Praktyczny kurs

ASEMBLERA

Wydanie II

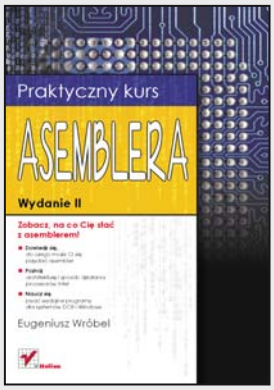
Zobacz, na co Cię stać z asemblerem!

- **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

Praktyczny kurs asemblera. Wydanie II



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

- 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

Zobacz, na co Cię stać z asemblerem!

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

Spis treści

Rozdział 3.

Architektura procesorów rodziny x86-32 widziana oczami programisty

Wszystkie instrukcje procesora[1], 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.

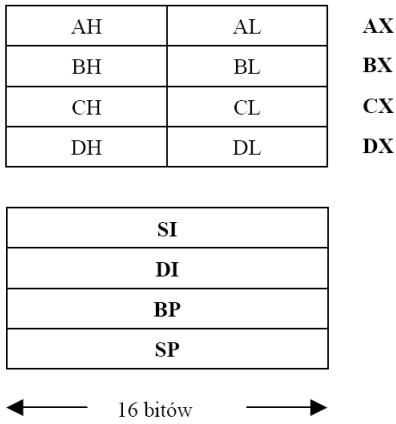
Każdy kolejny procesor firmy Intel należący do tzw. linii procesorów x86-32 był tak rozbudowywany, aby oprogramowanie działające na poprzednim, starszym modelu procesora mogło być w dalszym ciągu używane. Aby to było możliwe, w najnowszym procesorze Intel Core i7 możemy „zobaczyć" procesor 8086 i jego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

[1] 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[2]),

CX — licznik w operacjach na łańcuchach oraz w pętlach 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

[2] 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. *carry flag*)	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 + 0111 0100 1 0010 0010 CF=1
PF	znacznik parzystości (ang. *parity flag*)	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 + 1011 0001 1101 1101 PF=1
AF	znacznik przeniesienia pomocniczego (ang. *auxiliary flag*)	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 + 0110 0100 1001 0010 AF=1
ZF	znacznik zera (ang. *zero flag*)	Przyjmuje wartość 1 wtedy, gdy wynik operacji jest równy zero, i jest zerowany w przeciwnym razie.	1111 1111 + 0000 0001 0000 0000 ZF=1
SF	znacznik znaku (ang. *sign flag*)	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 + 0100 0001 1010 0001 SF=1
OF	znacznik przepełnienia (ang. *overflow flag*)	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 + 0101 1001 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

rozróżnia, 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_rowne    ; skocz do etykiety sa_rowne, 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_rowne`. 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.

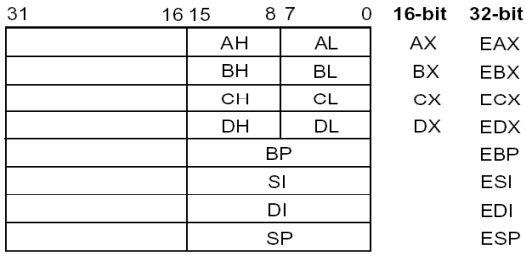
Tabela 3.2. *Znaczniki sterujące w rejestrze FLAGS*

Symbol znacznika	Nazwa znacznika	Znaczenie znacznika
TF	znacznik pracy krokowej (ang. *trap flag*)	Stan równy 1 powoduje wprowadzenie procesora w tryb pracy (ang. *single step mode*) umożliwiający wygenerowanie przerwania (ang. *single step interrupt*) 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. *interrupt flag*)	Ustawiony w stan 1 powoduje odblokowanie systemu przerwań procesora. Zewnętrzne przerwania maskowane mogą przerwać realizację wykonywanego aktualnie programu. Wyzerowanie znacznika powoduje, że przerwania te są przez procesor ignorowane.
DF	znacznik kierunku (ang. *direction flag*)	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[3]

