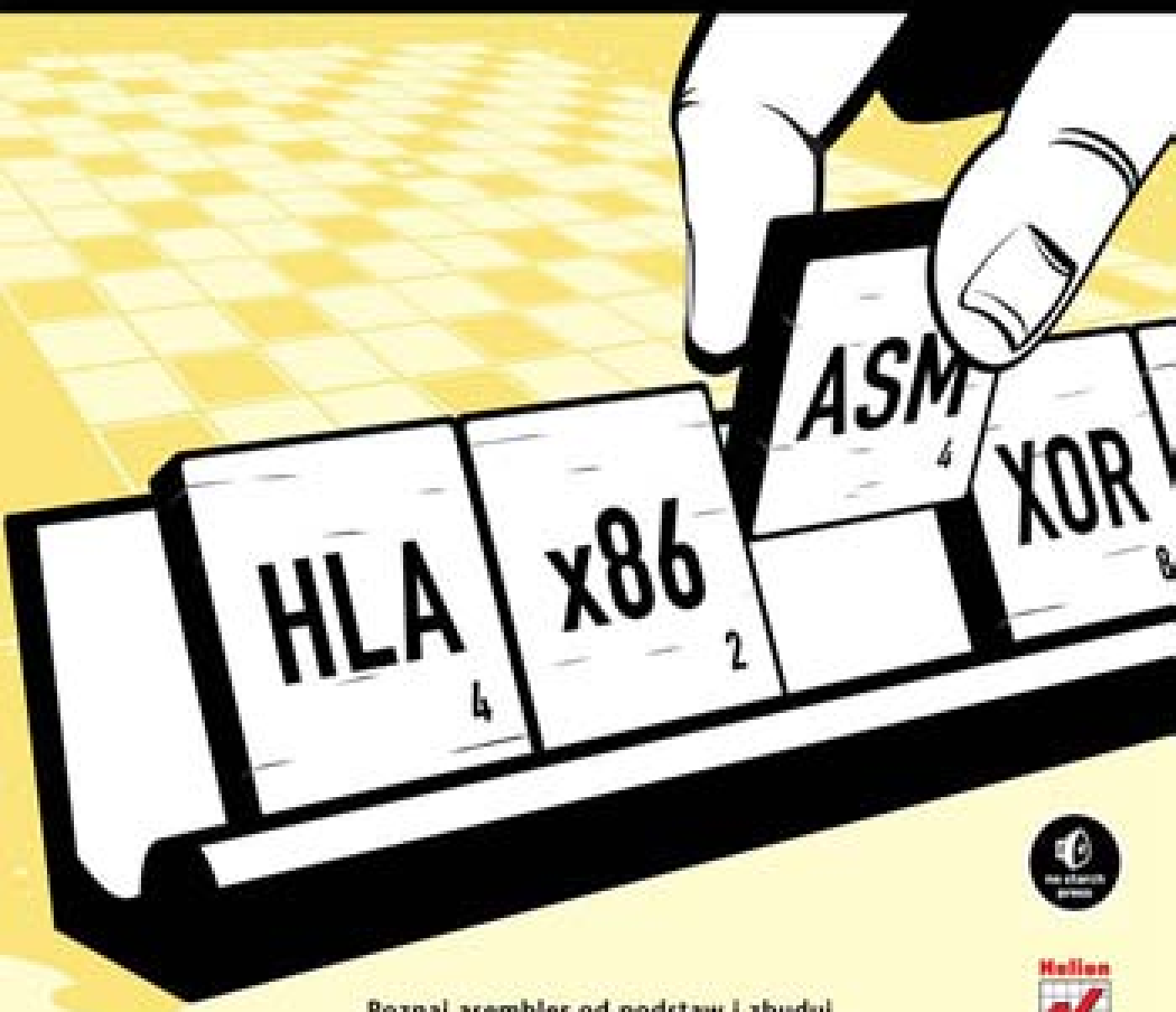
JAK PISAĆ, KOMPILOWAĆ I URUCHAMIAĆ PROGRAMY W JĘZYKU HLA?
JAK OBSŁUGIWAĆ ZBIORY ZNAKÓW W BIBLIOTECE STANDARDOWEJ HLA?
JAK OBLICZAĆ WARTOŚCI WYRAŻEŃ LOGICZNYCH?

ASEMBLER. SZTUKA PROGRAMOWANIA

WYDANIE II

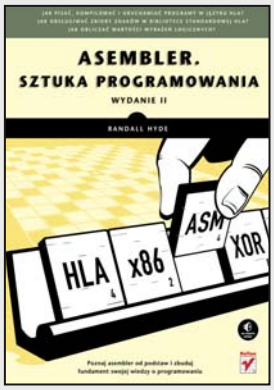
RANDALL HYDE



Poznaj asembler od podstaw i zbuduj
fundament swojej wiedzy o programowaniu

Helion

Asembler. Sztuka programowania. Wydanie II



Autor: Randall Hyde
Tłumaczenie: Przemysław Szeremiota
ISBN: 978-83-246-2854-4
Tytuł oryginału: The Art of Assembly Language, 2nd edition
Format: B5, stron: 816

Poznaj asembler od podstaw i zbuduj fundament swojej wiedzy o programowaniu

- Jak pisać, kompilować i uruchamiać programy w języku HLA?
- Jak obsługiwać zbiory znaków w bibliotece standardowej HLA?
- Jak obliczać wartości wyrażeń logicznych?

Poznanie asemblera jest jak położenie fundamentu pod budowlę całej twojej wiedzy informatycznej, ponieważ to właśnie ono ułatwia zrozumienie mechanizmów rządzących innymi językami programowania. Język asemblera, należący do języków programowania niższego poziomu, jest powszechnie stosowany do pisania sterowników, emulatorów i gier wideo. Jednak omawiany w tej książce język HLA posiada też wiele cech języków wyższego poziomu, takich jak C, C++ czy Java, dzięki czemu przy jego używaniu nie musisz rezygnować z licznych udogodnień, typowych dla takich języków.

Książka „Asembler. Sztuka programowania. Wydanie II" stanowi obszerne i wyczerpujące omówienie języka asemblera. Dzięki wielu jasnym przykładom, pozbawionym niepotrzebnej specjalistycznej terminologii, zawarty tu materiał staje się łatwo przyswajalny dla każdego, kto chciałby poznać programowanie niższego poziomu. Korzystając z tego podręcznika, dowiesz się m.in., jak deklarować i stosować stałe, zmienne skalarne, wskaźniki, tablice, struktury, unie i przestrzenie nazw. Nauczysz się realizować w języku asemblera struktury sterujące przebiegiem wykonania programu. Ponadto drugie wydanie zostało uaktualnione zgodnie ze zmianami, które zaszły w języku HLA. Uwzględnia także stosowanie HLA w kontekście systemów Windows, Linux, Mac OS X i FreeBSD.

- Wstęp do asemblera
- Anatomia programu HLA
- Reprezentacja danych
- Dostęp do pamięci i jej organizacja
- Stałe, zmienne i typy danych
- Procedury i moduły
- Niskopoziomowe struktury sterujące wykonaniem programu
- Makrodefinicje i język czasu kompilacji
- Manipulowanie bitami
- Klasy i obiekty

Podręcznik na najwyższym poziomie o językach programowania niższego poziomu

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

Spis treści

3

Dostęp do pamięci i jej organizacja

Z lektury dwóch poprzednich rozdziałów Czytelnik poznał sposób deklarowania i odwoływania się do zmiennych w programach języka asemblerowego. W rozdziale bieżącym pozna pełny obraz realizacji odwołań do pamięci w architekturze 80x86. Zaprezentowany zostanie również sposób organizacji danych pod kątem najefektywniejszego dostępu. Czytelnik dowie się też co nieco o stosie procesora 80x86 i sposobie manipulowania danymi na stosie. Rozdział zakończony zostanie omówieniem dynamicznego przydziału pamięci na **stercie**.

W tym rozdziale będziemy się zajmować kilkoma kluczowymi zagadnieniami, między innymi:

- trybami adresowania pamięci w procesorach 80x86,
- trybami adresowania indeksowanego i indeksowanego ze skalowaniem,
- organizacją pamięci,
- przydziałem pamięci do programu,
- koercją typów danych,

- stosem procesora 80x86,
- dynamicznym przydziałem pamięci.

Dowiemy się więc, jak efektywnie korzystać z zasobów pamięciowych komputera w programach pisanych w języku HLA.

3.1. Tryby adresowania procesorów 80x86

Procesory z rodziny 80x86 realizują dostęp do pamięci w kilku różnych trybach. Jak dotychczas wszystkie prezentowane odwołania do zmiennych realizowane były przez programy HLA w trybie, który określa się mianem trybu **adresowania bezpośredniego**. W tym rozdziale omówione zostaną jeszcze inne **tryby adresowania** dostępne programiście języka asemblerowego procesora 80x86. Dostępność wielu trybów adresowania pamięci pozwala na efektywny i elastyczny dostęp do pamięci, co ułatwia tworzenie zmiennych, wskaźników, tablic, rekordów i innych złożonych typów danych. Opanowanie wszystkich trybów adresowania pamięci realizowanych przez procesor 80x86 to pierwszy krok na drodze do opanowania języka asemblerowego procesorów 80x86.

Kiedy inżynierowie z firmy Intel projektowali procesor 8086, wyposażyli go w elastyczny, choć równocześnie ograniczony, zestaw trybów adresowania pamięci. Wraz z wprowadzeniem na rynek modelu 80386 zestaw ten został rozszerzony o kilka kolejnych trybów. Niemniej jednak w środowiskach 32-bitowych, czyli w systemach Windows, Mac OS X, Free BSD czy Linux, owe pierwotne tryby adresowania nie są specjalnie użyteczne. W rzeczy samej język HLA nie obsługuje nawet owych starszych, żeby nie powiedzieć przestarzałych, 16-bitowych trybów adresowania. Na szczęście wszystkie operacje możliwe do wykonania w owych trybach da się wykonać za pośrednictwem nowych, 32-bitowych trybów adresowania. Z tego względu omówienie 16-bitowych trybów adresowania można pominąć, jako że we współczesnych systemach operacyjnych są one bezużyteczne. Jedynie ci spośród Czytelników, którzy zamierzają tworzyć programy dla systemu MS-DOS i innych systemów szesnastobitowych, powinni zapoznać się z 16-bitowym adresowaniem pamięci (omówienie trybów szesnastobitowych znajduje się w „16-bitowej" wersji niniejszej książki publikowanej w wersji elektronicznej w witrynie *http://webster.cs.ucr.edu*).

3.1.1. Adresowanie przez rejestr

Większość instrukcji zestawu instrukcji maszynowych procesora 80x86 wykorzystuje w roli operandów rejestry ogólnego przeznaczenia. Dostęp do rejestru uzyskuje się, określając w miejsce operandu instrukcji nazwę rejestru. Na przykład dla instrukcji `mov` wygląda to następująco:

```
mov( operand-źródłowy, operand-docelowy );
```

Powyższa instrukcja kopiuje wartość operandu źródłowego do operandu docelowego. W szczególności operandami tymi mogą być 8-bitowe, 16-bitowe i 32-bitowe rejestry procesora.

Jedynym ograniczeniem nakładanym na takie operandy jest wymóg zgodności rozmiarów. Oto kilka przykładów zastosowania instrukcji `mov` procesora 80x86:

```
mov( bx, ax );        // Kopiowanie zawartości rejestru BX do rejestru AX
mov( al, dl );        // Kopiowanie zawartości rejestru AL do rejestru DL
mov( edx, esi );      // Kopiowanie zawartości rejestru EDX do rejestru ESI
mov( bp, sp );        // Kopiowanie zawartości rejestru BP do rejestru SP
mov( cl, dh );        // Kopiowanie zawartości rejestru CL do rejestru DH
mov( ax, ax );        // To również poprawna instrukcja!
```

Rejestry są wymarzonym miejscem do przechowywania zmiennych. Instrukcje odwołujące się do rejestrów są wykonywane szybciej od tych, które odwołują się do pamięci. Ich zapis jest też krótszy. Większość instrukcji obliczeniowych wymaga wprost, aby jeden z operandów był umieszczony w rejestrze, stąd adresowanie przez rejestr jest w kodzie asemblerowym procesora 80x86 bardzo częste.

