

Praktyczny kurs

# ASEMBLERA

Wydanie II

**Zobacz, na co Cię stać  
z asemblerem!**

- Dowiedz się,  
do czego może Ci się  
przydać asembler
- Poznaj  
architekturę i sposób działania  
procesorów Intel
- Naucz się  
pisać wydajne programy  
dla systemów DOS i Windows

Eugeniusz Wróbel



Helion

## » Idź do

- Spis treści
- Przykładowy rozdział
- Skorowidz

## » Katalog książek

- Katalog online
- Zamów drukowany katalog

## » Twój koszyk

- Dodaj do koszyka

## » Cennik i informacje

- Zamów informacje o nowościach
- Zamów cennik

## » Czytelnia

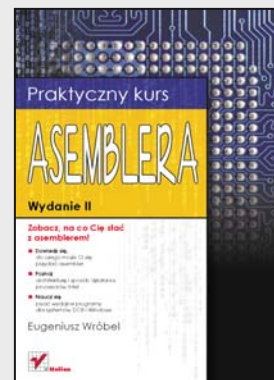
- Fragmenty książek online

## » Kontakt

Helion SA  
ul. Kościuszki 1c  
44-100 Gliwice  
tel. 32 230 98 63  
e-mail: helion@helion.pl  
© Helion 1991–2011

## Praktyczny kurs assemblera. Wydanie II

Autor: Eugeniusz Wróbel  
ISBN: 978-83-246-2732-5  
Format: 158×235, stron: 424



- Dowiedz się, do czego może Ci się przydać assembler
- Poznaj architekturę i sposób działania procesorów Intel
- Naucz się pisać wydajne programy dla systemów DOS i Windows

### Zobacz, na co Cię stać z assemblerem!

Programowanie w języku niskiego poziomu – choć czasem nieco uciążliwe – daje bardzo dużą swobodę w kwestii wykorzystania sprzętowych zasobów komputera i oferuje niemal nieograniczoną kontrolę nad sposobem działania programu. Aplikacje napisane za pomocą assemblera są bardzo szybkie i wydajne, a ponadto wymagają o wiele mniejszej ilości pamięci operacyjnej niż analogiczny kod, opracowany w językach wysokiego poziomu, takich jak C++, Java czy Visual Basic. Jeśli jesteś zainteresowany poszerzeniem swoich umiejętności programistycznych, z pewnością nadszedł czas, aby sięgnąć po assembler.

Książka „Praktyczny kurs assemblera. Wydanie II” wprowadzi Cię w podstawowe zagadnienia związane z zastosowaniem języka niskiego poziomu do programowania komputerów opartych na architekturze x86-32 procesorów Intel (oraz AMD). Poznasz sposoby wykorzystania zasobów sprzętowych, zasadę działania procesora i listę jego instrukcji. Nauczysz się też, jak używać różnych trybów adresowania w celu optymalnego zarządzania zawartością rejestrów i pamięci. Dowiesz się, jak prawidłowo pisać, łączyć, kompilować i uruchamiać programy, a także poznasz praktyczne przykłady zastosowania assemblera.

- Podstawowe informacje na temat assemblera i architektury x86-32 procesorów Intel (oraz AMD)
- Przegląd narzędzi przydatnych przy tworzeniu i uruchamianiu kodu
- Sposoby adresowania pamięci i korzystanie z rejestrów procesora
- Lista instrukcji procesorów o architekturze x86-32
- Definiowanie i używanie zmiennych
- Tworzenie podprogramów i makroinstrukcji
- Korzystanie z funkcji systemu MS DOS i BIOS-a oraz windowsowych bibliotek typu API
- Stosowanie assemblera do tworzenia programów uruchamianych pod systemem Windows
- Tworzenie assemblerowych bibliotek typu dll z wykorzystaniem środowiska Microsoft Visual Studio
- Przegląd metod optymalizacji kodu
- Praktyczne przykłady programów wykorzystujących język assemblera

### Wykorzystaj w pełni potencjał assemblera!

# Spis treści

	<b>Ostatni wykład Eugeniusza Wróbla .....</b>	<b>7</b>
	<b>Wprowadzenie do drugiego wydania .....</b>	<b>9</b>
<b>Rozdział 1.</b>	<b>Wprowadzenie .....</b>	<b>11</b>
	1.1. Co to jest asembler? .....	11
	1.2. Dlaczego programować w języku asemblera? .....	14
	1.3. Dlaczego warto poznać język asemblera? .....	16
	1.4. Wymagane umiejętności .....	16
	1.5. Konwencje stosowane w książce .....	17
<b>Rozdział 2.</b>	<b>Pierwszy program w asemblerze .....</b>	<b>21</b>
	2.1. „Hello, world!” pod kontrolą systemu operacyjnego MS DOS .....	22
	2.2. „Hello, world!” pod kontrolą systemu operacyjnego Windows .....	25
<b>Rozdział 3.</b>	<b>Architektura procesorów rodziny x86-32 widziana oczami programisty .....</b>	<b>33</b>
	3.1. Rejestry procesora 8086 .....	34
	3.2. Zwiększamy rozmiar rejestrów — od procesora 80386 do Intel Core i7 .....	38
	3.3. Zwiększamy liczbę rejestrów — od procesora i486 do Intel Core i7 .....	39
	3.4. Segmentowa organizacja pamięci .....	44
	3.5. Adresowanie argumentów .....	48
	3.6. Adresowanie argumentów w pamięci operacyjnej .....	49
	3.7. Architektura x86-32e .....	52
<b>Rozdział 4.</b>	<b>Narzędzia .....</b>	<b>55</b>
	4.1. Asembler MASM .....	56
	4.2. Program konsolidujący — linker .....	60
	4.3. Programy uruchomieniowe .....	62
	Microsoft CodeView .....	64
	Microsoft WinDbg .....	67
	OllyDbg .....	68
	4.4. Środowiska zintegrowane .....	70
	Microsoft Programmer’s WorkBench (PWB) .....	70
	Środowisko zintegrowane MASM32 SDK .....	71
	Środowisko zintegrowane RadASM .....	74
	WinAsm Studio .....	74
	4.5. Microsoft Visual Studio .....	75

<b>Rozdział 5.</b>	<b>Lista instrukcji procesorów x86-32 .....</b>	<b>81</b>
5.1.	Instrukcje ogólne — jednostki stałoprzecinkowej .....	84
5.2.	Koprocesor arytmetyczny — instrukcje jednostki zmiennoprzecinkowej .....	87
5.3.	Instrukcje rozszerzenia MMX .....	90
5.4.	Instrukcje rozszerzenia SSE .....	93
5.5.	Instrukcje rozszerzenia SSE2 .....	97
5.6.	Instrukcje rozszerzenia SSE3, SSSE3 oraz SSE4 .....	100
5.7.	Instrukcje systemowe .....	101
5.8.	Planowane rozszerzenie AVX .....	102
<b>Rozdział 6.</b>	<b>Ogólna struktura programu asemblerowego .....</b>	<b>105</b>
6.1.	Uprozczone dyrektywy definiujące segmenty .....	105
6.2.	Pełne dyrektywy definiowania segmentów .....	111
6.3.	Dyrektywy pomocnicze .....	114
<b>Rozdział 7.</b>	<b>Definiowanie i stosowanie zmiennych .....</b>	<b>123</b>
7.1.	Zmienne całkowite .....	124
7.2.	Zmienne zmiennoprzecinkowe .....	127
7.3.	Definiowanie tablic i łańcuchów .....	128
7.4.	Struktury zmiennych .....	130
7.5.	Dyrektywa definiująca pola bitowe .....	133
<b>Rozdział 8.</b>	<b>Podprogramy .....</b>	<b>137</b>
8.1.	Stos .....	137
8.2.	Wywołanie i organizacja prostych podprogramów .....	140
8.3.	Dyrektywa PROC – ENDP .....	141
8.4.	Parametry wywołania podprogramu .....	146
8.5.	Zmienne lokalne .....	155
<b>Rozdział 9.</b>	<b>Makroinstrukcje oraz dyrektywy asemblacji warunkowej .....</b>	<b>157</b>
9.1.	Makroinstrukcja definiowana .....	157
9.2.	Dyrektywa LOCAL .....	162
9.3.	Dyrektywy asemblacji warunkowej .....	163
9.4.	Makroinstrukcje niedefiniowane .....	166
9.5.	Makroinstrukcje tekstowe .....	167
9.6.	Makroinstrukcje operujące na łańcuchach (na tekstach) .....	168
<b>Rozdział 10.</b>	<b>Funkcje systemu MS DOS oraz BIOS .....</b>	<b>171</b>
10.1.	Co ma prawo przerwać wykonanie naszego programu? .....	171
10.2.	Obsługa klawiatury oraz funkcje grafiki na poziomie BIOS .....	174
10.3.	Wywoływanie podprogramów systemu operacyjnego MS DOS .....	180
<b>Rozdział 11.</b>	<b>Programowanie w asemblerze w środowisku Windows .....</b>	<b>187</b>
11.1.	Systemowe programy biblioteczne .....	188
11.2.	Pierwsze okno .....	191
11.3.	Struktury programowe typu HLL .....	197
11.4.	Program generatora okien Prostart .....	199
<b>Rozdział 12.</b>	<b>Wybrane zagadnienia optymalizacji programu .....</b>	<b>207</b>
12.1.	Kiedy i co powinniśmy optymalizować w programie? .....	209
12.2.	Optymalizujemy program przygotowany dla procesora x86-32 .....	211
	Modele pamięci — mieszanie kodów 16- i 32-bitowych .....	211
	Wyrównywanie danych .....	212
	Pamięć podręczna .....	213
	Unikanie rozgałęzień (skoków) .....	215
	Opóźnienia wynikające z pierwszego wykonania oraz rozwijanie pętli .....	216
	Opóźnienia związane z zapisywaniem i odczytywaniem .....	217

12.3.	Wspieramy proces optymalizacji za pomocą programu Vtune .....	218
12.4.	Na ile różnych sposobów możemy zakodować kopiowanie tablic? .....	219
	Metoda 1.: Z wykorzystaniem instrukcji MOVSB .....	221
	Metoda 2.: Z wykorzystaniem instrukcji MOVSD .....	221
	Metoda 3.: Jawna pętla z instrukcjami MOV .....	222
	Metoda 4.: Pętla z instrukcją MOV, rozwinięta .....	222
	Metoda 5.: Pętla rozwinięta, grupowanie operacji odczytu i zapisu .....	223
	Metoda 6.: Wykorzystujemy rejestry MMX .....	223
	Metoda 7.: Modyfikujemy metodę 6., stosując instrukcje MOVNTQ i SFENCE .....	224
	Metoda 8.: Na początku pętli z poprzedniej metody wprowadzamy instrukcję pobrania wstępnego do pamięci podręcznej .....	225
	Metoda 9.: Wykorzystujemy 128-bitowe rejestry rozszerzenia SSE .....	225
<b>Rozdział 13.</b>	<b>Podział programu na moduły i łączenie modułów zakodowanych w różnych językach programowania .....</b>	<b>227</b>
13.1.	Jak realizować połączenia międzymodułowe? .....	228
13.2.	Mieszamy moduły przygotowane w różnych językach .....	232
<b>Rozdział 14.</b>	<b>Tworzenie projektu assemblerowego w środowisku Microsoft Visual Studio .....</b>	<b>239</b>
14.1.	Wstawki assemblerowe w programie uruchamianym w języku C++ .....	239
14.2.	Assemblerowa biblioteka dll w środowisku Microsoft Visual Studio .....	245
<b>Rozdział 15.</b>	<b>Przykładowe programy dla systemu operacyjnego MS DOS .....</b>	<b>251</b>
15.1.	Pierwsze kroki w prostym trybie graficznym .....	252
15.2.	Pozorujemy głębię .....	255
15.3.	Generowanie fraktali .....	258
<b>Rozdział 16.</b>	<b>Przykładowe programy dla systemu operacyjnego Windows .....</b>	<b>265</b>
16.1.	Zegarek .....	265
16.2.	Wykorzystanie biblioteki OpenGL .....	270
16.3.	Prosty edytor graficzny .....	273
<b>Rozdział 17.</b>	<b>Biblioteki assemblerowe w środowisku Microsoft Visual Studio ...</b>	<b>293</b>
17.1.	Tworzenie projektu assemblerowego dla środowiska Visual Studio 2008 .....	293
17.2.	Szyfrowanie .....	301
17.3.	Edytor graficzny .....	307
17.4.	Steganografia .....	312
<b>Załącznik 1.</b>	<b>Interesujące strony w internecie .....</b>	<b>317</b>
<b>Załącznik 2.</b>	<b>Lista dyrektyw i pseudoinstrukcji języka MASM .....</b>	<b>319</b>
Z2.1.	Dyrektywy określające listę instrukcji procesora .....	319
Z2.2.	Organizacja segmentów .....	321
Z2.3.	Definiowanie stałych oraz dyrektywy związane z nazwami symbolicznymi .....	323
Z2.4.	Definiowanie zmiennych .....	324
Z2.5.	Dyrektywy asemlacji warunkowej .....	326
Z2.6.	Makroinstrukcje i dyrektywy z nimi związane .....	327
Z2.7.	Pseudoinstrukcje typu HLL .....	329
Z2.8.	Dyrektywy związane z podprogramami .....	329
Z2.9.	Dyrektywy wpływające na kształt listingu asemlacji .....	330
Z2.10.	Połączenia międzymodułowe .....	332
Z2.11.	Dyrektywy związane z diagnostyką procesu asemlacji .....	333
Z2.12.	Inne dyrektywy i pseudoinstrukcje .....	334