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*

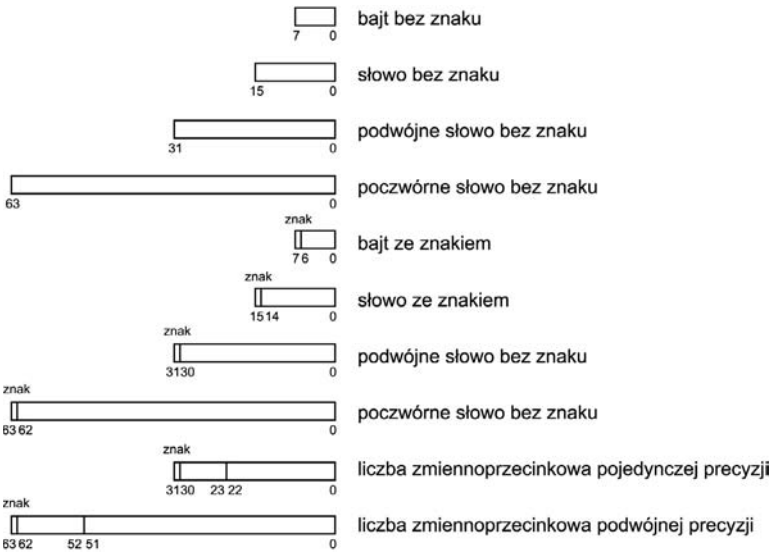
Jak widać na tym rysunku, wszystkie rejestry zostały powiększone do 32 bitów, zaś w nazwie pojawiła się na pierwszym miejscu litera „E", od angielskiego słowa *expand* — poszerzenie, powiększenie. W programie wykorzystywać możemy zarówno 32-bitowe rejestry: EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, jak 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:

[3] 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[4] do Intel Core i7[5]

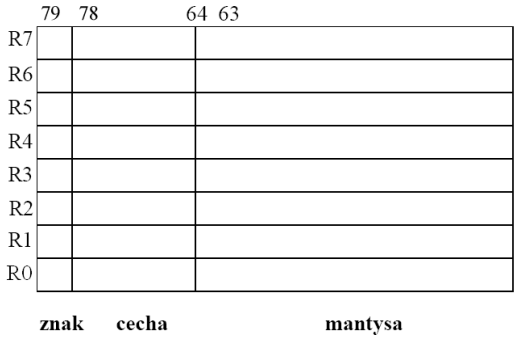
Wykonywanie złożonych obliczeń na liczbach zmiennoprzecinkowych przy wykorzystaniu dotychczas poznanych przez nas rejestrów jest uciążliwe i mało wydajne. Dlatego też równolegle do procesora 8086 powstał koprocesor arytmetyczny 8087 (jako oddzielny układ scalony), a później odpowiednio dla kolejnych 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

[4] Według wcześniejszego nazewnictwa firmy Intel — 80486.

[5] W architekturze x86-32.

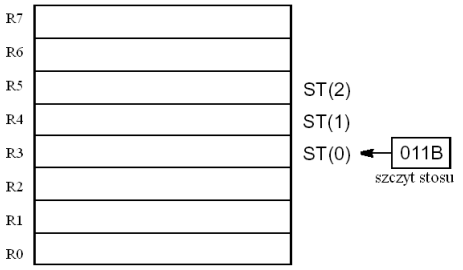
umieszczać można instrukcje wykonywane przez każdy z tych procesorów. Począwszy od procesora i486, koprocesor arytmetyczny włączony został do procesora głównego, przy zachowaniu wszystkich swoich funkcji. W procesorach o architekturze x86-32 występuje w postaci tzw. jednostki zmiennoprzecinkowej[6] —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

Rysunek 3.5. *80-bitowe rejestry koprocesora 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 koprocesora 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:

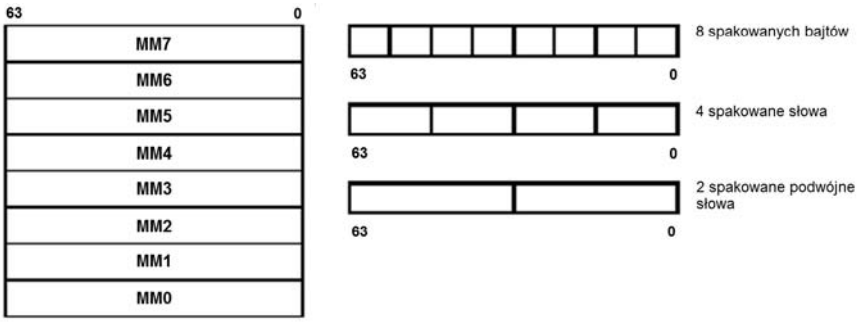
```
fldpi           ; załadowanie na szczyt stosu do rejestru ST(0)
                ; liczby stałej „pi”
fld zmienna     ; załadowanie na szczyt stosu do rejestru ST(0)
                ; zmiennej z pamięci
```

[6] Informatycy — szczególnie ci pamiętający czasy procesorów 8086, 80286 i 80386, jak i koprocesorów arytmetycznych 8087, 80287 i 80387 — w dalszym ciągu operują pojęciem *koprocesor 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)
fxch 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:

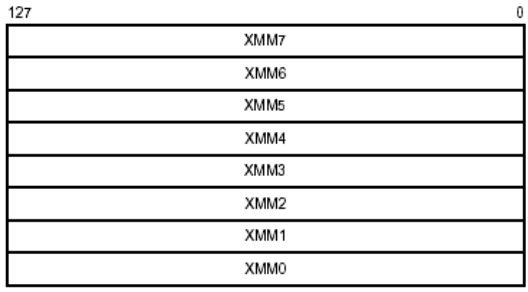
- jedną daną 64-bitową,
- dwie dane 32-bitowe (ang. *Packed Doubleword Integers*),
- cztery dane 16-bitowe (ang. *Packed Word Integers*),
- 8 danych bajtowych (ang. *Packed Byte Integers*),

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      ; przesuń logicznie zawartość (całego)
                  ; rejestru MM1 w lewo o 3 bity
```

```
paddb mm2, mm3    ; dodaj odpowiadające sobie bajty rejestrów MM2 i MM3
paddw 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 równolegle 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[7] zmianie.



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

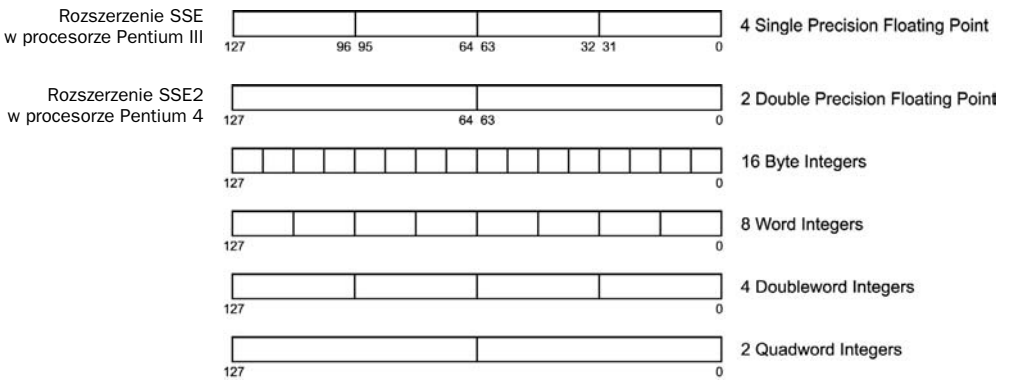
W procesorach o architekturze x86-32 w każdym ze 128-bitowych rejestrów możemy równolegle przetwarzać (rysunek 3.9):

- cztery 32-bitowe liczby zmiennoprzecinkowe pojedynczej precyzji[8],
- dwie 64-bitowe liczby zmiennoprzecinkowe podwójnej precyzji,
- szesnaście bajtów[9] traktowanych jako liczby stałoprzecinkowe,

[7] 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.

[8] 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.