3.1.2. 32-bitowe tryby adresowania procesora 80x86

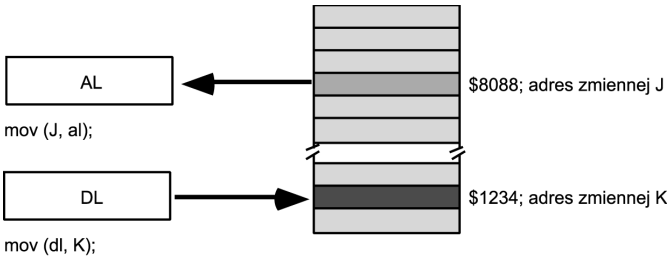
Procesor 80x86 realizuje dostęp do pamięci na setki rozmaitych sposobów. Na pierwszy rzut oka liczba trybów adresowania jest cokolwiek porażająca, ale na szczęście większość z nich to proste odmiany trybów podstawowych, stąd ich opanowanie nie przysparza większych trudności. A dobór odpowiedniego trybu adresowania to klucz do efektywnego programowania w asemblerze.

Tryby adresowania implementowane w procesorach z rodziny 80x86 obejmują adresowanie bezpośrednie, adresowanie bazowe, bazowe indeksowane, indeksowe oraz bazowe indeksowane z przemieszczeniem. Cała niezliczona reszta trybów adresowania to odmiany owych trybów podstawowych. I tak przeszliśmy od setek do zaledwie pięciu trybów. To już nieźle!

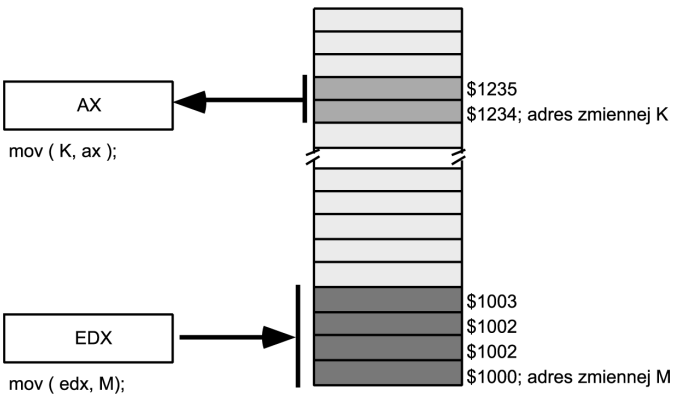
3.1.2.1. Adresowanie bezpośrednie

Najczęściej wykorzystywanym i najprostszym do opanowania trybem adresowania jest adresowanie **bezpośrednie** (ang. *displacement-only*). W tym trybie adres docelowy określany jest 32-bitową stałą. Jeśli na przykład zmienna `J` jest zmienną typu `int8` umieszczoną pod adresem $8088, to instrukcja `mov(J, al)` oznacza załadowanie do rejestru AL kopii bajta spod adresu $8088. Analogicznie, jeśli przyjąć, że zmienna `K` typu `int8` znajduje się pod adresem $1234, to instrukcja `mov( dl, K )` powoduje zachowanie wartości rejestru DL pod adresem $1234 (patrz rysunek 3.1).

Tryb adresowania bezpośredniego świetnie nadaje się do realizacji odwołań do prostych zmiennych skalarnych. Dla tego trybu przyjęto nazwę „adresowanie z przemieszczeniem", ponieważ bezpośrednio po kodzie instrukcji `mov` w pamięci zapisana jest trzydziestodwubitowa stała przemieszczenia. Przemieszczenie w procesorach 80x86 definiowane jest jako przesunięcie (ang. *offset*) od początkowego adresu pamięci (czyli adresu zerowego). W przykładach prezentowanych w tej książce znaczna liczba instrukcji to odwołania do pojedynczych bajtów w pamięci. Nie należy jednak zapominać, że w pamięci można przechowywać również obiekty rozmiarów słowa i podwójnego słowa, i również ich adres określa się, podając adres pierwszego bajta obiektu (patrz rysunek 3.2).



Rysunek 3.1. Tryb adresowania bezpośredniego



Rysunek 3.2. Odwołanie do słowa i podwójnego słowa w trybie adresowania bezpośredniego

3.1.2.2. Adresowanie pośrednie przez rejestr

Procesory z rodziny 80x86 pozwalają na odwołania do pamięci realizowane za pośrednictwem rejestru, w tak zwanym trybie adresowania pośredniego przez rejestr. Termin „pośrednie" oznacza tu, że operand nie jest właściwym adresem; dopiero wartość operandu określa adres odwołania. W adresowaniu pośrednim przez rejestr wartość rejestru to docelowy adres pamięci. Na przykład instrukcja `mov( eax, [ebx] )` informuje procesor, aby ten zachował zawartość rejestru EAX w miejscu, którego adres znajduje się w rejestrze EBX. Tryb adresowania pośredniego przez rejestr jest w języku HLA sygnalizowany nawiasami prostokątnymi.

Procesory 80x86 obsługują osiem wersji adresowania pośredniego przez rejestr; wersje te można zademonstrować na następujących przykładach:

```
mov( [eax], al );
mov( [ebx], al );
mov( [ecx], al );
mov( [edx], al );
mov( [edi], al );
mov( [esi], al );
mov( [ebp], al );
mov( [esp], al );
```

Wersje te różnią się tylko rejestrem, w którym przechowywany jest właściwy adres operandu. Wartość rejestru interpretowana jest jako przesunięcie operandu w pamięci.

W adresowaniu pośrednim przez rejestr konieczne jest stosowanie rejestrów 32-bitowych. Nie można określić przesunięcia w pamięci w rejestrze 16-bitowym ani tym bardziej w rejestrze 8-bitowym[1]. Teoretycznie 32-bitowy rejestr można załadować dowolną wartością i w ten sposób określić dowolny adres właściwego operandu:

```
mov( $1234_5678, ebx );
mov( [ebx], al );        // Próba odwołania się do adresu $1234_5678
```

Niestety (albo na szczęście) próba taka spowoduje najpewniej wygenerowanie przez system operacyjny błędu ochrony pamięci, ponieważ nie zawsze program może odwoływać się do dowolnych obszarów pamięci. Są jednak inne metody załadowania rejestru adresem pewnego obiektu; o tym później.

Adresowanie pośrednie przez rejestr ma bardzo wiele zastosowań. Można w ten sposób odwoływać się do danych, dysponując jedynie wskaźnikami na nie, można też, zwiększając wartość rejestru, przechodzić pomiędzy elementami tablicy. W ogólności tryb ten nadaje się do modyfikowania adresu docelowego odwołania w czasie działania programu.

Adresowanie pośrednie przez rejestr to przykład trybu adresowania z dostępem „w ciemno". Kiedy adres odwołania zadany jest wartością rejestru, nie ma mowy o nazwie zmiennej — obiekt docelowy identyfikowany jest wyłącznie wartością adresu. Obiekt taki można więc określić mianem „obiektu anonimowego".

Język HLA udostępnia prosty operator pozwalający na załadowanie 32-bitowego rejestru adresem zmiennej, o ile jest to zmienna statyczna. Operator pobrania adresu ma postać identyczną jak w językach C i C++ — jest to znak `&`. Poniższy przykład demonstruje sposób załadowania rejestru EBX adresem zmiennej `J`, a następnie zapisania w rejestrze EAX bieżącej wartości tej zmiennej przy użyciu adresowania pośredniego przez rejestr:

```
mov( &J, ebx );        // Załadowanie rejestru EBX adresem zmiennej J.
mov( eax, [ebx] );     // zapisanie w zmiennej J wartości rejestru EAX.
```

Co prawda łatwiej byłoby po prostu pojedynczą instrukcją `mov` umieścić wartość zmiennej `J` w rejestrze EAX, zamiast angażować dwie instrukcje po to tylko, aby zrobić to pośrednio przez rejestr. Łatwo można sobie jednak wyobrazić sekwencję kodu, w ramach której do rejestru EBX ładowany jest adres jednej z wielu zmiennych, w zależności od pewnych warunków, a potem — już niezależnie od nich — do rejestru EAX trafia wartość odpowiedniej zmiennej.

Ostrzeżenie

Operator pobrania adresu (`&`) nie jest operatorem o zastosowaniu tak ogólnym, jak jego odpowiednik znany z języków C i C++. Operator ten można w języku HLA zastosować wyłącznie

[1] Tak naprawdę procesory z rodziny 80x86 wciąż obsługują tryby adresowania pośredniego przez 16-bitowy rejestr. Tryb ten w środowisku 32-bitowym nie ma jednak zastosowania i jako taki nie jest obsługiwany w języku HLA.

do zmiennych statycznych[2]*. Nie można używać go do wyrażeń adresowych i zmiennych innych niż statyczne. Bardziej uniwersalny sposób pobrania adresu zmiennej w pamięci zostanie zaprezentowany w podrozdziale 3.13, przy okazji omawiania instrukcji ładowania adresu efektywnego.*

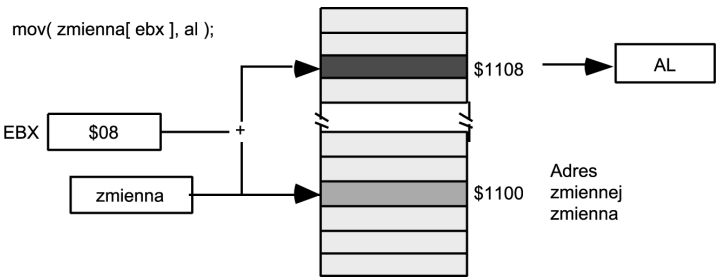
3.1.2.3. Adresowanie indeksowe

Tryb adresowania indeksowego wykorzystuje następującą składnię instrukcji:

```
mov( zmienna[ eax ], al );
mov( zmienna[ ebx ], al );
mov( zmienna[ ecx ], al );
mov( zmienna[ edx ], al );
mov( zmienna[ edi ], al );
mov( zmienna[ esi ], al );
mov( zmienna[ ebp ], al );
mov( zmienna[ esp ], al );
```

gdzie *zmienna* jest nazwą zmiennej programu.

W trybie adresowania indeksowego obliczany jest efektywny adres obiektu docelowego[3]; polega to na dodaniu do adresu *zmiennej* wartości zapisanej w 32-bitowym rejestrze umieszczonym w nawiasach prostokątnych. Dopiero suma tych wartości określa właściwy adres pamięci, do którego ma nastąpić odwołanie. Jeśli więc `zmienna` przechowywana jest w pamięci pod adresem $1100, a rejestr EBX zawiera wartość 8, to wykonanie instrukcji `mov( zmienna[ ebx ], al )` powoduje umieszczenie w rejestrze AL wartości zapisanej w pamięci pod adresem $1108. Całość została zilustrowana rysunkiem 3.3.



Rysunek 3.3. Adresowanie indeksowe

Tryb adresowania indeksowego jest szczególnie poręczny do odwoływania się do elementów tablic. Takie jego zastosowanie zostanie bliżej omówione w rozdziale 4.

---

[2] Zmienne statyczne obejmują obiekty deklarowane ze słowem kluczowym `static`, `readonly` oraz `storage`.

[3] Adres efektywny to adres ostateczny, do którego procesor odwoła się w wyniku wykonania instrukcji. Jest to więc efekt końcowy procesu ustalania adresu odwołania.

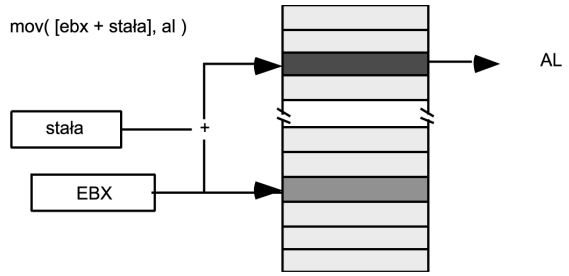
3.1.2.4. Warianty trybu adresowania indeksowego

Język HLA przewiduje dwie ważne odmiany podstawowego trybu adresowania indeksowego. Obie odmiany generują co prawda te same instrukcje maszynowe, ale ich składnia sugeruje odmienne przeznaczenie.

