne są informacje o położeniu plików składających się na cały program. Można wówczas w oknie **Solution Explorer** kliknąć prawym klawiszem myszki na nazwę projektu

(tu: pierwszy) i wybrać opcję Open Folder in Windows Explorer (tak jak pokazano na rysunku obok). W rezultacie zostanie wyświetlona lista plików uruchamianej aplikacji — przykładowa postać podana jest poniżej.



8. 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.



Zadania do wykonania

1. Przetłumaczyć i uruchomić program przykładowy podany w niniejszej instrukcji.
2. Wprowadzić do programu punkt zatrzymania (breakpoint) obok pierwszego rozkazu programu (`mov eax, 3`), następnie uruchomić program za pomocą *debuggera* (klawisz F5).
3. Po zatrzymaniu programu otworzyć okno **Registers**.
4. Wyznaczyć adres komórki pamięci, w której znajduje się pierwszy rozkaz programu (`mov eax, 3`).
5. Nacisnąć klawisz F10 w celu wykonania podanego wyżej rozkazu. Ile bajtów w pamięci komputera zajmuje ten rozkaz? Ile wynosi zawartość rejestru EAX po wykonaniu rozkazu?
6. Nacisnąć klawisz F10 w celu wykonania kolejnego rozkazu (`add eax, 5`). Ile bajtów w pamięci komputera zajmuje ten rozkaz? Ile wynosi zawartość rejestru EAX po wykonaniu rozkazu?
7. Powtórzyć działania opisane w pkt. 6 dla trzech kolejnych rozkazów programu.
8. Wykonanie rozkazu `call wyswietl_EAX` spowoduje wyświetlenie na ekranie zawartości rejestru EAX. Określić także ile bajtów w pamięci zajmuje ten rozkaz.
9. Określić położenie w pamięci rozkazu `add eax, ebx` w drugim fragmencie programu.
10. W trakcie wykonywania drugiego fragmentu programu istotne znaczenie ma stan rejestru znaczników (w debuggerze oznaczonego symbolem EFL), a w szczególności znaczników ZF i CF. Określić stan tych znaczników po wykonaniu rozkazu `add eax, ebx`.
11. Poprzez wielokrotne naciskanie klawisza F10 wykonać cały program rejestrując na kartce w każdym obiegu pętli zawartości rejestrów EAX, EBX, ECX oraz znaczników ZF i CF.
12. Wyjaśnić jak zmienia się zawartość rejestru EIP po wykonaniu rozkazu `jnz`.
13. Wprowadzić do programu rozkaz `mov edi, 4294967295` i następnie za pomocą rozkazu `add` zwiększyć liczbę w rejestrze `edi` o 1. Ile wynosi wynik dodawania? Jaka wartość przyjął znacznik CF?
14. Dopisać do programu przykładowego trzeci fragment, w którym zostanie obliczona suma wyrazów innego ciągu, np. $7 + 14 + 28 + 56 + 112 + \dots$