<b>Załącznik 3. Operatory stosowane w języku MASM .....</b>	<b>337</b>
Z3.1. Operatory stosowane w wyrażeniach obliczanych w czasie asemblacji .....	337
Z3.2. Operatory stosowane w wyrażeniach obliczanych w czasie wykonywania programu .....	341
<b>Załącznik 4. Symbole predefiniowane .....</b>	<b>343</b>
<b>Załącznik 5. Przegląd instrukcji procesora x86-32 .....</b>	<b>347</b>
Z5.1. Instrukcje ogólne (jednostki stałoprzecinkowej) .....	347
Z5.2. Instrukcje jednostki zmiennoprzecinkowej (koprocesora arytmetycznego) ....	354
Z5.3. Instrukcje rozszerzenia MMX .....	357
Z5.4. Instrukcje rozszerzenia SSE .....	360
Z5.5. Instrukcje rozszerzenia SSE2 .....	363
Z5.6. Instrukcje rozszerzenia SSE3 .....	367
Z5.7. Instrukcje systemowe .....	368
<b>Załącznik 6. Opis wybranych przerw systemu BIOS .....</b>	<b>371</b>
Z6.1. Funkcje obsługi klawiatury wywoływane przerwaniem programowym INT 16h .....	371
Z6.2. Funkcje obsługi karty graficznej wywoływane przerwaniem programowym INT 10h .....	373
<b>Załącznik 7. Wywołania funkcji systemu operacyjnego MS DOS .....</b>	<b>379</b>
Z7.1. Funkcje realizujące odczyt lub zapis znaku z układu wejściowego lub wyjściowego .....	379
Z7.2. Funkcje operujące na katalogach .....	381
Z7.3. Operacje na dysku .....	381
Z7.4. Operacje na plikach (zbiorach) dyskowych .....	383
Z7.5. Operacje na rekordach w pliku .....	385
Z7.6. Zarządzanie pamięcią operacyjną .....	386
Z7.7. Funkcje systemowe .....	387
Z7.8. Sterowanie programem .....	388
Z7.9. Funkcje związane z czasem i datą .....	389
Z7.10. Inne funkcje .....	390
<b>Załącznik 8. Opis wybranych funkcji API .....</b>	<b>391</b>
Z8.1. CheckDlgButton .....	391
Z8.2. CloseHandle .....	392
Z8.3. CopyFile .....	393
Z8.4. CreateFile .....	394
Z8.5. CreateWindowEx .....	396
Z8.6. DeleteFile .....	399
Z8.7. ExitProcess .....	399
Z8.8. GetFileSize .....	400
Z8.9. MessageBox .....	400
Z8.10. ShowWindow .....	403
<b>Załącznik 9. Tablica kodów ASCII oraz kody klawiszy .....</b>	<b>405</b>
Z9.1. Kody ASCII .....	405
Z9.2. Kody klawiszy .....	405
<b>Załącznik 10. FTP wydawnictwa .....</b>	<b>411</b>
<b>Skorowidz .....</b>	<b>413</b>

## Rozdział 3.

# Architektura procesorów rodziny x86-32 widziana oczami programisty

Wszystkie instrukcje procesora<sup>1</sup>, takie jak np. operacje arytmetyczne czy logiczne, z jakich składa się program, wykonywane są na zmiennych w rejestrach procesora lub w pamięci operacyjnej. W dalszej części tego rozdziału musimy zatem poznać:

- ◆ rejestry procesora dostępne programowo i podstawowe formaty danych związanych w tymi rejestrami,
- ◆ organizację pamięci operacyjnej,
- ◆ pojęcie adresu logicznego, liniowego i rzeczywistego,
- ◆ sposoby adresowania argumentów w pamięci operacyjnej,
- ◆ tryby pracy procesora.

Każdy kolejny procesor firmy Intel należący do tzw. linii procesorów x86-32 był tak rozbudowywany, aby oprogramowanie działające na poprzednim, starszym modelu procesora mogło być w dalszym ciągu używane. Aby to było możliwe, w najnowszym procesorze Intel Core i7 możemy „zobaczyć” procesor 8086 i jego rejestry z charakterystyczną dla niego segmentową organizacją pamięci i sposobami adresowania argumentów.

---

<sup>1</sup> W dalszej części książki 32-bitową architekturę procesorów firmy Intel nazywać będziemy konsekwentnie x86-32 dla podkreślenia, że wywodzi się ona od pierwszego 16-bitowego procesora 8086. Firma Intel od pewnego czasu zrezygnowała z tej nazwy na rzecz IA-32. Do tej dużej grupy procesorów zaliczamy wszystkie procesory rodziny Celeron, Pentium i Core. Z wyjątkiem podrozdziału 3.15 nie będziemy zajmować się w tej książce 64-bitowym rozszerzeniem architektury procesorów Intel, oznaczanym jako x86-32e lub też EM64T, a w przypadku procesorów firmy AMD — x86-64.

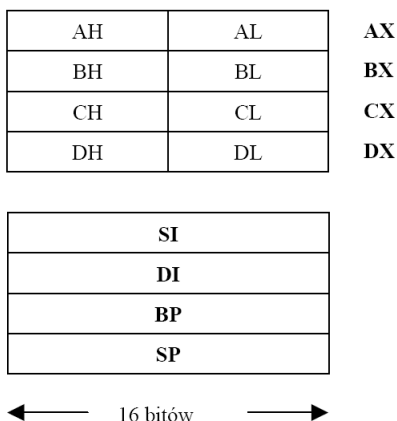
Warto wspomnieć w tym miejscu, iż w procesorach linii x86-32 odszukać można pewien ślad jeszcze wcześniejszych procesorów: 8-bitowych 8080 oraz 8085, a nawet ślad pierwszego mikroprocesora firmy Intel z 1969 roku, 4-bitowego procesora 4004.

## 3.1. Rejestry procesora 8086

Procesor 8086 to pierwszy procesor firmy Intel, w którym podstawowe rejestry dostępne programowo są 16-bitowe. Z niewielkim uproszczeniem możemy przyjąć, że współczesny procesor o architekturze x86-32 po włączeniu komputera do zasilania widziany jest przez programistę jako bardzo szybki procesor 8086. Będziemy mówili, że procesor pracuje wtedy w trybie adresacji rzeczywistej bądź w trybie 16-bitowym (choć programowo dostępne będą już rejestry 32-bitowe, wprowadzone w procesorze 80386). Programy uruchamiane pod kontrolą systemu operacyjnego MS DOS wykorzystywać będą ten właśnie tryb. Instrukcje procesora 8086 stanowią podzbiór zbioru instrukcji współczesnego procesora Intel Core i7. Mogą one operować na ośmiu podstawowych rejestrach (rysunek 3.1) oraz na argumentach w pamięci operacyjnej.

### Rysunek 3.1.

*Podstawowe rejestry procesora 8086*



Nazwy rejestrów przedstawiono na rysunku wielkimi literami, jednak w programie dla wygody najczęściej zapisywać będziemy je literami małymi. Język asemblera dopuszcza w tym zakresie pełną dowolność. Rejestry AX, BX, CX, DX, SI, DI, BP oraz SP w czasie wykonywania programu mogą zawierać odpowiednio:

- ◆ argumenty dla wykonywanych w programie operacji arytmetycznych oraz logicznych,
- ◆ argumenty służące do obliczania adresu w pamięci operacyjnej,
- ◆ wskaźniki do pamięci operacyjnej.

Niezależnie od tego, że z wyjątkiem rejestru **SP** wszystkie pozostałe będziemy mogli wykorzystywać do różnych z wymienionych powyżej celów, każdy z nich ma także swoją specyficzną funkcję, która związana jest z jego nazwą. Wyjaśnia to poniższe zestawienie:



- AX — główny rejestr jednostki stałoprzecinkowej procesora, służy jako akumulator dla argumentów instrukcji procesora oraz zapamiętania wyników,
- BX — rejestr bazowy, wskaźnik do danych w pamięci (w segmencie danych<sup>2</sup>),
- CX — licznik w operacjach na łańcuchach oraz w pętach programowych,
- DX — rejestr danych, rejestr adresowy układów wejścia-wyjścia,
- SI — rejestr indeksowy, wskaźnik do danych w segmencie danych; w operacjach na łańcuchach wskaźnik dla łańcucha źródłowego,
- DI — rejestr indeksowy, wskaźnik do danych; w operacjach na łańcuchach wskaźnik do łańcucha przeznaczenia,
- BP — rejestr bazowy, wskaźnik do danych w segmencie stosu,
- SP — wskaźnik szczytu stosu.

W rejestrach AX, BX, CX, DX możemy niezależnie adresować ich młodsze i starsze bajty, odpowiednio używając nazw: AH, AL, BH, BL, CH, CL, DH i DL.

Przypomnijmy sobie, że w naszym „pierwszym programie”, wyświetlającym na ekranie napis „Hello, world!”, wykorzystywaliśmy dwa spośród wymienionych wyżej rejestrów: DX oraz AH. Przez te rejestry przekazywaliśmy parametry do podprogramu systemowego.

Listą instrukcji procesora zajmować się będziemy w rozdziale 5., jednak już teraz pokażemy przykładowe instrukcje (rozkazy) wykorzystujące poznane rejestry:

```

mov ax, bx      ; skopiuj zawartość rejestru BX do rejestru AX
add ah, cl      ; dodaj binarnie zawartość rejestru CL do rejestru AH,
                ; wynik umieść w rejestrze AH
neg bx          ; negacja zawartości rejestru BX (operacja na bitach)
cmp di, si      ; porównaj zawartości rejestrów DI oraz SI

```

Wymienić musimy jeszcze dwa inne rejestry procesora 8086: wskaźnik instrukcji IP oraz rejestr znaczników FLAGS. Rejestry te pokazane są na rysunku 3.2.

**Rysunek 3.2.**  
Rejestry FLAGS  
oraz IP  
procesora 8086



Rejestr IP wskazywać będzie adres w segmencie kodu, z którego kolejno pobierane będą instrukcje programu do wykonania. Po pobraniu instrukcji jego zawartość powiększana będzie o taką liczbę bajtów, z ilu składa się dana instrukcja. W przypadku instrukcji

<sup>2</sup> O segmentowej organizacji pamięci będziemy mówić w dalszych częściach tego rozdziału.

skoku bądź wywołania podprogramu, czyli przeniesienia sterowania w inne miejsce programu, do rejestru IP zapisywany będzie adres, pod którym zaczynają się instrukcje tego fragmentu programu (bądź podprogramu), jakie mają być wykonywane.

Szczególne znaczenie w procesorze ma rejestr jednobitowych znaczników FLAGS. Wyróżniamy wśród nich 6 znaczników stanu oraz 3 znaczniki sterujące. Znaczniki stanu informują o pewnych cechach otrzymanego wyniku po wykonaniu operacji arytmetycznej bądź logicznej. Ilustruje to tabela 3.1.