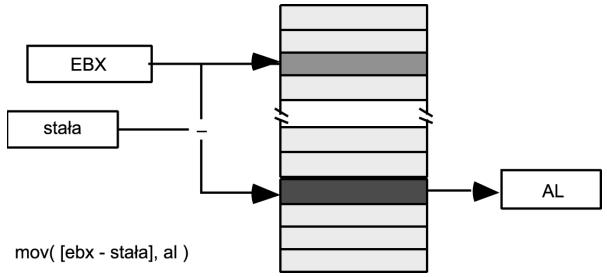
Pierwszy wariant korzysta z następującej składni:

```
mov( [ebx + stała], al );
mov( [ebx - stała], al );
```

W powyższym przykładzie wykorzystywany jest jedynie rejestr EBX, ale w trybie adresowania indeksowego można wykorzystywać wszystkie 32-bitowe rejestry ogólnego przeznaczenia. Adres efektywny jest w tym trybie wyliczany przez dodanie do zawartości rejestru EBX określonej stałej, ewentualnie odjęcie tej stałej od wartości rejestru EBX (patrz rysunki 3.4 oraz 3.5).



Rysunek 3.4. Adresowanie indeksowe: wartość rejestru plus stała



Rysunek 3.5. Adresowanie indeksowe: wartość rejestru minus stała

Ten konkretny wariant adresowania jest przydatny, jeśli 32-bitowy rejestr zawiera adres bazowy obiektu wielobajtowego i zachodzi konieczność odwołania się do adresu składowej obiektu, oddalonego od adresu bazowego o pewną liczbę bajtów. Tryb ten wykorzystuje się więc w odwołaniach do składowych (pól) struktur (rekordów), gdy struktura zadana jest wskaźnikiem. Tryb ten oddaje również nieocenione usługi w odwołaniach do zmiennych automatycznych (lokalnych względem procedury — patrz rozdział 5.).

Drugi wariant adresowania indeksowego to w istocie połączenie dwóch znanych nam już trybów. Jego składnia prezentuje się następująco:

```
mov( zmienna[ ebx + stała ], al );
mov( zmienna[ ebx - stała ], al );
```

Tutaj znów zastosowany został rejestr EBX, co nie oznacza, że w trybie tym nie można wykorzystywać pozostałych 32-bitowych rejestrów ogólnego przeznaczenia. Niniejsza wersja adresowania indeksowego jest szczególnie użyteczna w odwołaniach do składowych struktur przechowywanych w tablicy (patrz rozdział 4.).

W omawianym trybie adresowania adres efektywny operandu oblicza się przez dodanie bądź odjęcie stałej od adresu zmiennej, a następnie dodanie wyniku do zawartości rejestru. Warto pamiętać, że to kompilator, a nie procesor, oblicza sumę (bądź różnicę) stałej i adresu zmiennej. Powyższe instrukcje są bowiem na poziomie maszynowym implementowane za pośrednictwem pojedynczej instrukcji, dodającej pewną wartość do rejestru EBX. Z racji podstawiania przez kompilator w miejsce zmiennej jej stałego adresu, instrukcja:

```
mov( zmienna[ ebx + stała ], al );
```

redukowana jest do następującej instrukcji:

```
mov( stała1[ ebx + stała2 ], al );
```

Ze względu na sposób działania trybu adresowania powyższa instrukcja jest zaś równoważna następującej:

```
mov( [ ebx + (stała1 + stała2) ], al );
```

Obie stałe są sumowane na etapie kompilacji, co ostatecznie daje następującą instrukcję maszynową:

```
mov( [ ebx + suma_stałych ], al );
```

Oczywiście sprawy mają się identycznie również przy odejmowaniu. Różnica pomiędzy trybami adresowania z dodawaniem i odejmowaniem stałych może zostać bowiem łatwo zniwelowana — przy odejmowaniu stałą wystarczy obliczyć uzupełnienie do dwóch odejmowanej stałej, i tak otrzymaną wartość po prostu dodać do rejestru — dodawanie od odejmowania różni się więc tylko pojedynczą operacją negacji, również zresztą realizowaną na etapie kompilacji.

3.1.2.5. Adresowanie indeksowe skalowane

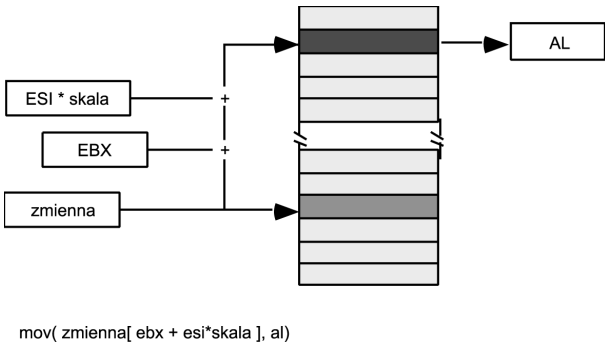
Tryb adresowania indeksowego skalowanego przypomina zaprezentowane tryby adresowania indeksowego. Różni się od nich zaledwie dwoma elementami: po pierwsze, w adresowaniu indeksowym skalowanym można uwikłać, oprócz wartości przemieszczenia, zawartość dwóch rejestrów;

po drugie, tryb adresowania indeksowego skalowanego pozwala na wymnożenie rejestru indeksowego przez współczynnik (skalę) o wartości 1, 2, 4 bądź 8. Składnię tego trybu określa się następująco:

```
zmienna[ rejestr-indeksowy32 * skala ]
zmienna[ rejestr-indeksowy32 * skala + przesunięcie ]
zmienna[ rejestr-indeksowy32 * skala - przesunięcie ]

[ rejestr-bazowy32 + rejestr-indeksowy32 * skala ]
[ rejestr-bazowy32 + rejestr-indeksowy32 * skala + przesunięcie ]
[ rejestr-bazowy32 + rejestr-indeksowy32 * skala - przesunięcie ]

zmienna[ rejestr-bazowy32 + rejestr-indeksowy32 * skala ]
zmienna[ rejestr-bazowy32 + rejestr-indeksowy32 * skala + przesunięcie ]
zmienna[ rejestr-bazowy32 + rejestr-indeksowy32 * skala - przesunięcie ]
```

W powyższych przykładach *rejestr-bazowy*$_{32}$ reprezentuje dowolny z 32-bitowych rejestrów ogólnego przeznaczenia, podobnie jak *rejestr-indeksowy*$_{32}$ (z puli dostępnych dla tego operandu rejestrów należy jednak wykluczyć rejestr ESP); *skala* jest stałą o wartości 1, 2, 4 bądź 8.

Skalowane adresowanie indeksowe różni się od prostego adresowania indeksowego przede wszystkim składową *rejestr-indeksowy*$_{32}$ ** skala*. W trybie tym adres efektywny obliczany jest przez dodanie wartości rejestru indeksowego pomnożonej przez współczynnik skalowania. Dopiero ta wartość wykorzystywana jest w roli indeksu. Sposób obliczania adresu efektywnego w tym trybie ilustrowany jest rysunkiem 3.6 (w roli rejestru bazowego występuje na nim rejestr EBX; rejestrem indeksowym jest ESI).



Rysunek 3.6. Adresowanie indeksowe skalowane

Jeśli dla sytuacji rozrysowanej na rysunku 3.6 przyjąć, że rejestr EBX zawiera wartość $100, rejestr ESI zawiera wartość $20, a zmienna została umieszczona w pamięci pod adresem $2000, wtedy instrukcja:

```
mov( zmienna[ ebx + esi*4 + 4], al);
```

spowoduje skopiowanie do rejestru AL pojedynczego bajta spod adresu $2184 ($2000+$100+$20*4+4).

Adresowanie indeksowe skalowane przydatne jest w odwołaniach do elementów tablicy, w której wszystkie elementy mają rozmiary dwóch, czterech bądź ośmiu bajtów. Wykorzystuje się go również w odwołaniach do elementów tablicy, kiedy dany jest wskaźnik do początkowego elementu tablicy.

3.1.2.6. Adresowanie w pigułce

Zapewne Czytelnik będzie powątpiewał w te słowa, ale właśnie poznał kilkaset trybów adresowania! Okazało się to nie takie trudne, prawda? Jeśli wciąż się to Czytelnikowi nie mieści w głowie, powinien wziąć pod uwagę, że, na przykład, tryb adresowania pośredniego przez rejestr nie jest pojedynczym trybem — obejmuje osiem trybów dla ośmiu różnych rejestrów. Wszystkie kilkaset trybów powstaje właśnie w wyniku kombinacji rejestrów, rozmiarów stałych i innych czynników. Tymczasem wystarczy zapoznać się z około dwudziestoma kilkoma postaciami odwołań do pamięci, aby posługiwać się całą dostępną gamą trybów adresowania. W praktyce zresztą w nawet najbardziej rozbudowanych wykorzystuje się i tak mniej niż połowę dostępnych trybów (wielu nie wykorzystuje się niemal wcale). Okazuje się więc, że opanowanie adresowania pamięci nie jest takie trudne.

3.2. Organizacja pamięci fazy wykonania

W systemach operacyjnych takich jak Mac OS X, FreeBSD, Linux czy Windows różne rodzaje danych programów umieszczane są w różnych sekcjach czy też obszarach pamięci. Co prawda przy uruchamianiu programu konsolidującego można ingerować w konfigurację pamięci programu, określając szereg opcji wywołania, ale domyślnie programy języka HLA w systemie Windows mają w pamięci reprezentację taką jak na rysunku 3.7 (to samo dotyczy zresztą systemów Linux, Mac OS X i FreeBSD; tam niektóre sekcje są jedynie inaczej rozmieszczone).

Rysunek 3.7. Typowe rozmieszczenie elementów programu HLA w pamięci

Najniższe adresy przestrzeni adresowej programu rezerwowane są przez system operacyjny. W ogólności aplikacje nie mogą odwoływać się do tego obszaru ani wykonywać w nim instrukcji. Obszar ten służy systemowi operacyjnemu między innymi do przechwytywania odwołań realizowanych za pośrednictwem wskaźników pustych (NULL). Jeśli instrukcja programu próbuje odwołać się do adresu zerowego (taki adres odpowiada wskaźnikowi pustemu), system operacyjny generuje błąd ochrony „general protection fault" sygnalizujący próbę odwołania do pamięci niedostępnej dla programu. Programiści często inicjalizują zmienne wskaźnikowe wartością NULL (zerem); wartość ta sygnalizuje potem, że wskaźnik nie wskazuje jeszcze na nic, a odwołanie za pośrednictwem takiego wskaźnika oznacza zazwyczaj błąd w programie polegający na nieprawidłowej inicjalizacji wskaźnika.

Pozostałych sześć obszarów mapy pamięci programu to obszary przypisane do poszczególnych rodzajów danych. Mamy tu obszar stosu, obszar sterty, obszar kodu, obszar danych niemodyfikowalnych (`readonly`), obszar zmiennych statycznych oraz obszar pamięci niezainicjalizowanej (`storage`). Każdy z tych obszarów służy do przechowywania określonych typów danych deklarowanych w programach języka HLA. Zostaną one szczegółowo omówione w kolejnych punktach.

3.2.1. Obszar kodu

Obszar kodu zawiera instrukcje maszynowe tworzące właściwy program HLA. Kompilator języka HLA tłumaczy instrukcje maszynowe kodu źródłowego do postaci sekwencji wartości jedno- bądź kilkubajtowych. Procesor interpretuje owe wartości jako instrukcje maszynowe (i ich operandy) i wykonuje je.

Kompilator HLA przez domniemanie podczas konsolidacji programu informuje system operacyjny, że program może z obszaru kodu czytać instrukcje i dane. Nie może natomiast zapisywać danych w obszarze kodu. W przypadku próby takiego zapisu system operacyjny wygeneruje błąd ochrony pamięci.

Instrukcje maszynowe to po prostu dane bajtowe. Teoretycznie można by napisać program, który zapisywałby dane w pamięci, a następnie przekazywał sterowanie do obszaru, w którym dane te zostały zapisane, co dałoby efekt samogenerowania programu w czasie jego działania. Możliwość ta skłania ku wizji programów **inteligentnych**, które w trakcie działania modyfikują swój kod, dostosowując się do postawionego zadania. Niestety, rzeczywistość skrzeczy i o tego typu efektach na razie nie ma mowy. Zasadniczo programy, które same się modyfikują, są bardzo trudne do diagnozowania i trudno jest śledzić ich wykonanie, ponieważ bez wiedzy programisty wciąż modyfikują kod. Większość współczesnych systemów operacyjnych wręcz utrudnia pisanie modyfikujących się programów, więc nie będziemy się więcej nimi zajmować w tej książce.