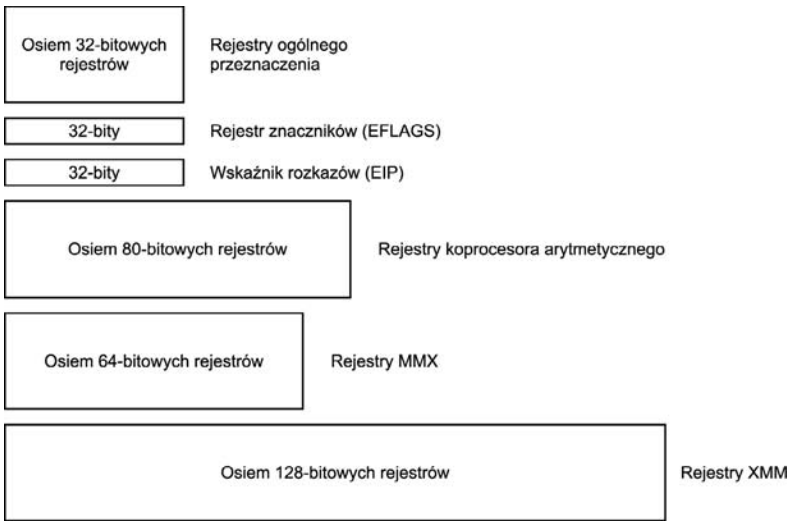
[9] Ten i kolejne wymienione tutaj formaty stałoprzecinkowe są rozszerzeniem technologii MMX na rejestry XMM.



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*



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*

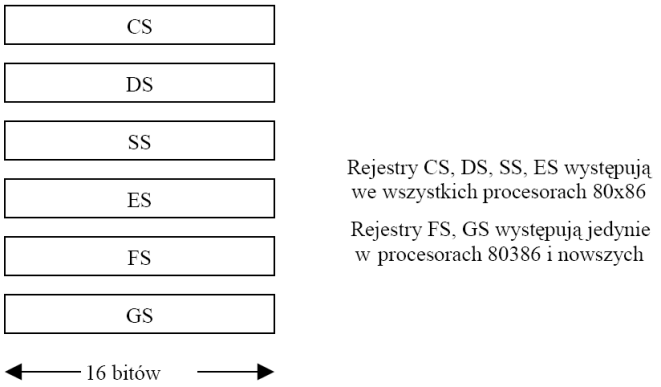
Procesor	Wielkość magistrali adresowej	Maksymalna wielkość pamięci operacyjnej	Wielkość offsetu	Maksymalna wielkość segmentu w pamięci
8086	20 bitów	1 MB	16 bitów	64 kB
80286	24 bity	16 MB	16 bitów	64 kB
Począwszy od procesora 80386 do Pentium włącznie	32 bity	4 GB	32 bity	4 GB
Począwszy od procesora 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*

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.

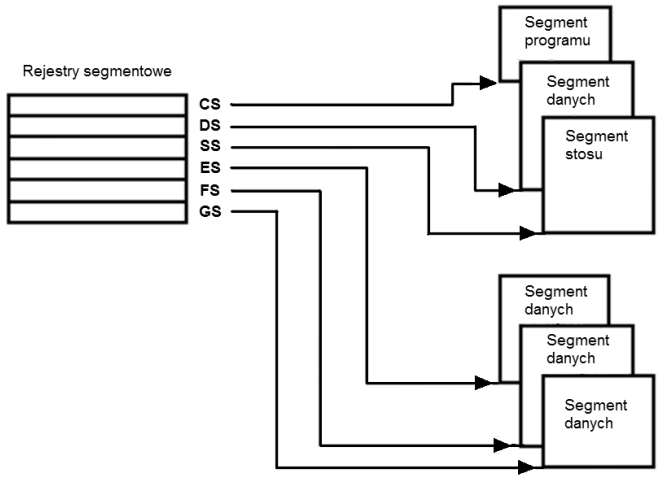
Jak już wiemy, w procesorze 8086 (lub inaczej: dla procesora x86-32 w trybie adresowania rzeczywistego) dla określenia fizycznego adresu początku segmentu potrzeba 20 bitów (porównaj tabelę 3.3). Rejestr segmentowy zawierać będzie w takim przypadku 16 starszych bitów tego 20-bitowego adresu. Brakujące 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),