**Tabela 3.1.** Znaczniki stanu w rejestrze FLAGS

Symbol znacznika	Nazwa znacznika	Jak się zachowuje po operacji arytmetycznej bądź logicznej	Przykład
CF	znacznik przeniesienia (ang. <i>carry flag</i> )	Przyjmuje wartość 1, gdy w wyniku wykonanej operacji nastąpiło przeniesienie (np. przy dodawaniu) z bitu najstarszego na zewnątrz lub też (np. przy odejmowaniu) nastąpiła pożyczka z zewnątrz do bitu najstarszego. W przeciwnym razie znacznik jest zerowany.	1010 1110 <u>+ 0111 0100</u> 1 0010 0010 CF=1
PF	znacznik parzystości (ang. <i>parity flag</i> )	Ustawiany jest na wartość 1 wtedy, gdy w wyniku zrealizowanej operacji liczba bitów o wartości 1 w młodszym bajcie wyniku jest parzysta. Gdy jest nieparzysta, znacznik jest zerowany.	0010 1100 <u>+ 1011 0001</u> 1101 1101 PF=1
AF	znacznik przeniesienia pomocniczego (ang. <i>auxiliary flag</i> )	Przyjmuje wartość 1, gdy nastąpiło przeniesienie z bitu 3 na 4 lub pożyczka z bitu 4 na 3. W przeciwnym razie wskaźnik jest zerowany. Wskaźnik AF wykorzystywany jest przy operacjach na liczbach BCD.	0010 1110 <u>+ 0110 0100</u> 1001 0010 AF=1
ZF	znacznik zera (ang. <i>zero flag</i> )	Przyjmuje wartość 1 wtedy, gdy wynik operacji jest równy zero, i jest zerowany w przeciwnym razie.	1111 1111 <u>+ 0000 0001</u> 0000 0000 ZF=1
SF	znacznik znaku (ang. <i>sign flag</i> )	Przyjmuje wartość 1, gdy najbardziej znaczący bit (bit znaku) w otrzymanym wyniku jest równy 1, i jest zerowany w przeciwnym razie. Stan znacznika SF jest zatem zgodny z bitem znaku.	0110 0000 <u>+ 0100 0001</u> 1010 0001 SF=1
OF	znacznik przepełnienia (ang. <i>overflow flag</i> )	Przyjmuje wartość 1, gdy przy realizowaniu określonej operacji wystąpiło przeniesienie na bit znaku lub też z tego bitu pobrana została pożyczka, ale nie wystąpiło przeniesienie (lub pożyczka) z bitu znaku (tzn. CF=0) W przeciwnym razie znacznik jest zerowany. Stan znacznika OF jest istotny w czasie operacji na liczbach ze znakiem.	0110 1010 <u>+ 0101 1001</u> 1100 0011 OF=1

W zależności od stanu pojedynczych znaczników lub ich logicznej kombinacji można dzięki zastosowaniu właściwych instrukcji skoków zmienić przebieg realizowanego programu. Analizując ustawienia znaczników stanu, warto pamiętać, że procesor nie

rozdziela, czy liczba traktowana jest przez nas w programie jako liczba ze znakiem, czy też bez znaku. Klasycznym przykładem ilustrującym, w jaki sposób procesor wykorzystuje znacznik zera ZF, może być następująca sekwencja instrukcji:

```
cmp al, bh      ; porównaj zawartość rejestrów – ustaw znaczniki w rej. FLAGS
jz sa_rownne   ; skocz do etykiety sa_rownne, jeśli znacznik zera został
                ; ustawiony
...            ; kontynuuj w przeciwnym razie
```

Wykonanie instrukcji `cmp` sprowadza się do wykonania operacji odejmowania. Wynik odejmowania nie zostaje zapamiętany, a jedynie ustawione zostają znaczniki, w tym znacznik zera ZF. Rozkaz skoku warunkowego `jz` (ang. *jump if zero flag*) bada stan tego znacznika i jeśli stwierdzi, że jest równy 1 (co oznacza, że w wyniku odejmowania wynik był równy zero, czyli porównywane rejestry zawierały te same wartości), przeniesie sterowanie do etykiety (adresu symbolicznego) o nazwie `sa_rownne`. W przeciwnym razie skok nie jest wykonywany i procesor przechodzi do realizacji kolejnej instrukcji.

Inny rodzaj znaczników w rejestrze FLAGS to znaczniki sterujące. Mogą być ustawiane bądź zerowane programowo w celu wymuszenia odpowiedniego sposobu pracy procesora. Ilustruje je tabela 3.2.

**Tabela 3.2.** Znaczniki sterujące w rejestrze FLAGS

Symbol znacznika	Nazwa znacznika	Znaczenie znacznika
TF	znacznik pracy krokowej (ang. <i>trap flag</i> )	Stan równy 1 powoduje wprowadzenie procesora w tryb pracy (ang. <i>single step mode</i> ) umożliwiający wygenerowanie przerwania (ang. <i>single step interrupt</i> ) i przejście do specjalnych procedur obsługi (np. programów uruchomieniowych) po każdym wykonanym rozkazie. Wyzerowanie znacznika TF powoduje powrót procesora do normalnej pracy.
IF	znacznik zezwolenia na przerwanie (ang. <i>interrupt flag</i> )	Ustawiony w stan 1 powoduje odblokowanie systemu przerwania procesora. Zewnętrzne przerwania maskowane mogą przerwać realizację wykonywanego aktualnie programu. Wyzerowanie znacznika powoduje, że przerwania te są przez procesor ignorowane.
DF	znacznik kierunku (ang. <i>direction flag</i> )	Wykorzystywany jest przy wykonywaniu operacji na łańcuchach (tablicach). Jeżeli ma wartość 0, przetwarzanie łańcuchów odbywa się z inkrementacją adresów, jeżeli zaś jest ustawiony — adresy maleją.



Procesory x86-32, a zatem także współczesne procesory serii Intel Core, po włączeniu do zasilania pracują w trybie adresacji rzeczywistej. Na tym etapie rozważań możemy, upraszczając, przyjąć, że dla programisty jest to wtedy bardzo szybki procesor 8086. W tym trybie ma on do dyspozycji 8 rejestrów 16-bitowych, z których każdy może zawierać argument wykonywanych instrukcji lub też może służyć do obliczenia adresu argumentu w pamięci operacyjnej.

## 3.2. Zwiększamy rozmiar rejestrów — od procesora 80386 do Intel Core i7<sup>3</sup>

W procesorze 80386 w miejsce poznanych poprzednio rejestrów 16-bitowych wprowadzono rejestry 32-bitowe. I tak już pozostało — aż do współczesnych procesorów o architekturze x86-32 (pracujących w trybie 32-bitowym). Procesory te posiadają szereg innych, dodatkowych rejestrów, które poznamy w dalszej części rozdziału. Układ podstawowych ośmiu rejestrów i ich przeznaczenie praktycznie się nie zmieniły, co pokazuje rysunek 3.3.

### Rysunek 3.3.

*Rejestry podstawowe procesora 80386 i nowszych*

31	16 15	8 7	0	16-bit	32-bit
	AH	AL		AX	EAX
	BH	BL		BX	EBX
	CH	CL		CX	ECX
	DH	DL		DX	EDX
	BP				EBP
	SI				ESI
	DI				EDI
	SP				ESP

Jak widać na tym rysunku, wszystkie rejestry zostały powiększone do 32 bitów, zaś w nazwie pojawiła się na pierwszym miejscu litera „E”, od angielskiego słowa *expand* — poszerzenie, powiększenie. W programie wykorzystywać możemy zarówno 32-bitowe rejestry: EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, jak i mapowane na nich, znane nam już rejestry 16-bitowe: AX, BX, CX, DX, SI, DI, BP, SP oraz, odpowiednio, ośmiobitowe: AH, AL, BH, BL, CH, CL, DH, CL. Rola rejestrów 32-bitowych jest generalnie zgodna z opisaną wcześniej rolą ich odpowiedników 16-bitowych. Dzięki temu zachowana została zgodność programowa „w dół”, tzn. programy przygotowane dla starszych procesorów mogą być wykonywane na procesorach nowszych.

Począwszy od procesora 80386, konsekwentnie mamy też do dyspozycji 32-bitowy rejestr wskaźnika rozkazów EIP, a także 32-bitowy rejestr znaczników EFLAGS. W rejestrze EFLAGS oprócz znanych nam z procesora 8086 znaczników stanu oraz znaczników sterujących pojawiła się nowa grupa znaczników: znaczniki systemowe. Są one związane z pracą procesora w trybie adresacji wirtualnej z ochroną i nie mogą być używane w programie użytkowym. Z tego też względu nie będziemy się nimi zajmować w tej książce.

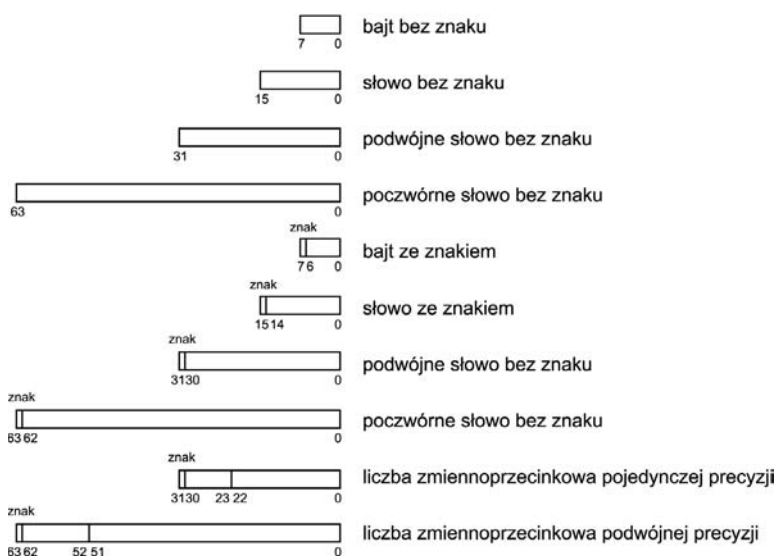
Podstawowe typy danych, jakimi operować będziemy, wykorzystując pokazane na rysunku 3.3 rejestry procesora, pokazuje rysunek 3.4.

Pokażmy teraz przykładowe instrukcje procesora, wykorzystujące pokazane na rysunku 3.3 rejestry procesora 80386:

<sup>3</sup> W architekturze x86-32.

Rysunek 3.4.

Podstawowe  
typy danych



and eax, esi ; iloczyn logiczny bitów rejestrów EAX i ESI  
 cwde ; (argument niejawny) konwersja liczby ze znakiem  
 ; w rejestrze AX do rejestru EAX (czyli z liczby  
 ; 16-bitowej robimy 32-bitową)  
 cmp bp, bx ; porównanie zawartości rejestrów BP i BX  
 ; (ustawia znaczniki w EFLAGS)  
 mov edi, esi ; skopiowanie zawartości rejestru indeksowego ESI do EDI  
 sub ecx, 8 ; odjęcie od zawartości rejestru ECX liczby 8

## 3.3. Zwiększamy liczbę rejestrów — od procesora i486<sup>4</sup> do Intel Core i7<sup>5</sup>

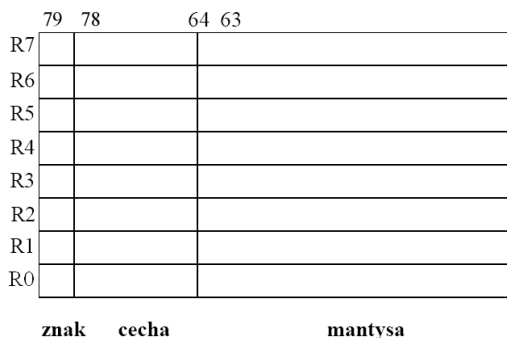
Wykonywanie złożonych obliczeń na liczbach zmiennoprzecinkowych przy wykorzystaniu dotychczas poznanych przez nas rejestrów jest uciążliwe i mało wydajne. Dlatego też równoległe do procesora 8086 powstał koprocesor arytmetyczny 8087 (jako oddzielny układ scalony), a później odpowiednio dla kolejnych procesorów 80286 i 80386 koprocesory: 80287 i 80387. Koprocesor arytmetyczny posiada 80-bitowe rejestry oraz listę instrukcji pozwalającą na stosunkowo proste wykonywanie nawet bardzo złożonych operacji matematycznych na liczbach zmiennoprzecinkowych. Dla programisty procesor główny oraz koprocesor arytmetyczny tworzyły od początku jak gdyby jeden procesor o powiększonych możliwościach. W programie w identyczny sposób

<sup>4</sup> Według wcześniejszego nazewnictwa firmy Intel — 80486.

<sup>5</sup> W architekturze x86-32.

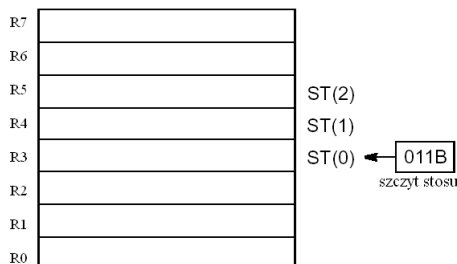
umieszczać można instrukcje wykonywane przez każdy z tych procesorów. Począwszy od procesora i486, koprocessor arytmetyczny włączony został do procesora głównego, przy zachowaniu wszystkich swoich funkcji. W procesorach o architekturze x86-32 występuje w postaci tzw. jednostki zmiennoprzecinkowej<sup>6</sup> — w odróżnieniu od jednostki stałoprzecinkowej, bazującej na omówionych wcześniej rejestrach 32-bitowych. Rejestry jednostki zmiennoprzecinkowej (koprocessora arytmetycznego) dostępne programowo przedstawia rysunek 3.5.