Kompilator języka HLA automatycznie umieszcza wszelkie dane związane z kodem maszynowym w obszarze kodu. Poza instrukcjami można w tym obszarze przechowywać również własne nieetykietowane dane, wykorzystując do tego następujące pseudoinstrukcje[4]:

- `byte`
- `word`
- `dword`

[4] Nie jest to lista pełna. Języka HLA pozwala w ogólności na osadzanie w obszarze kodu wartości poprzedzanych nazwą skalarnego typu danych. Owe typy danych zostaną omówione w rozdziale 4.

- uns8
- uns16
- uns32
- int8
- int16
- int32
- boolean
- char

Sposób zastosowania powyższych instrukcji ilustruje następująca składnia dla instrukcji `byte`:

```
byte lista-oddzielanych-przecinkami-stałych-jednobajtowych ;
```

A oto kilka konkretnych przykładów deklarowania danych nieetykietowanych w obszarze kodu:

```
boolean    true;
char       'A';
byte       0, 1, 2;
byte       "Ahoj!", 0;
word       0, 2;
int8       -5;
uns32      356789, 0;
```

Jeśli po pseudoinstrukcji pojawi się więcej niż jedna wartość stała, kompilator HLA umieszcza w strumieniu kodu każdą z nich po kolei. Stąd instrukcja `byte` powoduje wstawienie do tekstu kodu trzech danych bajtowych, o wartości odpowiednio: zero, jeden oraz dwa. Jeśli po instrukcji `byte` pojawia się literał łańcuchowy, HLA emituje w jego miejsce ciąg bajtów, których wartości odpowiadają kodom ASII kolejnych znaków literału. Stąd druga instrukcja `byte` powoduje wstawienie do tekstu kodu pięciu bajtów, których wartości odpowiadają znakom `'A'`, `'h'`, `'o'`, `'j'`, `'!'`, a za nimi pojedynczego bajta o wartości zero.

Należy jednak pamiętać, że procesor traktuje dane nieetykietowane osadzone w kodzie tak jak zwykłe instrukcje maszynowe, co wymusza podjęcie pewnych kroków zabezpieczających obszary danych przed wykonaniem. Na przykład, jeśli programista napisze:

```
mov( 0, ax );
byte 0, 1, 2, 3;
add( bx, cx );
```

to w ramach programu nastąpi — po wykonaniu pierwszej instrukcji `mov` — próba wykonania wartości bajtowych `0`, `1`, `2` oraz `3` jako instrukcji maszynowych. Takie osadzanie danych bajtowych pomiędzy instrukcjami kodu najczęściej powoduje błędne działanie programu. Dane takie, jeśli już są umieszczane w obszarze kodu, wymagają otoczenia ich instrukcjami skoku lub innymi, uniemożliwiającymi wykonanie danych jako instrukcji maszynowych.

3.2.2. Obszar zmiennych statycznych

Obszar sygnalizowany słowem kluczowym `static` to domyślnie obszar deklarowania zmiennych. Choć słowo kluczowe `static` może się pojawić jako część programu albo procedury, to należy pamiętać, że wszelkie deklarowane za tą klauzulą dane są przez kompilator i tak umieszczane nie w miejscu deklaracji, a w obszarze zmiennych statycznych.

W obszarze zmiennych statycznych można nie tylko deklarować zmienne, ale i osadzać dane nieetykietowane. Wykorzystuje się przy tym technikę identyczną jak w przypadku osadzania danych w obszarze kodu: wystarczy poprzedzić wartość pseudoinstrukcją `byte`, `word`, `dword`, `uns32` itp. Oto przykład:

```
static
    b:    byte := 0;
    byte  1, 2, 3;

    u:    uns32 := 1;
    uns32 5, 2, 10;

    c:    char;
    char  'a', 'b', 'c', 'd', 'e', 'f';

    bn:   boolean;
    boolean true;
```

Dane osadzane w obszarze zmiennych statycznych za pośrednictwem pseudoinstrukcji są zapisywane w tym obszarze zawsze za deklarowanymi w nim zmiennymi. Na przykład dane bajtowe o wartościach 1, 2 oraz 3 zostaną umieszczone w obszarze zmiennych statycznych dopiero za zmienną `b` inicjalizowaną zerem. Ponieważ z tak osadzanymi danymi nie są skojarzone żadne etykiety, nie można się do nich odwoływać w kodzie bezpośrednio jak do innych zmiennych, można natomiast wykorzystać adresowanie indeksowe (przykłady takich odwołań zostaną zaprezentowane w rozdziale 4.).

W powyższych przykładach zmienne `c` oraz `bn` nie są (przynajmniej w sposób jawny) inicjalizowane. Jednak nieokreślenie przez programistę wartości inicjalizującej nie oznacza, że pozostają one niezainicjalizowane — kompilator HLA domyślnie przyjmuje dla tych zmiennych inicjalizację zerem polegającą na wyzerowaniu wszystkich bitów zmiennych statycznych; zmienna `c` otrzyma więc początkową wartość NUL (zero odpowiada w zestawie ASCII znakowi pustemu). W szczególności należy pamiętać, że deklaracje zmiennych za słowem kluczowym `static` powodują rezerwowanie pamięci, nawet jeśli do zmiennych nie przypisano żadnej wartości.

3.2.3. Obszar niemodyfikowalny

Obszar danych niemodyfikowalnych przechowuje stałe, tablice i inne dane programu, które nie mogą w czasie jego wykonania podlegać żadnym modyfikacjom. Obiekty niemodyfikowalne deklaruje się w sekcji kodu sygnalizowanej słowem `readonly`. Sekcja ta ma charakter zbliżony do sekcji `static`; różnią się one trzema właściwościami:

- dane obszaru niemodyfikowalnego zapowiadane są w kodzie źródłowym słowem kluczowym `readonly`, a nie `static`;
- wszystkie stałe deklarowane w sekcji `readonly` są inicjalizowane;
- system nie pozwala na zapisywanie danych w obszarze niemodyfikowalnym w czasie działania programu.

Przykład:

```
readonly
    pi:         real32 := 3.14159;
    e:          real32 := 2.71;
    MaxU16      uns16 := 65_535;
    MaxI16      int16 := 32_767;
```

Wszystkie deklaracje w sekcji `readonly` muszą być uzupełnione o wyrażenie inicjalizacji — deklarowanych tu danych nie można przecież inicjalizować z poziomu już działającego programu[5]. Obiekty umieszczane w obszarze niemodyfikowalnym można traktować jako stałe, tyle że stałe te zajmują pamięć operacyjną i poza tym, że nie podlegają operacjom zapisu, zachowują się dokładnie tak jak zmienne obszaru zmiennych statycznych. Z tego względu obiektów obszaru niemodyfikowalnego nie można wykorzystywać wszędzie tam w programie, gdzie dozwolone jest zastosowanie stałej, czyli gdzie program oczekuje podania literału kodu źródłowego. W szczególności obiekty sekcji `readonly` (traktowane w programach jako stałe) nie nadają się do wykorzystania w roli stałych jako operandów instrukcji.

Podobnie jak w obszarze statycznym, w obszarze danych niemodyfikowalnych można osadzać dane nieetykietowane, poprzedzając je pseudoinstrukcjami `byte`, `word`, `dword` i tak dalej, jak poniżej:

```
readonly
    roArray:    byte := 0;
                byte 1, 2, 3, 4, 5;
    qwVal:      qword := 1;
                qword 0;
```

3.2.4. Obszar danych niezainicjalizowanych

W obszarze danych niemodyfikowalnych konieczne jest, z oczywistych względów, inicjalizowanie wszystkich deklarowanych tam obiektów. W obszarze zmiennych statycznych inicjalizacja jest nieobowiązkowa, ale dozwolona (a i tak wszystkie obiekty niezainicjalizowane jawnie są inicjalizowane zerem). W obszarze danych niezainicjalizowanych, którego deklaracje są w kodzie źródłowym programu zapowiadane słowem `storage`, wszystkie zmienne pozostają niezainicjalizowane. Zmienne obszaru niezainicjalizowanego deklaruje się następująco:

[5] Z jednym wyjątkiem opisanym w rozdziale 5.

```
storage
    UninitUns32:    uns32;
    i:              int32;
    character:      char;
    b:              byte;
```

W systemach Linux, FreeBSD, Mac OS X i Windows obszar zmiennych niezainicjalizowanych jest przy ładowaniu programu do pamięci wypełniany zerami. Nie należy jednak na tej niejawnej inicjalizacji polegać. Jeśli w programie niezbędny jest obiekt inicjalizowany wartością zerową, to należy zadeklarować go w obszarze zmiennych statycznych i określić stosowne wyrażenie inicjalizacji.

Zmienne deklarowane w obszarze danych niezainicjalizowanych zajmują w pliku wykonywalnym programu mniej miejsca niż dane pozostałych omówionych obszarów. Dla tamtych obszarów plik wykonywalny musi bowiem zawierać wartości początkowe obiektów. W obszarze danych niezainicjalizowanych jest to zbędne; faktyczna oszczędność miejsca w pliku wykonywalnym jest jednak własnością zależną od systemu operacyjnego i przyjętego w nim formatu obiektowego. Jako że obszar danych niezainicjalizowanych nie może zawierać wartości określanych statycznie (inicjalizowanych przy deklaracji), nie można w nim osadzać danych nieetykietowanych.

3.2.5. Atrybut @nostorage

Deklarowanie zmiennych w obszarze zmiennych statycznych, obszarze niemodyfikowalnym i obszarze danych niezainicjalizowanych z atrybutem `@nostorage` pozwala na opóźnienie przydziału pamięci dla zmiennej aż do momentu uruchomienia programu. Atrybut ten instruuje kompilator, aby ten przypisał do zmiennej adres, ale nie przydzielał do niej pamięci. Zmienna ta będzie dzielić pamięć z następnym obiektem, jaki pojawi się w danej sekcji deklaracji. Oto składnia opcji `@nostorage`:

```
nazwa-zmiennej: typ; @nostorage;
```

Nazwa typu jest tu uzupełniana o klauzulę `@nostorage;` — nie jest dozwolone określenie wartości początkowej zmiennej bez przydzielonej pamięci. Oto przykład zastosowania atrybutu `@nostorage` w sekcji `readonly`:

```
readonly
    abcd: dword; @nostorage;
          byte 'a', 'b', 'c', 'd';
```

W powyższym przykładzie zmienna `abcd` to podwójne słowo, którego najmniej znaczący bajt zawierać będzie wartość 97 (`'a'`). Bajt pierwszy zawierać będzie wartość 98 (`'b'`), bajt drugi — wartość 99 (`'c'`), a bajt najbardziej znaczący wartość 100 (`'d'`). Kompilator HLA nie

przydziela do zmiennej `abcd` podwójnego słowa pamięci, więc adres zmiennej skojarzony zostanie z czterema bajtami nieetykietowanymi, alokowanymi w pamięci bezpośrednio za zmienną `abcd`.

Atrybut `@nostorage` jest dozwolony jedynie w sekcjach `static`, `storage` oraz `readonly` (czyli w tak zwanych **obszarach deklaracji statycznych**). Język HLA nie przewiduje zastosowania tego atrybutu w sekcji `var`.

3.2.6. Sekcja deklaracji var

W języku HLA oprócz wymienionych wyżej dostępna jest też programiście sekcja deklaracji zapowiadana słowem `var`, w ramach której tworzone są **zmienne automatyczne**. Zmienne takie tworzone są w pamięci przy okazji rozpoczęcia wykonania pewnej jednostki programu (np. programu głównego albo procedury); pamięć zmiennych automatycznych jest zwalniana przy wychodzeniu z procedury czy programu. Naturalnie wszelkie zmienne automatyczne deklarowane w programie głównym charakteryzują się **czasem życia**[6] identycznym z czasem życia obiektów sekcji `static`, `storage` oraz `readonly` — dla zmiennych automatycznych programu głównego ich podstawowa cecha jest znoszona. W ogólności zastosowanie tych zmiennych ma więc sens wyłącznie w procedurach (patrz rozdział 5.). Mimo to język HLA dopuszcza deklarowanie zmiennych automatycznych również w ramach programu głównego.