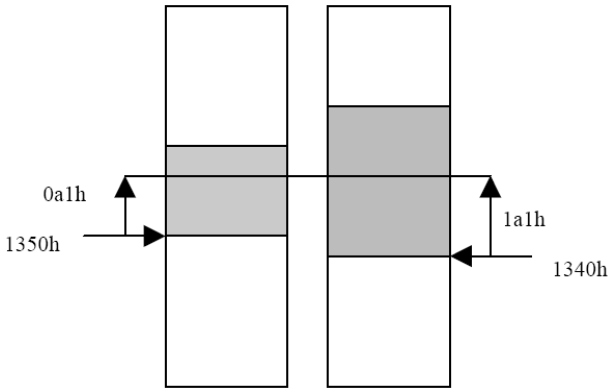
DS — wskazuje segment z danymi; większość operacji na danych będzie standardowo związana z tym segmentem,

SS — wskazuje segment stosu,

ES, FS, GS — wskazują dodatkowe segmenty danych.

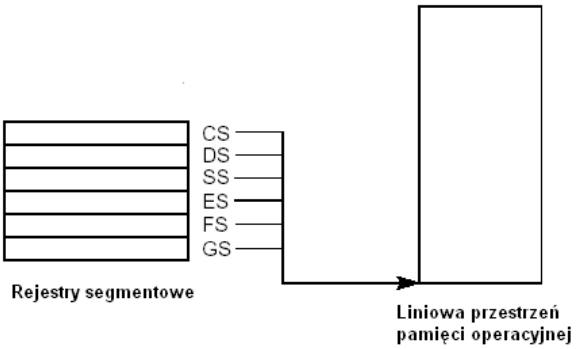
Warto zauważyć, że w trybie adresacji rzeczywistej ten sam bajt w pamięci możemy zaadresować różnymi adresami logicznymi. Występuje tutaj redundancja wynikająca z tego, że do zaadresowania przestrzeni jednomegabajtowej używamy aż 32 bitów adresu logicznego o postaci *segment:offset*. Przykład ilustrujący 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[10]. 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.

[10] 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[11],
- 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.