**Rysunek 3.5.**  
80-bitowe rejestry  
koprocessora  
arytmetycznego  
(z zaznaczeniem pól  
zajmowanych  
przez liczbę  
zmiennoprzecinkową)



Instrukcje operujące na tych rejestrach traktują je jako stos ośmiu rejestrów, przy czym rejestr wskazywany przez 3-bitowy wskaźnik jako szczyt stosu nazywać będziemy ST(0) lub po prostu ST. Kolejne rejestry to odpowiednio: ST(1), ST(2) itd. aż do ST(7). Pokazuje to rysunek 3.6.

**Rysunek 3.6.**  
Stosowa organizacja  
rejestrów koprocessora  
arytmetycznego



Zapis do rejestru ST(0), którym w danym momencie będzie np. rejestr R3, wiązać się będzie z wcześniejszym odjęciem od trzybitowego wskaźnika stosu jedynki i tym samym operacja zapisu wykonana zostanie do R2. Rejestr R3, będący poprzednio szczytem stosu, stanie się rejestrem ST(1). Do rejestrów ST(i) dane zapisywane będą w formacie: znak, wykładnik, mantysa — tak jak pokazują to rysunki 3.4 i 3.5. Poniżej pokażemy kilka przykładowych instrukcji procesora operujących na tych rejestrach:

```
fldpi          ; załadowanie na szczyt stosu do rejestru ST(0)
                ; liczby stałej „pi”
fld zmienna    ; załadowanie na szczyt stosu do rejestru ST(0)
                ; zmiennej z pamięci
```

<sup>6</sup> Informatycy — szczególnie ci pamiętający czasy procesorów 8086, 80286 i 80386, jak i koprocessorów arytmetycznych 8087, 80287 i 80387 — w dalszym ciągu operują pojęciem *koprocessor arytmetyczny*, mimo iż jest on w tej chwili integralną częścią procesora o architekturze x86-32.

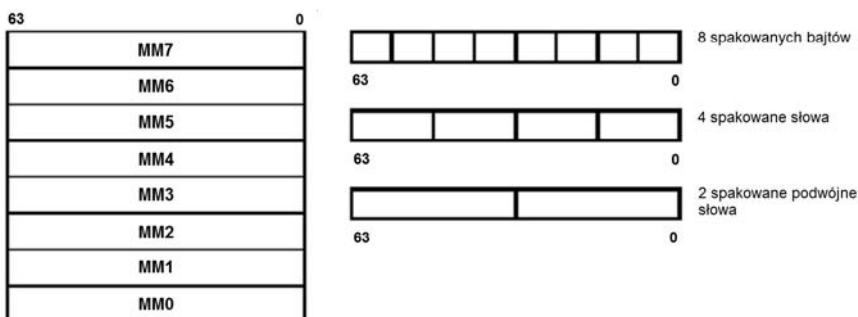
```

fsin          ; obliczenie funkcji sinus z liczby
              ; umieszczonej w ST(0)
fsub ST(0), ST(3) ; operacja na liczbach zmiennoprzecinkowych:
              ; ST(0) ← ST(0)-ST(3)
fchx ST(5)     ; zamiana zawartości rejestrów ST(0) i ST(5)

```

Wykonywanie operacji na liczbach zmiennoprzecinkowych nie jest — niestety — sprawą prostą. Musimy pamiętać o ułomności każdej cyfrowej maszyny matematycznej w zakresie przedstawiania osi liczbowej, możliwej do osiągnięcia dokładności obliczeń czy wreszcie konieczności operowania takimi podstawowymi w matematyce pojęciami, jak np. nieskończoność.

Kolejne rejestry udostępnione zostały programistom w procesorze Pentium MMX. Był to przejściowy model procesora produkowany przez krótki czas, dlatego mówi się raczej, że rozszerzenie procesora o nazwie MMX pojawiło się w Pentium II. Konieczność zwiększania mocy obliczeniowej wymaganej w coraz bardziej popularnych zastosowaniach multimedialnych spowodowała sięgnięcie do tzw. technologii **SIMD** (ang. *Single Instruction Multiple Data*). W technologii tej pojedyncza instrukcja procesora wykonywana jest równolegle na kilku danych. Rejestry oraz wprowadzone wraz z tzw. rozszerzeniem MMX typy danych pozwalające wykonywać tego typu operacje w procesorze Pentium II (i oczywiście także nowszych) przedstawia rysunek 3.7.



**Rysunek 3.7.** Rejestry MMX oraz typy danych

Każdy z ośmiu 64-bitowych rejestrów może zawierać:

- ♦ jedną daną 64-bitową,
- ♦ dwie dane 32-bitowe (ang. *Packed Doubleword Integers*),
- ♦ cztery dane 16-bitowe (ang. *Packed Word Integers*),
- ♦ 8 danych bajtowych (ang. *Packed Byte Integers*),

na których równocześnie wykonywać możemy określone operacje, w szczególności arytmetyczne i logiczne na liczbach całkowitych. Przykłady wykorzystania rejestrów MMX w programie asemblerowym pokazane będą w dalszych rozdziałach, teraz przedstawimy jedynie kilka przykładowych instrukcji procesora wykorzystujących te rejestry.

```

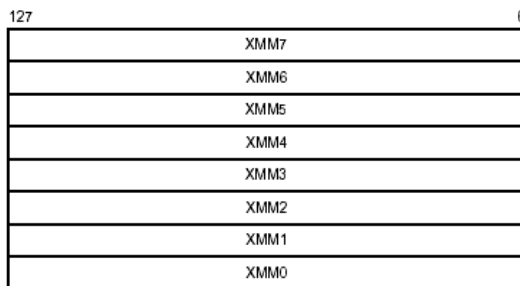
pxor mm1, mm2 ; wykonaj operację logiczną XOR na bitach
              ; rejestrów MM1 i MM2
psslq mm1, 3  ; przesun logicznie zawartość (całego)
              ; rejestru MM1 w lewo o 3 bity

```

`paddb mm2, mm3 ; dodaj odpowiadające sobie bajty rejestrów MM2 i MM3`  
`paddd mm2, mm3 ; dodaj odpowiadające sobie słowa rejestrów MM2 i MM3`

W procesorze Pentium III wprowadzono kolejnych osiem rejestrów, tym razem 128-bitowych, które stanowią podstawowe zasoby rozszerzenia procesora o architekturze x86-32 o nazwie SSE (ang. *the Streaming SIMD Extensions*). W procesorze Pentium 4 nie przybyło już żadnych dostępnych programowo rejestrów, jedynie rozbudowana została lista instrukcji. Rozszerzenie procesora Pentium 4 nazwano w związku z tym SSE2. Rejestry rozszerzeń SSE i SSE2 nazywamy XMM (rysunek 3.8). Podobnie jak opisane wcześniej rejestry MMX, pracują one w technologii SIMD. O ile jednak w poprzednich rejestrach mogliśmy przetwarzać równoległe kilka danych będących liczbami całkowitymi, o tyle teraz operacje będą możliwe także na liczbach zmiennoprzecinkowych. W pewnym uproszczeniu można powiedzieć, że rozszerzenie SSE/SSE2 łączy w sobie możliwości koprocesora arytmetycznego (przetwarzanie liczb zmiennoprzecinkowych) oraz MMX (przetwarzanie typu SIMD). Rozszerzenia SSE/SSE2 wprowadzone zostały w celu zwiększenia szybkości przetwarzania w takich zastosowaniach, jak grafika 2D i 3D, przetwarzanie obrazów, animacja, rozpoznawanie mowy, obsługa wideokonferencji itp. W lutym 2004 roku firma Intel wprowadziła na rynek wersję procesora Pentium 4 z jądrem o nazwie Prescott. W procesorze tym wprowadzono rozszerzenie o nazwie SSE3. Lista instrukcji kolejny raz została powiększona, jednak nie zmieniła się liczba dostępnych programowo rejestrów. W następnych wersjach procesorów o architekturze x86-32 wprowadzono kolejne rozszerzenia listy rozkazów (SSSE3, SSE4), jednak liczba oraz wielkość rejestrów dostępnych programowo w trybie pracy 32-bitowej nie uległy<sup>7</sup> zmianie.

**Rysunek 3.8.**  
*Rejestry XMM*  
*rozszerzenia*  
*SSE/SSE2/SSE3/SSE4*



W procesorach o architekturze x86-32 w każdym ze 128-bitowych rejestrów możemy równoległe przetwarzać (rysunek 3.9):

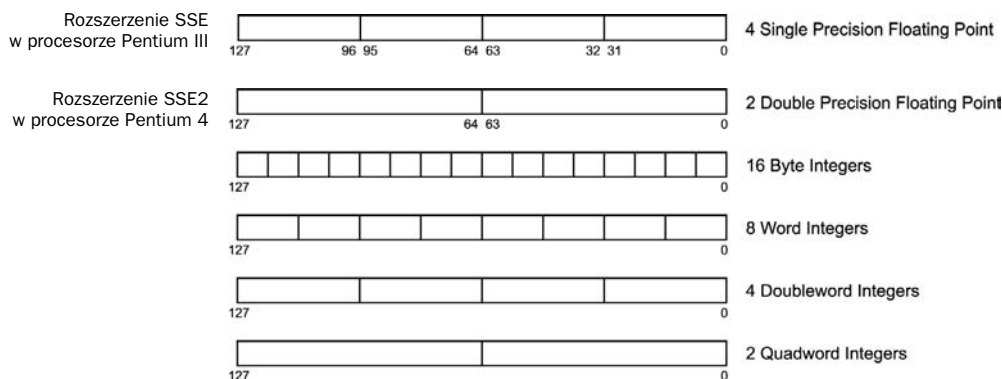
- ◆ cztery 32-bitowe liczby zmiennoprzecinkowe pojedynczej precyzji<sup>8</sup>,
- ◆ dwie 64-bitowe liczby zmiennoprzecinkowe podwójnej precyzji,
- ◆ szesnaście bajtów<sup>9</sup> traktowanych jako liczby stałoprzecinkowe,

<sup>7</sup> W chwili, gdy pisane są te słowa, firma Intel wprowadziła procesory Core i7, i5 oraz i3. W trybie 32-bitowym liczba rejestrów dostępnych programowo nie została zmieniona w stosunku do procesora Pentium III.

<sup>8</sup> W procesorze Pentium III w 128-bitowych rejestrach XMM instrukcje obsługują jedynie ten format danych. Pozostałe formaty dostępne są w procesorze Pentium 4 oraz nowszych.

<sup>9</sup> Ten i kolejne wymienione tutaj formaty stałoprzecinkowe są rozszerzeniem technologii MMX na rejestry XMM.

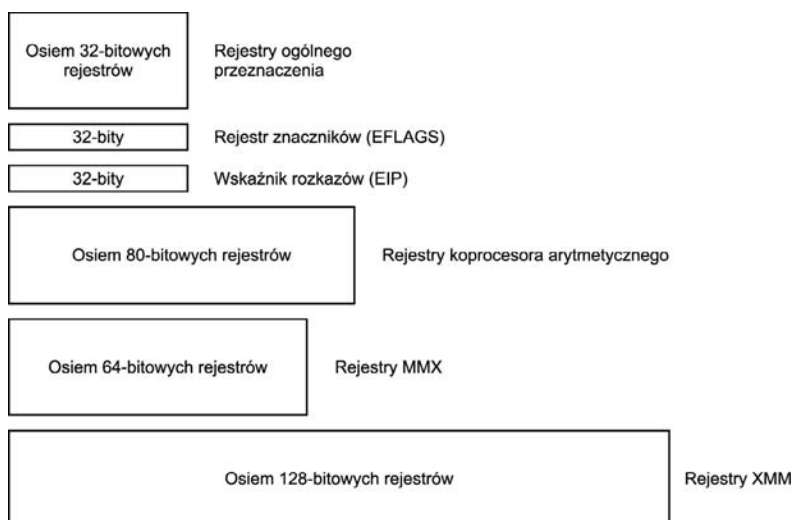




**Rysunek 3.9.** Typy danych w rejestrach XMM

- ♦ osiem słów (liczby stałoprzecinkowe),
- ♦ cztery podwójne słowa (liczby stałoprzecinkowe),
- ♦ dwa poczwórne słowa (liczby stałoprzecinkowe).
- ♦ Wiemy już, jakimi rejestrami możemy posługiwać się w procesorze o architekturze x86-32, pisząc program w języku asemblera. Spójrzmy jeszcze na rysunek 3.10, który w uproszczony sposób pokazuje wszystkie rejestry, jakie będziemy mieli do dyspozycji przy pisaniu programów użytkowych. Należy również zaznaczyć, że 64-bitowe rejestry MMX w rzeczywistości są mapowane na rejestrach koprocesora arytmetycznego ST(i), co w konsekwencji wymaga rozdzielenia w programie operacji wykonywanych na tych dwóch zbiorach rejestrów.

**Rysunek 3.10.**  
Rejestry procesora x86-32





Wskazówka

Nie są to wszystkie dostępne programowo rejestry procesora. Pewna liczba rejestrów związana jest z segmentową organizacją pamięci, inne z kolei pełnią istotną rolę przy organizowaniu pracy procesora w trybie adresacji wirtualnej z ochroną. Niektóre z tych rejestrów poznamy, omawiając sposoby adresowania argumentów w pamięci operacyjnej.