Pamięć dla zmiennych deklarowanych w sekcji `var` przydzielana jest w czasie działania programu (w tzw. fazie wykonania), więc kompilator nie może samodzielnie ich inicjalizować. Z tego względu składnia obowiązująca w deklaracjach umieszczanych za słowem `var` zbliżona jest do tej znanej z deklaracji zmiennych obszaru niemodyfikowalnego. Jedyną istotną różnicą składniową pomiędzy tymi sekcjami jest zastosowanie słowa kluczowego `var` w miejsce `storage`[7]:

```
var
    vInt:       int32;
    vChar:      char;
```

HLA przydziela pamięć dla zmiennych deklarowanych w sekcji `var` w obszarze stosu programu. Kompilator nie może przy tym określić dokładnego adresu owych zmiennych. Zamiast tego zmienne są alokowane w ramach rekordu aktywacji skojarzonego z bieżącą jednostką programu. Omówienie rekordów aktywacji znajduje się w rozdziale 5.; na razie Czytelnik powinien zapamiętać, że w programach języka HLA wskaźnik na bieżący rekord aktywacji przechowywany jest w rejestrze EBP. Kiedy więc w programie następuje odwołanie do obiektu deklarowanego w sekcji `var`, nazwa zmiennej występująca w odwołaniu jest automatycznie zastępowana przez kompilator konstrukcją `[EBP ± przesunięcie]`. *Przesunięcie* jest przy tym przesunięciem obiektu w ramach rekordu aktywacji. Oznacza to, że w programach HLA nie można wykorzystywać w pełni trybu adresowania indeksowego skalowanego (gdzie adres efektywny określany jest wartością rejestru bazowego i iloczynem skali i wartości rejestru indek-

[6] Czas życie zmiennej to okres, jaki upływa od momentu przydzielenia dla niej pamięci do momentu zwolnienia tej pamięci.

[7] Jest też kilka różnic pomniejszych, które nie będą jednak omawiane w książce; zainteresowanych Czytelników odsyłam do dokumentacji języka HLA.

sowego), ponieważ rejestr EBP zarezerwowany jest w programach HLA dla rekordu aktywacji. I choć adresowania indeksowego skalowanego nie wykorzystuje się tak często, to już sam fakt, że nie da się go w pełni wykorzystać w obecności sekcji `var`, powinien stanowić wystarczający powód, aby unikać stosowania tej sekcji w programie głównym.

3.2.7. Rozmieszczenie sekcji deklaracji danych w programie HLA

Sekcje zapowiadane słowami `static`, `storage`, `readonly` oraz `var` mogą występować w programie HLA zero albo więcej razy, występując pomiędzy nagłówkiem `program`, a odpowiadającą programowi klauzulą `begin`. Pomiędzy tymi punktami sekcje deklaracji danych mogą występować w dowolnej kolejności, co ilustruje poniższy przykład:

```
program demoDeclarations;

static
    i_static:    int32;

var
    i_auto:      int32;

storage
    i_uninit:    int32;

readonly
    i_readonly:  int32 := 5;

static
    j:           uns32;

var
    k:           char;

readonly
    i2:          uns8 := 9;

storage
    c:           char;

storage
    d:           dword;

begin demoDeclarations;

    // kod programu

end demoDeclarations;
```

Powyższy przykład, oprócz możliwości dowolnego porządkowania poszczególnych sekcji deklaracji danych, demonstruje też możliwość występowania w programie danej sekcji wielokrotnie. W przypadku obecności w kodzie wielokrotnych deklaracji tej samej kategorii (tu mamy na przykład trzykrotnie określoną sekcję `storage`), poszczególne sekcje deklaracji są przez kompilator konsolidowane do postaci pojedynczej sekcji.

3.3. Przydział pamięci dla zmiennych w programach HLA

Czytelnik orientuje się już, że procesor nie odwołuje się do zmiennych przez ich nazwy, na przykład `I`, `Profits` czy `LineCnt`. Procesor może operować jedynie wartościami liczbowymi reprezentującymi adresy, tylko takie wartości nadają się bowiem do wysterowania szyny adresowej. Procesor nie rozróżnia więc nazw, a adresy, jak $1234_5678, $0400_1000 czy $8000_CC00. Z drugiej strony język HLA pozwala programiście na wykorzystywanie zamiast adresów zmiennych (co byłoby cokolwiek uciążliwe — adresy, w przeciwieństwie do nazw, są trudne do zapamiętania) ich nazw. Możliwość przydatna, ale o tyle niedobra, że zastosowanie nazw powoduje ukrycie faktycznego sposobu realizacji odwołań. W niniejszym podrozdziale przyjrzymy się więc sposobowi, w jaki kompilator HLA przypisuje adresy do zmiennych, tak aby Czytelnik — wciąż posługując się wygodnymi nazwami, a nie adresami — miał pełne wyobrażenie o tym, co kryje się za nazwą zmiennej.

Spójrzmy ponownie na rysunek 3.7. Widać na nim, że poszczególne obszary pamięci sąsiadują ze sobą. Z tego względu zmiana rozmiaru jednego z obszarów powoduje zmianę adresów bazowych wszystkich pozostałych obszarów pamięci. Na przykład, jeśli program zostanie uzupełniony kilkoma choćby instrukcjami maszynowymi, spowoduje to zwiększenie rozmiaru obszaru kodu, co z kolei może wymusić zmianę adresu bazowego obszaru zmiennych statycznych, co w efekcie prowadzi do zmiany adresów wszystkich zadeklarowanych w programie zmiennych statycznych. Rozróżnianie zmiennych na podstawie ich adresów jest i tak dla programisty zadaniem ponad siły; gdyby jeszcze musiał uwzględniać ich przemieszczenie w wyniku najdrobniejszych nawet modyfikacji kodu, to zapewne oszalałby z przepracowania. Na szczęście programistę w tym niewdzięcznym zadaniu wyręcza kompilator.

W języku HLA z każdą z trzech sekcji deklaracji statycznych (czyli sekcją `static`, `readonly` oraz `storage`) skojarzony jest **licznik lokacji**. Początkowo liczniki owe zawierają wartości zero; w momencie zadeklarowania zmiennej w jednej z sekcji deklaracji statycznych HLA kojarzy z tą zmienną bieżącą wartość licznika lokacji (otrzymując adres zmiennej), a sam licznik jest zwiększany o rozmiar deklarowanego obiektu. W ramach przykładu załóżmy, że poniższa sekcja jest jedyną sekcją `static` w programie:

```
static
    b:      byte;       // licznik lokacji = 0, rozmiar obiektu = 1;
    w:      word;       // licznik lokacji = 1, rozmiar obiektu = 2;
    d:      dword;      // licznik lokacji = 3, rozmiar obiektu = 4;
    q:      qword;      // licznik lokacji = 7, rozmiar obiektu = 8;
    l:      lword;      // licznik lokacji = 15, rozmiar obiektu = 16;
    // Bieżąca wartość licznika lokacji to 31.
```

Rzecz jasna w fazie wykonania programu adresy wszystkich tych zmiennych nie będą odpowiadały wartościom licznika lokacji. Wszystkie one zostaną po pierwsze zwiększone o adres bazowy obszaru zmiennych statycznych, a po drugie, jeśli w innym module konsolidowanym z programem (na przykład w module biblioteki standardowej HLA) występują kolejne sekcje deklaracji `static` albo kolejne takie sekcje występują w tym samym pliku źródłowym, konsolidator musi scalić obszary zmiennych statycznych. Z tego względu ostateczne adresy zmiennych statycznych mogą się nieco różnić od tych obliczonych przez proste przemieszczenie adresu bazowego obszaru statycznego o wartość licznika lokacji.

Nie zmienia to jednak zasadniczej właściwości mechanizmu przydziału pamięci do zmiennych statycznych, czyli tego, że są one przydzielane w ciągłym obszarze pamięci jedna za drugą. Wracając do przykładu: zmienna `b` będzie okupowała pamięć przylegającą bezpośrednio do pamięci przydzielonej do zmiennej `d`, która będzie sąsiadować w pamięci ze zmienną `w` i tak dalej. Co prawda w przypadku ogólnym nie należy zakładać takiego właśnie, sąsiadującego rozmieszczenia zmiennych w pamięci, ale niekiedy założenie takie jest bardzo wygodne.

Należy przy tym pamiętać, że zmienne deklarowane w sekcjach `readonly`, `static` oraz `storage` okupują zupełnie odrębne obszary pamięci. Stąd nie wolno zakładać, że deklarowane poniżej trzy obiekty będą ze sobą sąsiadować w pamięci programu:

```
static
    b    :byte;
readonly
    w    :word := $1234;
storage
    d    :dword;
```

W rzeczy samej kompilator HLA nie gwarantuje nawet, że zmienne tego samego obszaru pamięci, ale deklarowane w osobnych sekcjach deklaracji, będą ze sobą sąsiadować, nawet jeśli w kodzie źródłowym sekcji tych nie rozdziela żadna inna sekcja deklaracji. Stąd nie wolno zakładać, że w wyniku kompilacji poniższych deklaracji zmienne `b`, `d` i `w` będą w obszarze pamięci zmiennych statycznych rozmieszczone sąsiadująco w tej właśnie kolejności; nie wolno nawet zakładać, że w ogóle zostaną rozmieszczone sąsiadująco:

```
static
    b    :byte;
static
    w    :word := $1234;
static
    d    :dword;
```

Jeśli konstrukcja programu wymaga, aby owe zmienne okupowały sąsiadujące komórki pamięci, należy umieścić je we wspólnej sekcji deklaracji.

Deklaracje zmiennych automatycznych w sekcji `var` są obsługiwane nieco inaczej niż zmienne sekcji deklaracji statycznych. Sposób przydzielania pamięci do tych zmiennych zostanie omówiony w rozdziale 5.

3.4. Wyrównanie danych w programach HLA

Aby programy działały szybciej, należy obiekty danych odpowiednio rozmieszczać w pamięci; w szczególności istotne jest **wyrównanie** obiektów. Odpowiednie wyrównanie objawia się tym, że adres bazowy danego obiektu jest całkowitą wielokrotnością pewnego rozmiaru, zwykle rozmiaru tego obiektu, jeśli mieści się on w 16 bajtach. Dla obiektów większych od 16-bajtowych stosuje się wyrównanie ośmiobajtowe albo szesnastobajtowe. Dla obiektów mniejszych stosuje się wyrównanie do adresów będących wielokrotnościami takiej potęgi liczby dwa, która daje rozmiar większy od rozmiaru obiektu. Odwołania do danych niewyrównanych do odpowiednich adresów mogą wymagać dodatkowego czasu procesora, więc gwoli zapewnienia maksymalnej szybkości działania programu warto pamiętać o odpowiednim wyrównywaniu danych w pamięci.

Wyrównanie danych jest tracone, kiedy w sąsiadujących ze sobą komórkach pamięci alokowane są obiekty o różnych rozmiarach. Na przykład dla zmiennej o rozmiarze bajta przydzielona zostanie pamięć o rozmiarze jednego bajta. Następna zmienna, deklarowana w danej sekcji deklaracji, otrzyma adres równy adresowi owego bajta zwiększony o jeden. Jeśli zdarzyłoby się, że ów bajt został umieszczony w pamięci pod adresem parzystym, zmienna sąsiadująca z bajtem będzie siłą rzeczy mieć adres nieparzysty; jeśli będzie to słowo bądź podwójne słowo, adres taki nie będzie optymalny. Stąd konieczność znajomości sposobów wymuszania odpowiedniego wyrównania obiektów.

Niech w programie HLA określone zostaną następujące deklaracje statyczne:

```
static
    dw:    dword;
    b:     byte;
    w:     word;
    dw2:   dword;
    w2:    word;
    b2:    byte;
    dw3:   dword;
```

Pierwsza z deklaracji statycznych w programie (przy założeniu, że program ten będzie działał pod kontrolą systemu operacyjnego Windows, Mac OS X, FreeBSD, Linux lub innego systemu 32-bitowego) zostanie umieszczona pod adresem o parzystej wartości będącej przy tym wielokrotnością liczby 4096. Stąd pierwsza zmienna w sekcji deklaracji, niezależnie od jej typu, zostanie optymalnie wyrównana. Kolejne zmienne są jednak umieszczane pod adresami liczonymi jako suma adresu bazowego obszaru pamięci i rozmiarów wszystkich poprzednich zmiennych. Jeśli więc założyć, że deklarowane powyżej zmienne zostaną po kompilacji w obszarze pamięci rozpoczynającym się od adresu 4096, to poszczególne zmienne otrzymają następujące adresy:

```
                        // adres początkowy     rozmiar
    dw:     dword;      // 4096                 4
    b:      byte;       // 4100                 1
    w:      word;       // 4101                 2
    dw2:    dword;      // 4103                 4
    w2:     word;       // 4107                 2
```

```
    b2:     byte;       // 4109             1
    dw3:    dword;      // 4110             4
```

Z wyjątkiem zmiennej pierwszej (wyrównanej do adresu 4096) oraz zmiennej bajtowej `b2` (wyrównanie zmiennych bajtowych jest zawsze dobre) wszystkie zmienne są tu wyrównane nieoptymalnie. Zmienne `w`, `w2` oraz `dw2` rozmieszczone zostały pod nieparzystymi adresami; zmienna `dw3` została wyrównana do adresu parzystego, ale niebędącego, niestety, wielokrotnością czwórki.