[11] 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

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

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

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. Przemieszczenie, 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
```

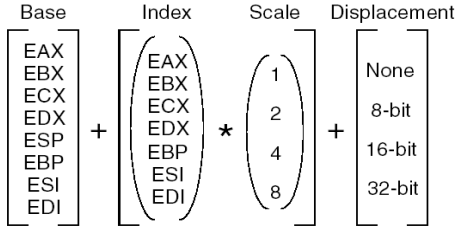
Reguły przyporządkowania segmentów w 32-bitowym trybie adresowania można zapisać w następujących punktach:

- 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 eac,[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

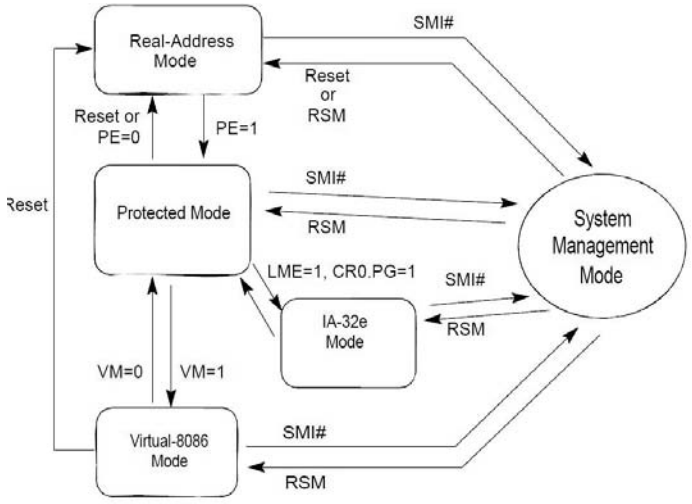
Podobnie jak w schemacie dla adresowania 16-bitowego, w wyrażeniu adresowym może wystąpić po jednym elemencie z każdej kolumny.

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 pojawią 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.

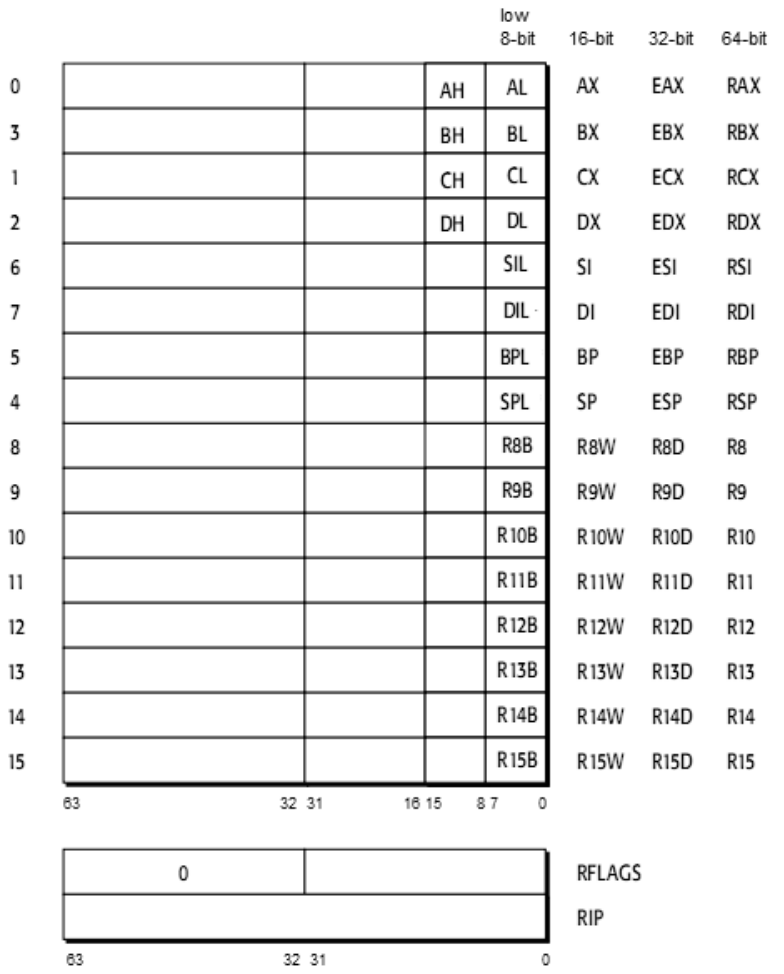
Rysunek 3.15, zaczerpnięty z dokumentacji firmy Intel, ilustruje umiejscowienie trybu x86-32e oraz możliwe przejścia między poszczególnymi trybami. Nie zagłębiając się w szczegóły niebędące przedmiotem rozważań w tej książce, możemy jednak zauważyć, że „dojście" do trybu x86-32e odbywa się od trybu adresacji 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

nich 32-bitowe rejestry, odpowiednio: R8D – R15D, oraz mapowane na nich z kolei 16-bitowe rejestry R8W – R15W i 8-bitowe rejestry R8L – R15L. Zmieniły się także nieco zasady dostępu do bajtów będących częścią powyższych rejestrów, co częściowo ilustruje rysunek 3.16. Podwojona została ponadto liczba 128-bitowych rejestrów XMM, oznaczonych jako XMM0 – XMM15.



Rysunek 3.16. *Rejestry ogólnego przeznaczenia dostępne w 64-bitowym trybie x86-32e*

W trybie 64-bitowym zwiększeniu uległa liczba dostępnych w programie asemblerowym typów danych. Praktycznie nie funkcjonuje segmentacja pamięci, choć rejestry segmentowe dalej istnieją i pełnią pewną rolę w adresowaniu pamięci. Lista rozkazów rozbudowana została o nowe rozkazy, a wiele innych — znanych z trybu 32-bitowego — pozwala teraz na operowanie argumentami 64-bitowymi. Jest też pewna liczba rozkazów, które w trybie 64-bitowym nie są dostępne.

Asembler pozwalający tłumaczyć program źródłowy dla trybu x86-32e firmy Microsoft występuje jako plik *ml64.exe* i można go znaleźć między innymi w instalacji Microsoft Visual Studio 2008.

Skorowidz

A

B

C

D

E

F

G

H

I

J

K

L

Ł

M

N

O

P

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. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

sięgnij po WIĘCEJ
KOD KORZYŚCI

ISBN 978-83-246-2732-5

Cena 69,00 zł

Informatyka w najlepszym wydaniu