Rejestry MMX są mapowane na rejestrach koprocesora ST(i), w związku z tym w programie nie możemy równocześnie wykorzystywać obu tych zbiorów rejestrów. Począwszy od procesora Pentium 4, jest to już mniejszy problem, ponieważ wszystkie operacje typu MMX możemy wykonywać także na rejestrach XMM. Jest to korzystne także ze względu na ich dwa razy większy rozmiar w stosunku do rejestrów MMX.

## 3.4. Segmentowa organizacja pamięci

Dane przetwarzane w programie mogą znajdować się bądź w omówionych już rejestrach procesora, bądź też w pamięci operacyjnej. Poznamy teraz sposób adresowania danych w pamięci. Pamięć operacyjna ma organizację bajtową, co oznacza, że każdy bajt w pamięci ma swój własny fizyczny adres. Maksymalna wielkość pamięci operacyjnej, czyli liczba bajtów, jakie procesor może zaadresować, zależy od wielkości magistrali adresowej. Dla kolejnych procesorów x86 przedstawia to tabela 3.3.

**Tabela 3.3.** Wielkość pamięci operacyjnej w różnych procesorach firmy Intel

Procesor	Wielkość magistrali adresowej	Maksymalna wielkość pamięci operacyjnej	Wielkość offsetu	Maksymalna wielkość segmentu w pamięci
8086	20 bitów	1 MB	16 bitów	64 kB
80286	24 bity	16 MB	16 bitów	64 kB
Począwszy od procesora 80386 do Pentium włącznie	32 bity	4 GB	32 bity	4 GB
Począwszy od procesora Pentium PRO do Pentium 4	36 bitów	64 GB	32 bity	4 GB
Począwszy od Pentium 4 EE (Prescott)	40 bitów	1 TB	32 bity	4 GB

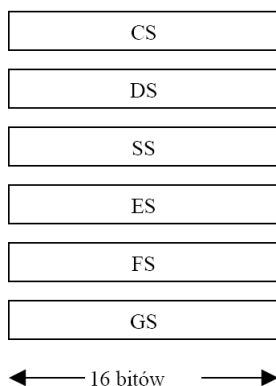
W programie nie będziemy się jednak posługiwać adresem fizycznym, tylko adresem logicznym. Fizyczna pamięć operacyjna umownie podzielona zostanie na pewne spójne fragmenty, zwane segmentami. Każdy bajt w pamięci będziemy adresować poprzez adres logiczny składający się z dwóch części:

- ◆ z adresu początku segmentu,
- ◆ z adresu względem początku segmentu, który nazywać będziemy *offsetem*.

Wielkość *offsetu* związana jest wprost z rozmiarem dostępnych w procesorze rejestrów ogólnego przeznaczenia, które — jak za chwilę pokażemy — będą mogły uczestniczyć w jego obliczaniu. I tak w procesorze 8086, a więc także w procesorach x86-32 pracujących w trybie adresacji rzeczywistej, offset ma 16 bitów, z czego wynika maksymalna wielkość segmentu wynosząca 64 kilobajty. Począwszy od procesora 80386 pracującego w trybie adresacji wirtualnej, offset ma 32 bity, zaś maksymalna wielkość segmentu to 4 gigabajty.

Do określenia adresu początku segmentu służą rejestry procesora, zwane rejestrami segmentowymi. Procesor 8086 ma je cztery, zaś w procesorze 80386 i dalszych mamy do dyspozycji 6 takich rejestrów. Wielkość rejestrów segmentowych wynosi 16 bitów (rysunek 3.11), co powoduje, że ich zawartość nie może być wprost fizycznym adresem początku segmentu. Wymagałoby to bowiem odpowiednio większych rejestrów: dla procesora 8086 rejestru 20-bitowego, zaś dla współczesnych procesorów x86-32 — rejestru 40-bitowego lub, dla niektórych, 52-bitowego.

**Rysunek 3.11.**  
*Rejestry segmentowe  
procesora Pentium 4*



Rejestry CS, DS, SS, ES występują  
we wszystkich procesorach 80x86

Rejestry FS, GS występują jedynie  
w procesorach 80386 i nowszych

Adres logiczny zapisywać będziemy, podając obie jego części oddzielone dwukropkiem, np.:

```
1000:0000
3af8:076b
DS:2ab7
ES:DI
CS:IP
```

Jak widać, w wyrażeniu określającym adres logiczny mogą wystąpić konkretne liczby (na ogół w kodzie szesnastkowym) bądź nazwy rejestrów, które zawierają odpowiednią wartość segmentu lub offsetu. Warto zwrócić uwagę na adres logiczny CS:IP, który określa, skąd pobierane będą do wykonania kolejne instrukcje programu.

Jak już wiemy, w procesorze 8086 (lub inaczej: dla procesora x86-32 w trybie adresowania rzeczywistego) dla określenia fizycznego adresu początku segmentu potrzeba 20 bitów (porównaj tabelę 3.3). Rejestr segmentowy zawierać będzie w takim przypadku 16 starszych bitów tego 20-bitowego adresu. Brakujące cztery najmłodsze bity będą miały zawsze wartość zero. Tak więc adres początku segmentu w procesorze

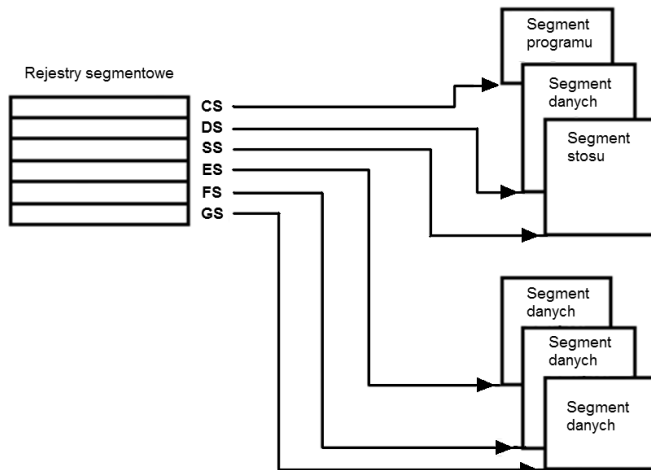
8086 zawsze musi być podzielny przez 16. Warto wiedzieć, według jakiego algorytmu procesor 8086 przelicza adres logiczny używany w programie na adres fizyczny. Ilustruje to prosty przykład przeliczony dla adresu logicznego 2a87:1005.

segment	2a87	0010 1010 1000 0111
offset	+ 100a	0001 0000 0000 1010
adres fizyczny	2b87a	0010 1011 1000 0111 1010

Segmentową organizację pamięci operacyjnej ilustruje schematycznie rysunek 3.12. W programie możemy zdefiniować wiele różnych segmentów, lecz w danym momencie dostęp będziemy mieli jedynie do sześciu, wskazywanych przez zawartość poszczególnych rejestrów segmentowych. Segmenty w programie mogą się nakładać na siebie w pamięci, mogą być ułożone jeden po drugim lub też między poszczególnymi segmentami mogą występować przerwy. Dla najprostszego małego programu musimy zdefiniować po jednym segmencie z programem, danymi oraz stosem. Tak właśnie zrobiliśmy w naszym pierwszym programie „Hello, world!” w rozdziale 2., wybierając model pamięci **SMALL**.

### Rysunek 3.12.

*Segmentowa organizacja pamięci operacyjnej*



Każdy z rejestrów segmentowych związany jest z segmentem o ściśle określonej roli:

CS — wskazuje segment z programem (w skrócie segment kodu programu albo — jeszcze krócej — segment kodu),

DS — wskazuje segment z danymi; większość operacji na danych będzie standardowo związana z tym segmentem,

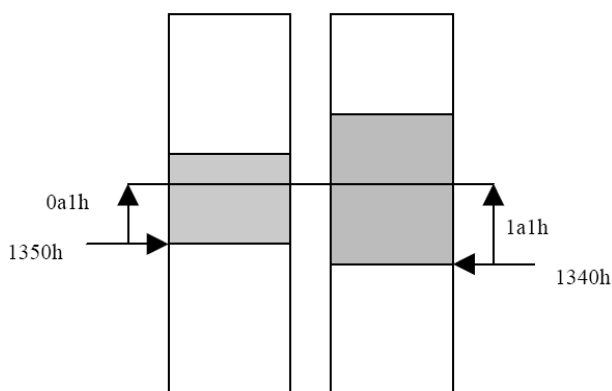
SS — wskazuje segment stosu,

ES, FS, GS — wskazują dodatkowe segmenty danych.

Warto zauważyć, że w trybie adresacji rzeczywistej ten sam bajt w pamięci możemy zaadresować różnymi adresami logicznymi. Występuje tutaj redundancja wynikająca z tego, że do zaadresowania przestrzeni jednomegabajtowej używamy aż 32 bitów adresu logicznego o postaci *segment:offset*. Przykład ilustrujący to zagadnienie pokazany jest na rysunku 3.13.

**Rysunek 3.13.**

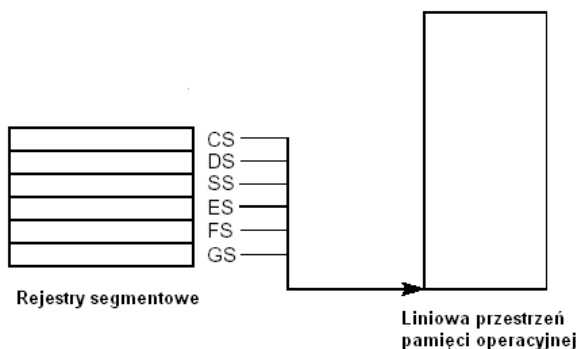
Różne adresy logiczne  
wskazują ten sam  
adres fizyczny 135a1h  
w pamięci operacyjnej



W przypadku procesorów x86-32 pracujących w trybie adresacji wirtualnej z ochroną przeliczanie zawartości 16-bitowego rejestru segmentowego, zwanego w tym trybie selektorem, na odpowiadający mu adres początku segmentu jest znacznie bardziej złożone. Ponieważ wymagałoby to bardzo obszernego i drobiazgowego opisu, a przeliczenie adresu jest procesem niewidocznym dla programu użytkowego, zagadnienie to nie będzie tutaj przedstawione<sup>10</sup>. Programowanie w tym trybie pod kontrolą systemu Windows pozwala zapomnieć o segmentowej organizacji pamięci i operować jedynie 32-bitowym offsetem, który traktowany jest jako adres liniowy w przestrzeni 4-gigabajtowej. Programy użytkowe uruchamiane w środowisku Windows mają do dyspozycji tzw. płaski model pamięci, który można traktować jako szczególny przypadek modelu segmentowego. Pokazuje to rysunek 3.14.

**Rysunek 3.14.**

Płaski model pamięci



Wszystkie rejestry segmentowe wskazują w tym modelu na ten sam 4-gigabajtowy segment rozpoczynający się od adresu 0.

<sup>10</sup> Znajomość tych zagadnień jest niezbędna w przypadku tworzenia własnego systemu operacyjnego.

## 3.5. Adresowanie argumentów

Instrukcje procesora mogą być bezargumentowe, mogą mieć jeden, dwa lub trzy argumenty. Niektóre argumenty występują w instrukcji jawnie, inne mogą być zakodowane wewnątrz instrukcji. Argumentami instrukcji mogą być:

- ◆ argumenty bezpośrednie będące częścią instrukcji,
- ◆ rejestry,
- ◆ argumenty w pamięci operacyjnej,
- ◆ układy wejścia-wyjścia komputera.

Omówieniem instrukcji procesorów zajmiemy się dopiero w rozdziale 5., jednak teraz na potrzeby objaśnienia sposobu adresowania argumentów posłużymy się znaną już z pierwszego rozdziału instrukcją MOV (kopiuj) oraz instrukcją ADD (dodaj). Dwa pierwsze (spośród wyżej wymienionych) sposoby adresowania argumentów są, jak się wydaje, oczywiste i możemy zilustrować je następującymi prostymi przykładami:

*; dwa argumenty będące rejestrami:*

```
mov ax, si      ; skopiuj zawartość rejestru indeksowego SI do AX
mov edi, ebx   ; skopiuj zawartość rejestru bazowego EBX do indeksowego EDI
add esi, ebp   ; dodaj do rejestru indeksowego ESI zawartość rejestru EBP
add cl, ch     ; dodaj zawartość rejestru CH do zawartości rejestru CL
```

*; jeden argument będący rejestrem oraz argument bezpośredni:*

```
mov ah, 2      ; wpisz do rejestru AH liczbę 2 (binarnie)
mov bp, 0      ; wyzeruj rejestr bazowy BP
add cx, 100h   ; dodaj do zawartości rejestru CX wartość 100 szesnastkowo
add ebx, 0ffh  ; dodaj do rejestru EBX wartość 0ff szesnastkowo
```

Adresowaniu argumentów w pamięci operacyjnej poświęcimy cały następny podrozdział, jest to bowiem zagadnienie złożone i ważne. Teraz natomiast zajmiemy się adresowaniem układów wejścia-wyjścia.