Najprostszym sposobem zagwarantowania odpowiedniego wyrównania wszystkich zmiennych jest zadeklarowanie jako pierwszych wszystkich obiektów o rozmiarze podwójnego słowa, a za nimi wszystkich obiektów o rozmiarze słowa; obiekty jednobajtowe powinny być deklarowane na końcu, jak poniżej:

```
static
    dw:     dword;
    dw2:    dword;
    dw3:    dword;
    w:      word;
    w2:     word;
    b:      byte;
    b2:     byte;
```

Takie ułożenie deklaracji owocuje rozmieszczeniem zmiennych pod następującymi adresami (przyjmujemy, że adresem bazowym obszaru zmiennych statycznych jest 4096):

```
                    // adres początkowy     rozmiar
    dw:     dword;  // 4096                 4
    dw2:    dword;  // 4100                 4
    dw3:    dword;  // 4104                 4
    w:      word;   // 4108                 2
    w2:     word;   // 4110                 2
    b:      byte;   // 4112                 1
    b2:     byte;   // 4113                 1
```

Jak widać, wyrównanie poszczególnych zmiennych jest już zgodne z regułami sztuki.

Niestety, bardzo rzadko możliwe jest takie ułożenie zmiennych programu. Niemożność każdorazowego optymalnego ułożenia zmiennych wynika z szeregu przyczyn technicznych; w praktyce wystarczającą przyczyną jest brak logicznego powiązania deklaracji zmiennych, jeśli te są układane wyłącznie ze względu na rozmiar obiektu; tymczasem dla przejrzystości kodu źródłowego niektóre zmienne należy grupować niezależnie od ich rozmiarów.

Rozwiązaniem tego konfliktu interesów jest dyrektywa `align` języka HLA. Składnia dyrektywy prezentuje się następująco:

```
align( stała-całkowita );
```

Stała całkowita określona w argumencie dyrektywy może być jedną z następujących wartości: 1, 2, 4, 8 lub 16. Jeśli w sekcji deklaracji za słowem `static` kompilator HLA napotka dyrektywę `align`, następna deklarowana w sekcji zmienna zostanie wyrównana do adresu będącego całkowitą wielokrotnością argumentu dyrektywy. Analizowany przez nas przykład można z wykorzystaniem dyrektywy `align` przepisać następująco:

```
static
    align( 4 );
    dw:    dword;
    b:     byte;
    align( 2 );
    w:     word;
    align( 4 );
    dw2:   dword;
    w2:    word;
    b2:    byte;
    align( 4 );
    dw3:   dword;
```

Jak działa dyrektywa `align`? To całkiem proste. Jeśli kompilator wykryje, że bieżący adres (czyli bieżąca wartość licznika lokacji) nie jest całkowitą wielokrotnością wartości określonej argumentem dyrektywy, wprowadza do obszaru pamięci szereg bajtów danych nieetykietowanych (bajtów wyrównania), uzupełniając nimi poprzednią deklarację, tak aby bieżąca wartość licznika lokacji osiągnęła pożądaną wartość. Program staje się przez to nieco większy (o dosłownie kilka bajtów), ale w zamian dostęp do danych programu jest nieco szybszy; jeśli faktycznie zwiększenie rozmiaru programu miałoby się ograniczyć do kilku dodatkowych bajtów, wymiana ta byłaby bardzo atrakcyjna.

Warto przyjąć regułę, że w celem maksymalizowania szybkości odwołań do obiektów danych należy je wyrównywać do adresów będących całkowitymi wielokrotnościami ich rozmiaru. Słowa powinny więc być wyrównywane do parzystych adresów pamięci (`align(2);`), podwójne słowa do adresów podzielnych bez reszty przez cztery (`align(4);`), słowa poczwórne należy wyrównywać do adresów podzielnych przez osiem i tak dalej. Jeśli rozmiar obiektu nie jest równy potędze dwójki, należy wyrównać go do adresów podzielnych przez potęgę dwójki najbliższą jego rozmiarowi, lecz od niego większą. Wyjątkiem są obiekty typu `real80` oraz `tbyte`, które należy wyrównywać do adresów podzielnych przez osiem.

Wyrównywanie danych nie zawsze jest konieczne. Architektura pamięci podręcznej współczesnych procesorów z rodziny 80x86 pozwala bowiem na efektywną (w większości przypadków) obsługę również danych niewyrównanych. Dyrektywy wyrównania powinny więc być stosowane wyłącznie wobec tych danych, w przypadku których szybkość dostępu ma zasadnicze znaczenie dla wydajności programu. W przypadku takich zmiennych ewentualny koszt optymalizacji dostępu w postaci kilku dodatkowych bajtów programu będzie z pewnością do przyjęcia.

3.5. Wyrażenia adresowe

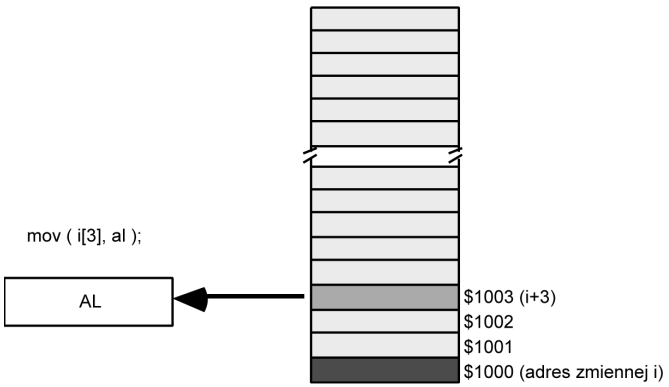
Prezentowane w poprzednich podrozdziałach tryby adresowania ilustrowane były kilkoma postaciami odwołań, w tym:

```
zmienna[ rejestr32 ];
zmienna[ rejestr32 + przesunięcie ];
zmienna[ rejestr32-nie-ESP * skala ];
zmienna[ rejestr32 + rejestr32-nie-ESP * skala ];
zmienna[ rejestr32-nie-ESP * skala + przesunięcie ];
zmienna[ rejestr32 + rejestr32-nie-ESP * skala + przesunięcie ];
```

Istnieje jeszcze jedna forma odwołania, niewprowadzająca nowego trybu adresowania, a będąca jedynie rozszerzeniem trybu adresowania bezpośredniego:

```
zmienna[ przesunięcie ]
```

W tej ostatniej postaci odwołania adres efektywny obliczany jest przez dodanie stałego *przesunięcia* określonego wewnątrz nawiasów prostokątnych do adresu *zmiennej*. Na przykład instrukcja `mov(adres[3], al)` powoduje załadowanie rejestru AL wartością znajdującą się w pamięci pod adresem odległym o trzy bajty od adresu zmiennej `adres` (patrz rysunek 3.8).



Rysunek 3.8. Wyrażenie adresowe w odwołaniu do danej umieszczonej za zmienną

Wartość przesunięcia musi być wyrażona jako stała, czyli literał liczbowy. Jeśli, na przykład, zmienna `i` jest zmienną typu `int32`, wtedy wyrażenie `zmienna[i]` nie jest dozwolonym wyrażeniem adresowym. Aby odwoływać się do danych za pośrednictwem indeksu dynamicznego modyfikowanego w trakcie działania programu, należy skorzystać z trybu adresowania indeksowego, ewentualnie indeksowego skalowanego.

Kolejnym ważnym spostrzeżeniem jest to, że przesunięcie w wyrażeniu *adres*[*przesunięcie*] jest adresem bajtowym. Mimo że składnia wyrażenia adresowego przypomina tę znaną z języków C, C++ i Pascal, to tutaj przesunięcie nie stanowi indeksu w sensie licznika elementów tablicy — chyba że *adres* jest tablicą bajtów.

W niniejszej książce *wyrażeniem adresowym* będzie dowolny z trybów adresowania procesora 80x86, który obejmuje przemieszczenie (na przykład zawiera nazwę zmiennej) albo przesunięcie. Za poprawne wyrażenia adresowe będą także uważane następujące odwołania:

```
[ rejestr32 + przesunięcie ]
[ rejestr32 + rejestr-nieESP32 * skala + przesunięcie ]
```

Natomiast poniższe wyrażenia *nie* będą uznawane za poprawne wyrażenie adresowe, jako że nie angażują ani przemieszczenia, ani przesunięcia:

```
[ rejestr32 ]
[ rejestr32 + rejestr-nieESP32 * skala ]
```

Wyrażenia adresowe są o tyle szczególne, że instrukcje zawierające wyrażenia adresowe zawsze kodują stałą przemieszczenia jako składową instrukcji maszynowej. Oznacza to, że struktura instrukcji maszynowej przewiduje pewną liczbę bitów (zwykle 8 bądź 32) dla wartości stałej określającej przemieszczenie. Stała ta obliczana jest jako suma określonego w kodzie przemieszczenia (czyli np. adresu zmiennej względem adresu bazowego) oraz ewentualnego przesunięcia. Kompilator automatycznie sumuje te wartości (albo je odejmuje, kiedy w wyrażeniu adresowym w miejsce plusa występuje minus).

Jak dotychczas przesunięcie we wszystkich przykładach adresowania reprezentowane było pojedynczą stałą liczbową — literałem liczbowym. Tymczasem w języku HLA wszędzie tam, gdzie powinno zostać określone przesunięcie, dopuszczalne jest stosowanie **wyrażeń stałowartościowych**. Wyrażenie stałowartościowe składa się z jednego lub kilku składowych będących stałymi łączonych za pośrednictwem operatorów takich jak operatory dodawania, odejmowania, dzielenia i mnożenia, operator modulo i szeregu innych operatorów. Weźmy następujący przykład:

```
mov( x[ 2*4 + 1], al );
```

Powyższa instrukcja spowoduje skopiowanie pojedynczego bajta spod adresu `X+9` do rejestru AL.

Wartość wyrażenia adresowego jest zawsze obliczana w czasie kompilacji, nigdy w fazie wykonania programu. Kiedy kompilator HLA napotka w kodzie źródłowym wyrażenie podobne do prezentowanego wyżej, oblicza wartość 2*4+1 i dodaje otrzymany rezultat do bazowego adresu zmiennej `X` w pamięci. Całość przemieszczenia, na którą składa się przemieszczenie zmiennej `X` względem adresu bazowego oraz przesunięcie 2*4+1, kodowane jest następnie w instrukcji maszynowej, co znakomicie zmniejsza nakłady czasowe potrzebne do ustalenia adresu efektywnego w fazie wykonania. Obliczanie wyrażeń adresowych w czasie kompilacji

nakłada na wszystkie składowe wyrażenia stałowartościowego wymóg określoności wartości podczas kompilacji — kompilator nie jest w stanie przewidzieć wartości zmiennej w czasie wykonania programu, stąd w wyrażeniach adresowych nie mogą być wykorzystywane zmienne.

Wyrażenia adresowe przydają się w odwołaniach do danych znajdujących się w pamięci poza zasięgiem zmiennej, na przykład do zmiennych nieetykietowanych wprowadzanych do kodu bądź obszarów danych pseudoinstrukcjami `byte`, `word`, `dword` i im podobnymi. Rozważmy, na przykład, program z listingu 3.1.