Procesory x86-32 obsługują przestrzeń adresową zawierającą maksymalnie 65 536 (64 k) ośmiobitowych układów wejścia-wyjścia (portów we-wy). W przestrzeni tej mogą być definiowane także porty 16- i 32-bitowe. Porty we-wy można adresować bezpośrednio lub za pośrednictwem rejestru DX. Dla ilustracji posłużymy się tutaj instrukcjami procesora: IN (wprowadź z układu wejściowego) oraz OUT (wyprowadź do układu wyjściowego).

```
in al, 3f8h    ; wprowadź do rejestru AL bajt z układu wejściowego
               ; o adresie 3fah
mov dx, 0dff0h ; zapisz w rejestrze DX adres portu wejściowego
in ax, dx     ; wprowadź do rejestru AX słowo z układu wejściowego,
               ; którego adres jest zapisany w rejestrze DX
out 61h, bl   ; wyprowadź do układu wyjściowego o adresie 61 szesnastkowo
               ; zawartość rejestru BL
mov dx, 378h  ; zapisz do rejestru DX adres portu wyjściowego
out dx, al    ; wyprowadź do układu wyjściowego o adresie znajdującym się
               ; w rejestrze DX zawartość rejestru AL
```

## 3.6. Adresowanie argumentów w pamięci operacyjnej

Procesory x86-32 pozwalają na zaadresowanie argumentów w pamięci operacyjnej na wiele różnych sposobów. Poznanie tych sposobów oraz ich właściwe wykorzystanie w różnych konstrukcjach programu jest jedną z ważniejszych umiejętności programisty. Każdy argument w pamięci operacyjnej określony jest przez adres, pod jakim się znajduje. Standardowo operacje na danych związane są z segmentem danych wskazywanym przez zawartość rejestru segmentowego DS i dlatego najczęściej w programie operujemy jedynie offsetem, czyli adresem względem początku segmentu. W obliczaniu końcowego offsetu, który nazywać będziemy także **adresem efektywnym**, uczestniczyć mogą rejestry procesora należące do jednostki stałoprzecinkowej. I tak:

- ♦ w procesorze 8086 oraz w nowszych procesorach o architekturze x86-32 pracujących w 16-bitowym trybie adresacji rzeczywistej w obliczaniu adresu efektywnego mogą uczestniczyć rejestry indeksowe DI, SI oraz bazowe BX, BP,
- ♦ w procesorach o architekturze x86-32 w trybie adresacji rzeczywistej do obliczania adresu efektywnego będzie można wykorzystywać rejestry 32-bitowe (podobnie jak w trybie adresacji wirtualnej), jednak zasadniczo jest to tryb bazujący na rejestrach 16-bitowych — przez analogię do procesora 8086<sup>11</sup>,
- ♦ w procesorach x86-32 pracujących w 32-bitowym trybie adresacji wirtualnej w obliczaniu adresu efektywnego mogą uczestniczyć wszystkie rejestry 32-bitowe: EAX, EBX, ECX, EDX, ESI, EDI, EBP i ESP.

Dla uproszczenia mówić będziemy w skrócie o adresowaniu 16-bitowym oraz 32-bitowym — mając na myśli wielkość offsetu.

Dostępne tryby adresowania argumentów w pamięci operacyjnej w trybie 16-bitowym zebrane są w tabeli 3.4. Rejestr ujęty w nawiasy kwadratowe oznacza, że jego zawartość to offset uczestniczący w obliczaniu adresu efektywnego argumentu w pamięci operacyjnej. Liczba ujęta w nawiasy kwadratowe jest wartością przemieszczenia względem początku segmentu, czyli wprost offsetem. Wielkość argumentu w pamięci operacyjnej wynika jednoznacznie z wielkości drugiego argumentu, którym jest rejestr.

Należy jeszcze kolejny raz przypomnieć, że standardowo argumenty w pamięci adresowane w sposób pokazany w tabeli 3.4 znajdują się w segmencie danych wskazywanym przez zawartość rejestru DS, z wyjątkiem tych, dla których w obliczaniu adresu efektywnego uczestniczy rejestr bazowy BP. W tym przypadku obliczony adres efektywny dotyczy argumentu w segmencie stosu wskazywanym przez zawartość rejestru SS. W każdym przypadku to standardowe przyporządkowanie segmentów możemy zmienić, dopisując do wyrażenia nazwę rejestru segmentowego zakończoną dwukropkiem, tak jak pokazano to w niektórych przykładach z ostatniej kolumny w tabeli 3.4. Warto jednak pamiętać, że powoduje to powiększenie instrukcji procesora o jednobajtowy przedrostek.

---

<sup>11</sup> Mieszanie 16- i 32-bitowych trybów omówione zostanie w następnym podrozdziale.

Tabela 3.4. Tryby adresowania w procesorze 8086

Tryb	Składnia	Adres efektywny	Przykład
Przez przemieszczenie		Wartość przemieszczenia wyrażona liczbą bądź przez nazwę symboliczną	mov bx, ds:[10] add cx, licznik mov zmienna, ax
Pośrednio przez rejestr bazowy lub indeksowy	[BX] [BP] [DI] [SI]	Zawartość rejestru	mov dl, [bx] add bx, [bp] mov [di], ah add [si], cx mov ax, ds:[bp]
Pośrednio przez rejestr indeksowy i bazowy	[BX][DI] lub [BX+DI] [BP][DI] lub [BP+DI] [BX][SI] lub [BX+SI] [BP][SI] lub [BP+SI]	Suma zawartości obu rejestrów	add bx, [bp+di] mov byte ptr [bx][si], 15 add cs:[bx+si], ax mov ax, [bp][si]
Pośrednio przez rejestr bazowy, indeksowy i przemieszczenie (displacement)	disp[BX][DI] lub [BX+DI + disp] disp [BP][DI] lub [BP+DI + disp] disp [BX][SI] lub [BX+SI + disp] disp [BP][SI] lub [BP+SI + disp]	Suma zawartości rejestrów indeksowego i bazowego oraz przemieszczenia (displacement)	mov ax, zmienna[bx][di] add cx, [bp+si+8] add tabela[bx+si+2], al mov 6[bp][di], ch



Wskazówka

Pracując w trybie 16-bitowym, czyli używając do generowania adresu efektywnego rejestrów 16-bitowych, warto pamiętać następujący schemat ilustrujący wszystkie możliwości generowania adresu efektywnego:

$$\begin{bmatrix} \text{SI} \\ \text{DI} \end{bmatrix} + \begin{bmatrix} \text{BX} \\ \text{BP} \end{bmatrix} + \begin{bmatrix} \text{none} \\ 8 \text{ bit} \\ 16 \text{ bit} \end{bmatrix}$$

Offset = Index + Base + Displacement

W wyrażeniu adresowym może (ale nie musi) wystąpić po jednym elemencie z każdej kolumny. Jeżeli w wyrażeniu występuje rejestr BP, to obliczamy adres efektywny dla argumentu w segmencie stosu (SS), w pozostałych przypadkach — w segmencie danych (DS).

Przejdźmy teraz do 32-bitowego adresowania w procesorach x86-32. Procesory te, przy niezmienionej samej idei obliczania adresu efektywnego z wykorzystaniem rejestrów indeksowych, bazowych i przemieszczenia, dają znacznie większe możliwości poprzez fakt, iż w obliczaniu adresu efektywnego może uczestniczyć każdy z 32-bitowych rejestrów procesora: EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP. Każdy z wymienionych rejestrów może pełnić funkcję rejestru bazowego bądź indeksowego. Wyjątkiem jest



rejestr ESP, który nie może pełnić roli rejestru indeksowego. Dodatkowo zawartość rejestru traktowanego jako indeksowy może być przemnożona przez współczynnik skali o wartości 1, 2, 4 lub 8. Przeszczenie, jeżeli występuje, może być 8-, 16- lub 32-bitowe. Nie można mieszać w wyrażeniu adresowym rejestrów 16- i 32-bitowych. Poniżej pokażemy przykłady poprawnych instrukcji procesora wykorzystujących adresowanie argumentów w pamięci za pomocą 32-bitowych rejestrów:

```
.386 ; będziemy stosować rozkazy procesora 80386
mov eax, [edx+10]
mov esi, [edx][eax]
mov tablica[ecx+ebp], bl
add [esi*2], eax
add eax, tablica[ecx*4][edx+2]
add bx, es:[eax+ebp+1]
add al, [ecx*1] ; to też jest poprawne, ECX pełni rolę
; rejestru indeksowego
```

Reguły przyporządkowania segmentów w 32-bitowym trybie adresowania można zapisać w następujących punktach:

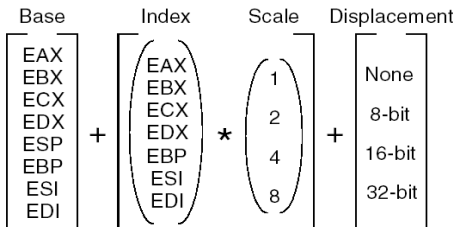
- ♦ jeśli rejestrem bazowym jest EBP lub ESP, to standardowym rejestrem segmentowym jest SS, we wszystkich pozostałych przypadkach jest to rejestr DS,
- ♦ jeżeli w wyrażeniu występują dwa rejestry, tylko jeden z nich może mieć współczynnik skalowania; rejestr ze współczynnikiem skalowania jest wtedy rejestrem indeksowym,
- ♦ jeżeli skalowanie nie jest stosowane, to pierwszy rejestr w wyrażeniu jest rejestrem bazowym.

Powyższe zasady ilustrują następujące przykłady:

```
mov eax,[edx] ; EDX jest rejestrem bazowym – segment DS
mov eax,[ebp][edx] ; EBP jest rejestrem bazowym (jest pierwszy) – segment SS
mov eax,[edx][ebp] ; EDX jest rejestrem bazowym (jest pierwszy) – segment DS
mov eax,[edx*2][ebp] ; EBP jest rejestrem bazowym (nie skalowany) – segment SS
mov eax,[edx][ebp*8] ; EDX jest rejestrem bazowym (nie skalowany) – segment DS
```



Sposoby obliczania adresu efektywnego w trybie 32-bitowym można przedstawić za pomocą następującego schematu:



Offset = Base + (Index \* Scale) + Displacement

Podobnie jak w schemacie dla adresowania 16-bitowego, w wyrażeniu adresowym może wystąpić po jednym elemencie z każdej kolumny.

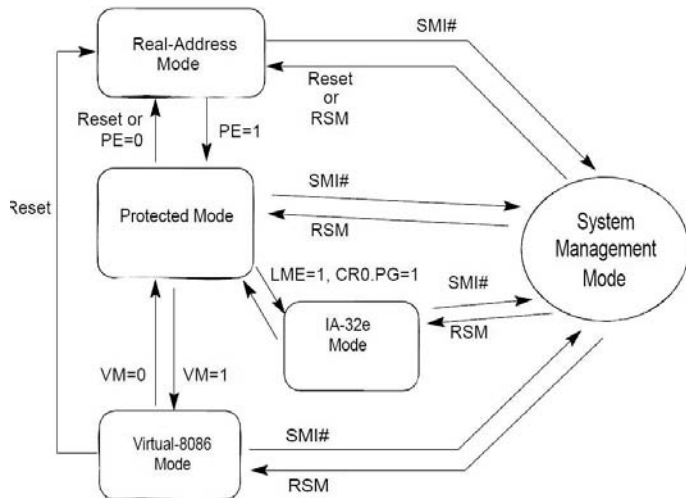
## 3.7. Architektura x86-32e

Współczesne procesory x86-32, począwszy od roku 2003 (niektóre modele procesora Pentium 4 oraz wszystkie następne, takie jak: Core 2 oraz Core i7), posiadają dodatkowy 64-bitowy tryb pracy. Warto zatem przynajmniej zasygnalizować, jakie nowe możliwości pojawiają się przed programistą w języku asemblera, gdy odpowiednie narzędzia staną się dostępne w takim samym stopniu jak narzędzia 32-bitowe.

Architekturę 64-bitową wprowadziła do procesorów o architekturze x86-32 po raz pierwszy firma AMD, nazywając ją x86-64. Firma Intel, wprowadzając do swoich procesorów to rozwiązanie, użyła początkowo nazwy x86-32e dla podkreślenia, że jest to architektura „rozszerzona” (ang. *extension*). Inna nazwa tego rozszerzenia używana przez firmę Intel to EM64T (ang. *Extended Memory 64 Technology*). Obecnie oficjalnie używana jest nazwa Intel 64, której nie należy mylić z architekturą IA-64, w oparciu o którą zbudowana jest zupełnie inna linia procesorów opracowanych przez firmy Hewlett-Packard oraz Intel, o nazwie Itanium.

Rysunek 3.15, zaczerpnięty z dokumentacji firmy Intel, ilustruje umiejscowienie trybu x86-32e oraz możliwe przejścia między poszczególnymi trybami. Nie zagłębiając się w szczegóły niebędące przedmiotem rozważań w tej książce, możemy jednak zauważyć, że „dojście” do trybu x86-32e odbywa się od trybu adresacji rzeczywistej (od którego procesor rozpoczyna pracę po podłączeniu do zasilania) poprzez 32-bitowy tryb adresacji wirtualnej z ochroną (*Protected Mode*), będący głównym przedmiotem zainteresowania w tej książce.

**Rysunek 3.15.**  
*Przejścia między poszczególnymi trybami procesora o architekturze x86-32 (na podstawie dokumentacji firmy Intel)*



W trybie 64-bitowym programista otrzymuje powiększone do 64 bitów rejestry ogólnego przeznaczenia RAX, RBX, RCX, RDX, RSI, RDI, RBP, RSP, na których mapowane są omówione w poprzednich podrozdziałach 32-bitowe rejestry, odpowiednio: EAX, EBX, ECX, EDX, ESI, EDI, EBP oraz ESP. Dodatkowo do dyspozycji mamy 8 nowych 64-bitowych rejestrów ogólnego przeznaczenia, o nazwach R8 – R15, i mapowane na



# Skorowidz

\$, 343  
%OUT, 334  
.186, 319  
.286, 319  
.286P, 319  
.287, 319  
.386, 22, 320  
.386P, 320  
.387, 320  
.486, 320  
.486P, 320  
.586, 320  
.586P, 320  
.686, 320  
.686P, 320  
.8086, 320  
.8087, 320  
.ALPHA, 321  
.BREAK, 197, 329  
.CODE, 28, 106, 321  
.CONST, 107, 321  
.CONTINUE, 197, 329  
.CREF, 191, 330  
.DATA, 22, 321  
.DATA?, 107, 321  
.def, 246  
.DOSSEG, 321  
.ELSE, 329  
.ENDIF, 329  
.ENDW, 329  
.ERR, 333  
.ERRB, 333  
.ERRDEF, 333  
.ERRDIF, 334  
.ERRE, 334  
.ERRIDN, 334  
.ERRNB, 334  
.ERRNDEF, 334  
.ERRNZ, 334  
.EXE, 194  
.EXIT, 24, 26, 230, 334  
.FARDATA, 107, 322  
.FARDATA?, 322  
.IF, 197, 329  
.INC, 189  
.K2D, 320  
.LALL, 330  
.LFCOND, 165  
.LIB, 189  
.LIST, 191, 331  
.LISTALL, 331  
.LISTIF, 165, 331  
.LISTMACRO, 331  
.LISTMACROALL, 331  
.MMX, 321  
.MODEL, 22, 322  
    BASIC, 106  
    C, 106  
    COMPACT, 106  
    FARSTACK, 106  
    FLAT, 106  
    FORTRAN, 106  
    HUGE, 106  
    LARGE, 106  
    MEDIUM, 106  
    NEARSTACK, 106  
    OS\_DOS, 106  
    OS\_OS2, 106  
    PASCAL, 106  
    SMALL, 106  
    STDCALL, 106  
    SYSCALL, 106  
    TINY, 106  
.NO87, 321  
.NOCREF, 191, 331  
.NOLIST, 191, 331

.NOLISTIF, 165, 331  
 .NOLISTMACRO, 331  
 .RADIX, 126, 323  
 .REPEAT, 197, 329  
 .SALL, 332  
 .SEQ, 322  
 .SFCOND, 332  
 .STACK, 23, 107, 323  
 .STARTUP, 22, 26, 107, 182, 230, 335  
 .TFCOND, 332  
 .TYPE, 340  
 .UNTIL, 180, 329  
 .UNTILCXZ, 329  
 .WHILE, 197  
 .XALL, 332  
 .XCREF, 191, 332  
 .XLIST, 191, 332  
 .XMM, 321  
 :REQ, 327  
 :VARARG, 328  
 ?, 343  
 @@:, 343  
 @B, 343  
 @CatStr, 168, 343  
 @code, 343  
 @CodeSize, 343  
 @Cpu, 343  
 @CurSeg, 343  
 @data, 344  
 @DataSize, 344  
 @Date, 344  
 @Environ, 344  
 @F, 344  
 @fardata, 344  
 @fardata?, 344  
 @FileCur, 344  
 @FileName, 344  
 @InStr, 168, 344  
 @Line, 344  
 @Model, 344  
 @SizeStr, 168, 344  
 @stack, 345  
 @SubStr, 168, 345  
 @Time, 345  
 @Version, 345  
 @WordSize, 345  
 80286, 39, 208  
 80386, 39  
 80486, 208  
 8086, 208  
 8087, 39

## A

AAA, 86  
 ABS, 229, 338  
 ADC, 86  
 ADD, 86  
 ADDR, 28, 338  
 adres  
   efektywny, 49  
   fizyczny, 51  
   logiczny, 44  
   powrotu, 139  
   symboliczny, 28, 140  
   wyrównany, 212  
 adresowanie  
   8086, 50  
   argumentów, 48  
   argumentów w pamięci operacyjnej, 49  
   Pentium, 41  
   porty we-wy, 48  
   rzeczywiste, 34  
   wirtualna z ochroną, 52  
   wirtualne, 38  
 AF, 36  
 ALIGN, 212, 321  
 alokacja pamięci, 183  
 AMD, 52, 317  
 AND, 95, 338  
 AND NOT, 95  
 ANSII, 188  
 API, 26, 188  
 architektura procesorów  
   EM64T, 52  
   IA-64, 52  
   Intel Core 2, 52  
   Intel Core i7, 34, 52  
   MMX, 41  
   Pentium 4, 42, 52  
   Pentium II, 41  
   Pentium MMX, 41  
   SIMD, 42  
   SSE, 42  
   SSE2, 42  
   SSE3, 42  
   x86-32, 33  
   x86-32e, 52  
 ASCII, 23, 69, 86, 118  
   kody, 405  
 asemblacja warunkowa, 163  
 asembler  
   korzyści, 15  
   zalety, 15  
 ASSUME, 334  
 AVX (Advanced Vector Extensions), 102

**B**

benchmark, 218  
 BIOS  
   przerwania, 371  
   karta graficzna, 174, 373  
   klawiatura, 174, 371  
 bitmapa, 257  
 BST\_CHECKED, 392  
 BST\_INDETERMINATE, 392  
 BST\_UNCHECKED, 392  
 BYTE, 23, 124, 324

**C**

C++, 239  
   wstawki asemblerowe, 239  
 CALL, 29, 138  
 Callback-Function, 195  
 CARRY?, 341  
 casemap, 27  
 CATSTR, 168, 327  
 CF, 36  
 Child-object, 194  
 Client-Area, 194  
 cmd, 21  
 CMP, 197  
 COMM, 324  
 COMMENT, 330  
 CPUID, 213  
 CreateWindowEx, 270  
 CreateWindowsEx, 196  
 czarno-białe, 289

**D**

data, 28  
 DATA, 106  
 DB, 324  
 DD, 124, 324  
 debugger, 62  
 debugowanie, 62  
   Microsoft Code View, 64  
   Microsoft WinDbg, 67  
   OllyDbg, 68  
 DefWindowProc, 196  
 DF, 37, 324  
 DirectX, 188  
 DIV, 86  
 DLL (Dynamic Link Library), 188, 245  
 DOSSEG, 321  
 DQ, 124, 324  
 DT, 124, 324

DUP, 338  
 DW, 124, 324  
 DWORD, 124, 324

**E**

EAX, 223  
 ECHO, 333  
 edytor graficzny, 273, 307  
   czarno-białe, 289  
   negatyw, 277  
   OpenGL, 270  
   pastele, 286  
   rozmycie, 282  
   szarość, 287  
   wyostrzenie, 289  
 EFLAGS, 38  
 ELSE, 326  
 ELSEIF, 326  
 EM64T, 52  
 END, 28, 230, 323  
 ENDF, 326  
 ENDM, 327, 328  
 ENDP, 141, 329, 330  
 ENDS, 322, 325  
 ENTER, 149  
 EQ, 339  
 equ, 168  
 EQU, 322, 323  
 ESP, 138  
 EVEN, 212, 322  
 EXITM, 327  
 ExitProcess, 27, 196  
 EXTERN, 228, 332  
 EXTERNDEF, 230, 332  
 EXTRN, 333

**F**

FAR, 229, 324  
 FIFO, 195  
 finit, 242  
 FLAGS, 36  
 FOR, 166, 327  
 FORC, 166, 327  
 fraktale, 258  
 funkcja przywołująca, 195  
 funkcje BIOS, 171  
 funkcje MS DOS, 171  
   czas, 389  
   data, 389  
   dysk, 381  
   katalogi, 381

funkcje MS DOS  
 odczyt i zapis znaku, 379  
 operacje na rekordach w pliku, 385  
 pamięć operacyjna, 386  
 pliki, 383  
 sterowanie programem, 388  
 systemowe, 387  
 fwait, 242  
 FWORD, 124, 324

**G**

GE, 339  
 generator okien, 199, 265  
 GetLocalTime, 267  
 GetMessage, 196  
 GetTimeFormat, 267  
 GOTO, 327  
 GROUP, 322  
 GT, 339

**H**

handle, 194  
 hCursor, 196  
 hIcon, 196  
 HIGH, 339  
 HIGHWORD, 339  
 HLL, 197  
 hotspots, 218

**I**

i486, 40  
 IA-64, 52  
 IDE, *Patrz* środowiska zintegrowane  
 IDIV, 86  
 IF, 37, 163, 326  
 IFB, 163, 326  
 IFDEF, 163, 326  
 IFDIF, 163, 326  
 IFE, 163  
 IFIDN, 163  
 IFNB, 163, 326  
 IFNDEF, 163, 326  
 IMUL, 86  
 INCLUDE, 27, 333  
 INCLUDELIB, 28, 333  
 InitScene, 271  
 INSTR, 168, 327  
 instrukcje MMX  
 dane spakowane, 358  
 konwersja, 357

kopiowanie, 357  
 operacje logiczne, 359  
 porównania, 359  
 przesunięcia, 359  
 instrukcje  
 jednostki stałoprzecinkowe, 84  
 koprocessor arytmetyczny, 87  
 MMX, 90  
 SSE, 93  
 SSE2, 97  
 SSE3, 100  
 systemowe, 101  
 instrukcje koprocessora arytmetycznego  
 arytmetyka, 354  
 kopiowanie, 354  
 ładowanie stałych, 356  
 porównania, 355  
 sterujące, 357  
 transcendentalne, 356  
 instrukcje procesora  
 arytmetyka dziesiętna, 349  
 arytmetyka binarna, 349  
 bitowe, 349  
 kopiowanie, 347  
 łańcuchy, 352  
 operacje logiczne, 349  
 przesunięcia, 349  
 rejestr znaczników, 353  
 skoki, 351  
 instrukcje SSE  
 dane spakowane, 360  
 konwersja danych, 362  
 kopiowanie, 360  
 operacje logiczne, 361  
 pamięć podręczna, 363  
 porównania, 361  
 rejestr sterujący, 362  
 rozpakowywanie, 361  
 tasowanie, 361  
 instrukcje SSE2  
 dane spakowane, 364  
 konwersja danych, 365  
 kopiowanie, 363  
 operacje logiczne, 364  
 porównania, 365  
 rozpakowywanie, 365  
 tasowanie, 365  
 instrukcje SSE3  
 konwersja danych, 367  
 wartość 128 bit, 367  
 instrukcje systemowe, 368  
 int, 23, 24  
 INT, 138

Intel, 317  
 INTO, 138  
 INVOKE, 28, 144, 189, 232, 330  
 IRP, 166, 327  
 IRPC, 166, 327

**J**

jednostki stałoprzecinkowe, 84  
 język  
   maszynowy, 13  
   wewnętrzny, 13  
   asemblera, 12  
   wysokiego poziomu, 12  
 JMP, 197  
 Jxxx, 197

**K**

kernel32.inc, 27  
 klawiatura, 391  
   kody, 405  
 kody  
   ASCII, 405  
   klawiszy, 405  
   kolory, 176  
 kolejka komunikatów, 195  
 kolory, 176  
 komunikaty, 400  
 koprocessor arytmetyczny, 39  
   instrukcje, 87