Listing 3.1. Przykład zastosowania wyrażenia adresowego

```
program adrsExpressions;
#include( "stdlib.hhf" );
static
    i:  int8; @nostorage;
        byte 0, 1, 2, 3;

begin adrsExpressions;

    stdout.put
    (
        "i[0]=", i[0], nl,
        "i[1]=", i[1], nl,
        "i[2]=", i[2], nl,
        "i[3]=", i[3], nl
    );

end adrsExpressions;
```

Uruchomienie programu z listingu 3.1 spowoduje wyprowadzenie na wyjście wartości 0, 1, 2 oraz 3, zupełnie tak, jakby były one kolejnymi elementami tablicy. Jest to możliwe dzięki temu, że pod adresem zmiennej `i` umieszczony został nieetykietowany bajt o wartości zero. Zmienna `i` została bowiem zadeklarowana z atrybutem `@nostorage`, co oznacza, że jej adres ma się pokrywać z adresem następnego zadeklarowanego w danej sekcji obiektu. W naszym przykładzie obiektem tym jest akurat nieetykietowany bajt danych o wartości 0. Dalej, wyrażenie adresowe `i[1]` jest przez kompilator HLA realizowane jako instrukcja pobrania bajta znajdującego się pod adresem odległym o jeden bajt od adresu zmiennej `i`. Tam z kolei znajduje się nieetykietowany bajt o wartości 1. Analogicznie dla wyrażeń `i[2]` oraz `i[3]` program wyprowadza wartości dwóch pozostałych nieetykietowanych bajtów.

3.6. Koercja typów

Choć język HLA nie szczyci się szczególnie ścisłą kontrolą typów, kompilator tego języka potrafi wymusić na programiście przynajmniej zastosowanie w danej instrukcji operandów o odpowiednich rozmiarach. Weźmy na przykład następujący, celowo niepoprawny program:

```
program hasErrors;
static
    i8:    int8;
    i16:   int16;
    i32:   int32;

begin hasErros;

    mov( i8, eax );
    mov( i16, al );
    mov( i32, ax );

end hasErrors;
```

Kompilacja programu zakończy się zgłoszeniem błędu kompilacji wszystkich trzech konstytuujących program instrukcji `mov`. Przyczyną błędów jest naturalnie niezgodność rozmiarów operandów. W pierwszej instrukcji następuje próba załadowania 32-bitowego rejestru EAX wartością 8-bitowej zmiennej; w drugiej instrukcji programista próbuje załadować 8-bitowy rejestr AL wartością 16-bitową, a w trzeciej rejestr 16-bitowy AX ma być załadowany wartością 32-bitową. Tymczasem instrukcja `mov` nakłada na operandy wymóg identyczności rozmiarów.

Niewątpliwie tego rodzaju kontrola typów jest zaletą języka HLA[8], niekiedy jednak zaczyna programiście przeszkadzać. Na przykład w poniższym fragmencie kodu:

```
static
    byte_values:    byte; @nostorage;
                    byte 0, 1;

…

   mov( byte_values, ax );
```

W tym przykładzie programista faktycznie zamierza załadować 16-bitowy rejestr 16-bitowym słowem, którego adres jest identyczny z adresem zmiennej 8-bitowej `byte_values`. Rejestr AL miałby być załadowany wartością 0, a rejestr AH wartością 1 (zauważmy, że w mniej znaczącym bajcie pamięci zmiennej przechowywane jest 0, a w bardziej znaczącym bajcie pamięci — 1). Niestety, kompilator HLA zablokuje tego rodzaju próbę, podejrzewając błąd niezgodności rozmiarów operandów (w końcu zmienna `byte_values` to zmienna 8-bitowa, a rejestr AX ma 16 bitów). Programista może obejść przeszkodę, ładując rejestr dwoma instrukcjami maszynowymi: jedną ładująca rejestr AL bajtem spod adresu zmiennej `byte_values` i drugą — ładującą rejestr AH wartością następnego bajta, `byte_values[1]`. Niestety, taka dekompozycja instrukcji powoduje zmniejszenie wydajności programu (a najprawdopodobniej właśnie troska o tę wydajność zmusiła programistę do umieszczenia w kodzie tak karkołomnej jak zaprezentowana kon-

[8] W końcu niezgodność rozmiarów operandów jest najczęściej efektem nieuwagi programisty.

strukcji). Byłoby więc pożądane, aby dało się poinstruować kompilator o zamiarze dotrzymania wymogu zgodności rozmiarów i że adres zmiennej `byte_values` ma być interpretowany jako adres nie bajta, a słowa. Możliwość tę daje koercja typów.

Koercja typów[9] to proces, w ramach którego kompilator HLA informowany jest o tym, że dany obiekt będzie traktowany jako obiekt typu określonego wprost w kodzie, niekoniecznie zgodnego z typem podanym w deklaracji. Składnia koercji typu zmiennej wygląda następująco:

```
(type nowa-nazwa-typu wyrażenie-adresowe)
```

Nowa nazwa typu określa typ docelowy koercji, który ma zostać skojarzony z adresem pamięci wyznaczanym *wyrażeniem adresowym*. Operator koercji może być wykorzystywany wszędzie tam, gdzie dozwolone jest określenie adresu w pamięci. Znając koercję typów, można poprawić poprzedni przykład, tak aby dał się skompilować bez błędów:

```
mov( (type word byte_values), ax);
```

Powyższa instrukcja nakazuje załadowanie rejestru AX wartością słowa rozpoczynającego się pod adresem `byte_values`. Jeśli założyć, że pod adresem tym nadal znajdują się wartości bajtów nieetykietowanych prezentowanych w przykładzie, rejestr AL zostanie załadowany wartością zero, a AH — wartością jeden.

Koercja typów jest koniecznością, kiedy w roli operandu instrukcji bezpośrednio modyfikującej pamięć (a więc instrukcji `neg`, `shl`, `not` i im podobnym) ma wystąpić zmienna anonimowa. Rozważmy następujący przykład:

```
not( [ebx] );
```

Instrukcji takiej nie da się skompilować, ponieważ nie sposób na jej podstawie określić rozmiaru operandu docelowego. Kompilator nie ma więc wystarczających informacji do skonstruowania kodu instrukcji maszynowej — nie wie, czy program ma dokonać inwersji bitów pojedynczego bajta wskazywanego zawartością rejestru EBX, czy może całego znajdującego się pod tym adresem słowa albo i podwójnego słowa. Aby określić rozmiar operandu niezbędny do zakodowania instrukcji maszynowej, należy wykonać koercję typu odwołania do zmiennej anonimowej, jak w poniższych instrukcjach:

```
not( (type byte [ebx]) );
not( (type dword [ebx]) );
```

[9] W niektórych innych językach identyczny proces nosi nazwę **rzutowania**.

Ostrzeżenie

Nie wolno wykorzystywać koercji typów na chybił trafił, bez pełnej świadomości skutków, jakie koercja przyniesie. Początkujący programiści języka asemblerowego często korzystają z koercji typów jako środka uciszania kompilatora, kiedy ten zwraca nie do końca dla nich zrozumiałe komunikaty o błędach niezgodności typów.

Przykładem niepoprawnej koercji może być następująca instrukcja (zakładamy, że `byteVar` to zmienna jednobajtowa):

```
mov( eax, (type dword byteVar) );
```

Gdyby nie koercja typów, kompilator odmówiłby kompilacji kodu ze względu na niedopasowanie rozmiarów operandów instrukcji `mov`. Następuje tu bowiem próba skopiowania 32-bitowej zawartości rejestru do zmiennej jednobajtowej. Jeśli koercja została przez początkującego programistę zastosowana wyłącznie celem uciszenia kompilatora, to niewątpliwie cel ten zostanie osiągnięty — kompilator nie będzie już ostrzegał o niedopasowaniu typów. Program może być jednak mimo bezbłędnej kompilacji niepoprawny. Operator koercji nie eliminuje bowiem źródła potencjalnego problemu, jakim jest próba umieszczenia wartości 32-bitowej w zmiennej 8-bitowej. Próba taka musi zakończyć się po prostu umieszczeniem czterech bajtów w pamięci, *poczynając od adresu* zmiennej `byteVar`. Tak więc trzy bajty kopiowane z rejestru nadpiszą wartości trzech bajtów sąsiadujących w pamięci ze zmienną `byteVar`. Błędy tego rodzaju często objawiają się nieoczekiwanymi, tajemniczymi modyfikacjami zmiennych programu[10], albo, gorzej, prowokują błąd ochrony pamięci. Ten ostatni może wystąpić, jeśli na przykład jeden z trójki bajtów sąsiadujących ze zmienną `byteVar` będzie już należeć do obszaru pamięci niemodyfikowalnej. Warto więc w odniesieniu do stosowania operatora koercji przyjąć następującą regułę: „jeśli nie wiadomo dokładnie, jaki wpływ na działanie programu ma zastosowanie operatora koercji, stosować go po prostu nie należy".

Nie wolno zapominać, że operator koercji typów nie realizuje żadnej translacji czy konwersji danych przechowywanych w obiekcie, do którego odwołanie zostało poddane działaniu operatora. Operator ten ma wpływ jedynie na działanie kompilatora, instruując go co do sposobu interpretowania rozmiaru operandu. W szczególności, koercja wartości jednobajtowej ze znakiem do rozmiaru trzydziestu dwóch bitów nie spowoduje automatycznego rozszerzenia znakiem ani też koercja do typu zmiennoprzecinkowego nie spowoduje konwersji obiektu do postaci zmiennoprzecinkowej.

3.7. Koercja typu rejestru

Za pośrednictwem operatora koercji można też wykonać rzutowanie rejestru na określony typ. Domyślnie bowiem rejestry 8-bitowe są w języku HLA obiektami typu `byte`, rejestry 16-bitowe mają przypisany typ `word`, a rejestry 32-bitowe to obiekty typu `dword`. Przy użyciu operatora

[10] Jeśli bezpośrednio za zmienną `byteVar` w pamięci programu znajduje się inna zmienna, jej wartość zostanie w wyniku wykonania instrukcji `mov` na pewno nadpisana, niezależnie od tego, czy jest to efekt przewidziany i pożądany przez programistę.

koercji można interpretację typu rejestru zmieniać pod warunkiem, że **typ docelowy będzie identycznego rozmiaru co rozmiar rejestru**. Koercja typu rejestru nie ma większego zastosowania, jednak czasem trudno się bez niej obejść. Jedną z sytuacji, w których się ona przydaje, jest konstruowanie wyrażeń logicznych w wysokopoziomowych instrukcjach języka HLA (jak `if` czy `while`) i przekazywanie zawartości rejestrów do procedur wejścia-wyjścia, gdzie koercja umożliwia odpowiednią interpretację tej zawartości.

W wyrażeniach logicznych języka HLA obiekty typu `byte`, `word` i `dword` interpretowane są zawsze jako wartości bez znaku. Stąd bez koercji typu rejestru poniższa instrukcja `if` miałaby zawsze wartość `false` (trudno bowiem, aby wartość bez znaku była mniejsza od zera):

```
if( eax < 0 ) then

    stdout.put( "Wartosc rejestru EAX jest ujemna!", nl );

endif;
```

Słabość tę można wyeliminować, stosując w wyrażeniu logicznym instrukcji `if` koercję typu rejestru:

```
if( (type int32 eax) < 0 ) then

    stdout.put( "Wartosc rejestru EAX jest ujemna!", nl );

endif;
```

Na podobnej zasadzie wartości typu `byte`, `word` oraz `dword` są przez procedurę `stdout.put` interpretowane jako liczby szesnastkowe. Jeśli więc zachodzi potrzeba wyświetlenia zawartości rejestru, to jego przekazanie wprost do procedury wyjścia `stdout.put` spowoduje wyprowadzenie jego wartości w zapisie szesnastkowym. Jeśli programista chce wymusić inną interpretację zawartości rejestru, musi skorzystać z koercji typu rejestru:

```
stdout.put( "AL interpretowany jako znak = '", (type char AL), "'", nl );
```

Identyczną rolę pełni koercja typu rejestru w wywołaniach procedur wejściowych jak `stdin.get`. Ta procedura bowiem, jeśli argument określa operand docelowy jako operand typu `byte`, `word` bądź `dword`, interpretuje wprowadzane dane jako wartości szesnastkowe; niekiedy zachodzi więc konieczność dokonania koercji typu rejestru.

3.8. Pamięć obszaru stosu oraz instrukcje push i pop

Wcześniej w rozdziale wspomniano, że wszystkie zmienne deklarowane w sekcji `var` lądują w obszarze pamięci zwanym obszarem stosu. Jednak obszar stosu nie służy wyłącznie do przechowywania obiektów automatycznych — pamięć stosu wykorzystywana jest do wielu różnych celów i na wiele sposobów. W niniejszym podrozdziale poznamy stos procesora, zaprezentowane zostaną też dwie z instrukcji służących do manipulowania danymi na stosie: `push` oraz `pop`.