**L**

L1, 213  
 L2, 213  
 LABEL, 323  
 LALL, 161  
 langtype, 228  
 LE, 339  
 LENGTH, 339  
 LENGTHOF, 129, 339  
 LFCOND, 331  
 LIFO (Last In First Out), 137  
 LINK.EXE, 25, 189  
 linker, 60, 189  
 LISTMACROALL, 161  
 LOCAL, 162, 328, 330  
 LOW, 339  
 LOWWORD, 339  
 LT, 339

**Ł**

łańcuchy, 128

**M**

MACRO, 157, 328  
 Main, 196  
 makroinstrukcje, 157  
   łańcuchy, 168  
   niedefiniowane, 166  
   tekstowe, 167  
   zalety, 161  
 MASK, 339  
 MASM, 56, 318  
   dyrektywy, 319  
   operatory, 337  
   pseudoinstrukcje, 319  
   symbole predefiniowane, 343  
 MASM32, 189, 318  
 MASM32 SDK, 191  
 MessageBox, 27, 28  
 Message-Loop, 195  
 Message-Queue, 195  
 Microsoft Code View, 64  
 Microsoft Visual Studio, 239, 293  
   edytor graficzny, 307  
   steganografia, 312  
   szyfrowanie, 301  
   tworzenie projektu, 293  
 Microsoft WinDbg, 67  
 ML.ERR, 24  
 ML.EXE, 24  
 MMX, 273  
   instrukcje, 90  
   rejestry, 41  
 MOD, 339  
 model pamięci  
   płaski, 27  
   small, 22  
 moduły, 227  
   asemblacja, 231  
   języki wyższego poziomu, 232  
   konsolidacja, 230  
   nazwy globalne, 228  
   połączenia, 228  
   różnojęzykowe, 232  
 MOV, 23, 222  
 MOVAPS, 272  
 MOVNTQ, 224  
 MOVQ, 224  
 MOVSB, 221  
 MOVSD, 221  
 MOVUPS, 272



MS DOS  
 fraktale, 258  
 funkcje, 171  
 podprogramy, 180  
 prosty program, 22  
 przykładowe programy, 251  
 tryb graficzny, 252

MS DOS., 317  
 MsgLoop, 196  
 MUL, 86  
 MW\_DESTROY, 196

**N**

NAME, 333  
 NE, 339  
 NEAR, 142, 229, 324  
 negatyw, 277  
 NONUNIQUE, 325  
 NOT, 340

**O**

obiekty  
 dzieci, 194  
 rodzice, 194  
 OF, 36  
 offset, 23, 29  
 OFFSET, 340  
 okno, 396  
 OllyDbg, 67, 68, 318  
 OPATTR, 340  
 OpenGL, 188, 270  
 OPTION, 335  
 optymalizacja, 207  
 analizowanie, 218  
 grupowanie odczytu, 223  
 grupowanie zapisu, 223  
 miejsca krytyczne, 218  
 MMX, 223  
 MOV, 222  
 MOVNTQ, 224  
 MOVSB, 221  
 MOVSD, 221  
 pamięć, 211  
 pamięć podręczna, 213  
 pierwsze wykonanie, 216  
 rozgałęzienia, 215  
 rozwijanie pętli, 216  
 SFENCE, 224  
 SSE, 225  
 wspieranie, 218  
 x86-32, 211

OR, 95, 340  
 ORG, 322  
 organizacja pamięci, 44  
 OVERFLOW?, 341  
 OWORD, 124

**P**

PAGE, 331  
 PAGE+, 331  
 Paint\_Proc, 267  
 pamięć  
 alokacja, 183  
 modele, 106  
 COMPAT, 106  
 FLAT, 106  
 HUGE, 106  
 LARGE, 106  
 MEDIUM, 106  
 SMALL, 106  
 TINY, 106  
 podręczna, 213  
 Parent-object, 194  
 PARITY?, 341  
 pastele, 286  
 pętla komunikatów, 195  
 PF, 36  
 podprogramy, 137  
 MS DOS, 180  
 organizacja, 140  
 parametry, 146  
 wywołanie, 140  
 pola bitowe, 133  
 POP, 138  
 POPAD, 138  
 POPCONTEXT, 335  
 POPF, 138  
 POPFD, 138  
 PostQuitMessage, 196  
 poziomy systemu komputerowego, 12  
 PROC, 141, 189, 232, 330  
 proces, 399  
 Prostart, 199, 265  
 PROTO, 189, 196, 330  
 przzerwania, 24, 171  
 karta graficzna, 174  
 klawiatura, 174  
 przestrzeń klienta, 194  
 PTR, 340  
 PUBLIC, 228, 234, 333  
 PURGE, 328  
 PUSH, 29, 138  
 PUSHA, 138  
 PUSHAD, 138

PUSHCONTEXT, 335  
 PUSHF, 138  
 PUSHFD, 138

**Q**

QWORD, 124, 325

**R**

READONLY, 322  
 REAL10, 127, 324  
 REAL4, 127, 324  
 REAL8, 127  
 RECORD, 134, 325  
 RegisterClass, 270  
 RegisterWinClass, 196  
 regpushed, 169  
 rejestry
 

- 8086, 34
- AX, 35
- BP, 35
- BX, 35
- CX, 35
- DI, 35
- DX, 35
- FLAGS, 36
- koprocesor arytmetyczny, 39
- MMX, 41
- ogólnego przeznaczenia, 53
- segmentowe, 45
- SI, 35
- SP, 35
- XMM
  - typy danych, 43
- YMM, 102

 REPEAT, 166, 180, 328  
 REPT, 166, 328  
 RestoreRegs, 169  
 RET, 138  
 RETI, 138  
 rozmycie, 282

**S**

SaveRegs, 169  
 SBB, 86  
 SBYTE, 124, 325  
 SDWORD, 124, 325  
 SEG, 340  
 SEGMENT, 322  
 SF, 36  
 SFENCE, 224

SHL, 340  
 SHORT, 340  
 ShowWindow, 196  
 SHR, 340  
 SIGN?, 341  
 SIMD, 41  
 SIZE, 340  
 SIZEOF, 129, 340  
 SIZESTR, 168, 328  
 SP, 138  
 SSE, 42
 

- instrukcje, 93

 SSE2, 42
 

- instrukcje, 97

 SSE3, 42
 

- instrukcje, 100

 STACK, 137  
 start:, 28  
 stdcall, 27  
 steganografia, 312  
 stos, 137  
 STRUC, 325  
 STRUCT, 131, 325  
 struktury programowe, 197  
 SUB, 86  
 SUBSTR, 168, 328  
 SUBTITLE, 332  
 SUBTTL, 332  
 SWORD, 124, 325  
 symbole predefiniowane, 343  
 SYSTEMTIME, 267  
 szyfrowanie, 301

**Ś**

środowiska zintegrowane, 70
 

- MASM32 SDK, 71
- Microsoft Programmer's WorkBench (PWB), 70
- Microsoft Visual Studio, 75
- RadASM, 74
- WinAsm Studio, 74

**T**

tablice, 128  
 TBYTE, 124, 325  
 TEST, 197  
 TEXTEQU, 140, 167, 328  
 TextOut, 267  
 TF, 37  
 THIS, 340  
 TITLE, 332

tryb graficzny, 252  
 TYPE, 129, 341  
 TYPEDEF, 323

## U

uchwyt, 194, 392  
 UNICODE, 69, 188  
 UNION, 131, 325  
 UpdateWindow, 196  
 user32.inc, 27

## V

vararg, 169  
 Via Technologies, 317  
 Vtune, 218, 318  
   hotspots, 218  
   miejsca krytyczne, 218

## W

WHILE, 166, 328  
 WIDTH, 341  
 Windows  
   API, 188  
   biblioteki systemowe, 188  
   Callback-Function, 195  
   CheckDlgButton, 391  
   CloseHandle, 392  
   CopyFile, 393  
   CreateFile, 394  
   CreateWindowEx, 396  
   CreateWindowsEx, 196  
   DeleteFile, 399  
   DLL, 188  
   ExitProcess, 399  
   funkcja przywołująca, 195  
   generator okien, 199  
   GetFileSize, 400  
   GetMessage, 196  
   handle, 194  
   HLL, 197  
   klawiatura, 391  
   kolejka komunikatów, 195  
   komunikaty, 195, 400  
   linker, 189  
   MessageBox, 400  
   Message-Queue, 195  
   MsgLoop, 196  
   obiekty *Patrz* obiekty  
   okno, 191, 396  
   pętla komunikatów, 195  
   pliki, 393

PostQuitMessage, 196  
 proces, 399  
 programowanie, 187  
 Prostart, 199  
 prosty program, 25  
 przestrzeń klienta, 194  
 przykładowe programy, 265  
 RegisterWinClass, 196  
 ShowWindow, 196, 403  
 struktury programowe, 197  
 uchwyt, 194, 392  
 WinMain, 196  
 WndProc, 195  
 Windows, 391  
 Windows Message, 195  
 windows.inc, 27  
 WinMain, 196  
 WM\_MOUSEMOVE, 195  
 WM\_PAINT, 267  
 WM\_TIMER, 267  
 WndProc, 195  
 WORD, 124, 325  
 wyrównany adres, 212

## X

x86-32, 208  
   optymalizacja, 211  
 XMM, 42  
 XOR, 95, 341

## Z

ZERO?, 341  
 ZF, 36, 37  
 zmienne  
   całkowite, 124  
   definiowanie, 123  
   lokalne, 155  
   łańcuchy, 128  
   poła bitowe, 133  
   różne języki, 237  
   struktury, 130  
   tablice, 128  
   zmiennoprzecinkowe, 127  
 znaczniki  
   AF, 36  
   CF, 36  
   EFLAGS, 38  
   FLAGS, 37  
   PF, 36  
   stanu, 36  
   sterujące, 37  
   ZF, 36

# Praktyczny kurs **ASEMBLERA** Wydanie II

Programowanie w języku niskiego poziomu — choć czasem nieco uciążliwe — daje bardzo dużą swobodę w kwestii wykorzystania sprzętowych zasobów komputera i oferuje niemal nieograniczoną kontrolę nad sposobem działania programu. Aplikacje napisane za pomocą asemblera są bardzo szybkie i wydajne, a ponadto wymagają o wiele mniejszej ilości pamięci operacyjnej niż analogiczny kod opracowany w językach wysokiego poziomu, takich jak C++, Java czy Visual Basic. Jeśli jesteś zainteresowany poszerzeniem swoich umiejętności programistycznych, z pewnością nadszedł czas, aby sięgnąć po asembler.

Książka „Praktyczny kurs asemblera. Wydanie II” wprowadzi Cię w podstawowe zagadnienia związane z zastosowaniem języka niskiego poziomu do programowania komputerów opartych na architekturze x86-32 procesorów Intel (oraz AMD). Poznasz sposoby wykorzystania zasobów sprzętowych, zasadę działania procesora i listę jego instrukcji. Nauczysz się też jak używać różnych trybów adresowania w celu optymalnego zarządzania zawartością rejestrów i pamięci. Dowiesz się, jak prawidłowo pisać, łączyć, kompilować i uruchamiać programy, a także poznasz praktyczne przykłady zastosowania asemblera.

- **Podstawowe informacje na temat asemblera i architektury x86-32 procesorów Intel (oraz AMD)**
- **Przegląd narzędzi przydatnych przy tworzeniu i uruchamianiu kodu**
- **Sposoby adresowania pamięci i korzystanie z rejestrów procesora**
- **Lista instrukcji procesorów o architekturze x86-32**
- **Definiowanie i używanie zmiennych**
- **Tworzenie podprogramów i makroinstrukcji**
- **Korzystanie z funkcji systemu MS DOS i BIOS-a oraz windowsowych bibliotek typu API**
- **Stosowanie asemblera do tworzenia programów uruchamianych pod systemem Windows**
- **Tworzenie asemblerowych bibliotek typu dll z wykorzystaniem środowiska Microsoft Visual Studio**
- **Przegląd metod optymalizacji kodu**
- **Praktyczne przykłady programów wykorzystujących język asemblera**

**Wykorzystaj w pełni potencjał asemblera!**

**helion.pl**  
księgarnia  
internetowa

Nr katalogowy: 5508



Księgarnia internetowa:  
<http://helion.pl>



Zamówienia telefoniczne:  
**0 801 339900**



**0 601 339900**



**Helion**

Sprawdź najnowsze promocje:

☛ <http://helion.pl/promocje>

☛ Książki najchętniej czytane:

☛ <http://helion.pl/bestsellery>

Zamów informacje o nowościach:

☛ <http://helion.pl/nowosci>

**Helion SA**

ul. Kosciuszki 1c, 44-100 Gliwice

tel.: 32 230 98 83

e-mail: [helion@helion.pl](mailto:helion@helion.pl)

<http://helion.pl>

sięgnij po **WIĘCEJ**



**KOD KORZYŚCI**

ISBN 978-83-246-2732-5



9 788324 627325

**Cena 69,00 zł**

Informatyka w najlepszym wydaniu