Obszar stosu to ten fragment pamięci programu, w której procesor przechowuje swój **stos**. Stos jest dynamiczną strukturą danych, która zwiększa lub zmniejsza swój rozmiar w zależności od bieżących potrzeb programu. Stos zawiera też ważne dla poprawnego działania programu informacje, w tym zmienne lokalne (automatyczne), informacje o wywołaniach procedur i dane tymczasowe.

W procesorach 80x86 pamięć stosu kontrolowana jest za pośrednictwem rejestru ESP zwanego też wskaźnikiem stosu. Kiedy program zaczyna działanie, system operacyjny inicjalizuje wskaźnik stosu adresem ostatniej komórki pamięci w obszarze pamięci stosu (największym możliwym adresem w obszarze pamięci stosu). Zapis danych do tego obszaru odbywa się jako „odkładanie danych na stos" (ang. *pushing*) i „zdejmowanie danych ze stosu" (ang. *popping*).

3.8.1. Podstawowa postać instrukcji push

Oto składnia instrukcji `push` procesora 80x86:

```
push( rejestr16 );
push( rejestr32 );
push( pamięć16 );
push( pamięć32 );
pushw( stała );
pushd( stała );
```

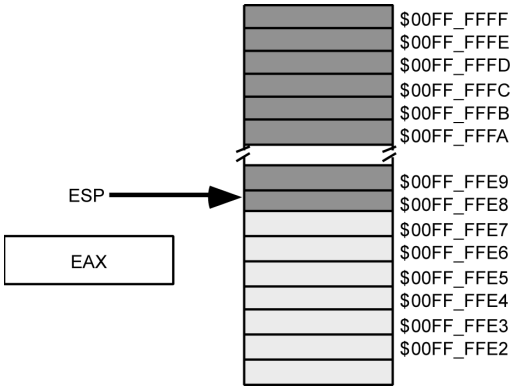
Zaprezentowanych wyżej sześć wersji instrukcji `push` pozwala na odkładanie na stos obiektów typu `word` i `dword`, czyli zawartości rejestrów 16- i 32-bitowych, jak również wartości przechowywanych w postaci słów i podwójnych słów w pamięci. W szczególności zaś nie jest możliwe odkładanie na stos wartości typu `byte`.

Działanie instrukcji `push` można rozpisać następującym pseudokodem:

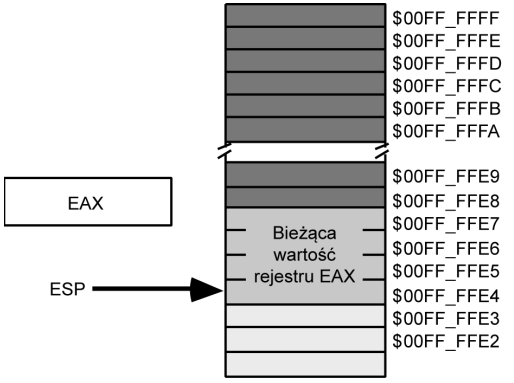
```
ESP := ESP - rozmiar-operandu (2 bądź 4)
[ESP] := wartość-operandu
```

Operandami instrukcji `pushw` i `pushd` są zawsze stałe o rozmiarze odpowiednio: słowa bądź podwójnego słowa.

Jeśli na przykład rejestr ESP zawiera wartość $00FF_FFE8, to wykonanie instrukcji `push( eax );` spowoduje ustawienie rejestru ESP na wartość $00FF_FFE4 i skopiowanie bieżącej wartości rejestru EAX pod adres $00FF_FFE4; proces ten ilustrują rysunki 3.9 oraz 3.10.



Rysunek 3.9. Stan pamięci stosu przed wykonaniem instrukcji push



Rysunek 3.10. Stan pamięci stosu po wykonaniu instrukcji push

Wykonanie instrukcji `push( eax );` nie wpływa przy tym w żaden sposób na zawartość rejestru EAX.

Choć procesory z rodziny 80x86 implementują 16-bitowe wersje instrukcji manipulujących pamięcią stosu, to owe wersje mają zastosowanie głównie w środowiskach 16-bitowych, jak system DOS. Tymczasem gwoli maksymalnej wydajności warto, aby wartość wskaźnika stosu była zawsze całkowitą wielokrotnością liczby cztery; program może zresztą w systemie takim jak Windows czy Linux zostać awaryjnie zatrzymany, kiedy system wykryje, że wskaźnik stosu zawiera wartość niepodzielną bez reszty przez cztery. Jedynym uzasadnieniem dla odkładania na stosie danych innych niż 32-bitowe jest więc konstruowanie za pośrednictwem stosu wartości o rozmiarze podwójnego słowa składanej z dwóch słów umieszczonych na stosie jedno po drugim.

3.8.2. Podstawowa postać instrukcji pop

Do zdejmowania danych umieszczonych wcześniej na stosie służy instrukcja pop. W swej podstawowej wersji instrukcja ta przyjmuje jedną z czterech postaci:

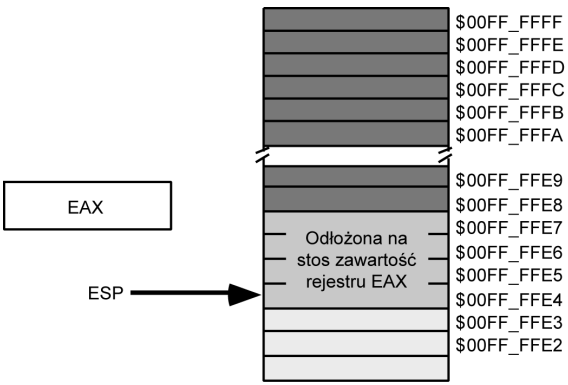
```
pop( rejestr16 );
pop( rejestr32 );
pop( pamięć16 );
pop( pamięć32 );
```

Podobnie jak to ma miejsce w przypadku instrukcji push, instrukcja pop obsługuje jedynie operandy 16- i 32-bitowe; ze stosu nie można zdejmować wartości ośmiobitowych. Podobnie jednak jak przy instrukcji push, zdejmowania ze stosu wartości 16-bitowych powinno się unikać, chyba że operacja taka stanowi jedną z dwóch operacji zdejmowania ze stosu realizowanych pod rząd) — zdjęcie ze stosu danej 16-bitowej powoduje, że wartość rejestru wskaźnika stosu nie dzieli się bez reszty przez cztery, co nie jest pożądane. W przypadku instrukcji pop dochodzi jeszcze jedno ograniczenie: nie da się pobrać wartości ze stosu, określając w instrukcji operand w postaci stałej — jest to zresztą ograniczenie o tyle naturalne, że operand instrukcji push jest operandem źródłowym i jako taki może być stałą; trudno natomiast, aby stałą był operand docelowy, a taki występuje w instrukcji pop.

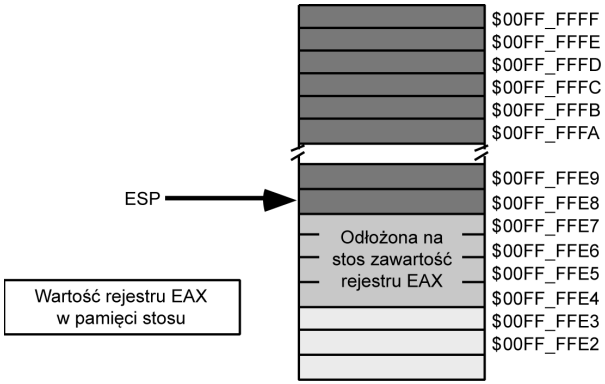
Sposób działania instrukcji pop można opisać następującym pseudokodem:

```
operand := [ESP]
ESP := ESP + rozmiar-operandu (2 bądź 4)
```

Operacja zdejmowania ze stosu jest, jak widać, operacją dokładnie odwrotną do operacji odkładania danych na stosie. Instrukcja pop realizuje bowiem kopiowanie wartości spod adresu wskazywanego wskaźnikiem stosu jeszcze przed jego zwiększeniem. Obraz pamięci stosu przed i po wykonaniu instrukcji pop ilustrują rysunki 3.11 oraz 3.12.



Rysunek 3.11. Stan pamięci stosu przed wykonaniem instrukcji pop



Rysunek 3.12. Stan pamięci stosu po wykonaniu instrukcji pop

Należy podkreślić, że wartość zdjęta ze stosu wciąż znajduje się w obszarze pamięci stosu. Zdejmowanie danej ze stosu nie oznacza zamazywania pamięci stosu; efekt „zniknięcia" danej ze stosu osiągany jest przez przesunięcie wskaźnika stosu tak, aby wskazywał wartość sąsiadującą z wartością zdjętą (o wyższym adresie). Nigdy jednak nie należy próbować odwoływać się do danej zdjętej już ze stosu — następne odłożenie czegokolwiek na stos powoduje już bowiem nadpisanie obszaru, w którym owa dana się wcześniej znajdowała. A ponieważ nie wolno zakładać, że stos manipulowany jest wyłącznie kodem programu (stos jest wykorzystywany tak przez system operacyjny, jak i kod wywołujący procedury), nie powinno się inicjować odwołań do danych, które zostały już zdjęte ze stosu i co do których istnieje jedynie podejrzenie (bo przecież nie pewność), że jeszcze są obecne w pamięci stosu.

3.8.3. Zachowywanie wartości rejestrów za pomocą instrukcji push i pop

Najważniejszym chyba zastosowaniem instrukcji `pop` i `push` jest zachowywanie zawartości rejestrów w obliczu potrzeby ich czasowego innego niż dotychczasowe wykorzystania. W architekturze 80x86 gospodarka rejestrami jest o tyle problematyczna, że procesor ten zawiera wyjątkowo małą liczbę rejestrów ogólnego przeznaczenia. Rejestry znakomicie nadają się do przechowywania wartości tymczasowych (np. wyników pośrednich etapów obliczeń), ale są też potrzebne do realizacji różnych trybów adresowania. Z tego względu programista często staje w obliczu niedostatku rejestrów, zwłaszcza kiedy kod realizuje złożone obliczenia. Ratunkiem mogą być wtedy instrukcje `push` oraz `pop`.

Rozważmy następujący zarys programu:

```
// sekwencja instrukcji wykorzystujących rejestr EAX

// sekwencja instrukcji, na potrzeby których należy zwolnić rejestr EAX

// kontynuacja sekwencji instrukcji wykorzystujących rejestr EAX
```

Do zaimplementowania takiego planu znakomicie nadają się instrukcje `push` oraz `pop`. Za ich pomocą można najpierw zachować, a następnie przywrócić zawartość rejestru EAX; w międzyczasie można zaś zrealizować kod wymagający zwolnienia tego rejestru:

```
// sekwencja instrukcji wykorzystujących rejestr EAX
push( eax );
// sekwencja instrukcji, na potrzeby których należy zwolnić rejestr EAX
pop( eax );
// kontynuacja sekwencji instrukcji wykorzystujących rejestr EAX
```

Umiejętnie osadzając w kodzie instrukcje `push` i `pop`, można zachować na stosie wynik obliczeń realizowanych za pośrednictwem rejestru EAX na czas wykonania kodu, który ten rejestr wykorzystuje w innym celu. Po zakończeniu owego fragmentu kodu można przywrócić poprzednio zachowaną wartość EAX i kontynuować przerwane obliczenia.

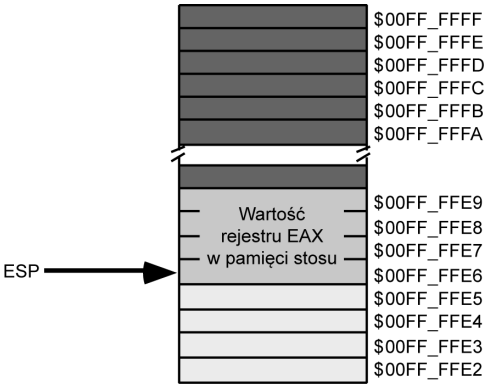
3.9. Stos jako kolejka LIFO

Nie jest powiedziane, że stos należy wykorzystywać do odkładania wyłącznie pojedynczych danych. Stos jest bowiem po prostu implementacją kolejki LIFO (ang. *last in, first out*, czyli ostatnie na wejściu — pierwsze na wyjściu). Obsługa takiej kolejki dla całych sekwencji danych wymaga jednak uważnego kontrolowania kolejności odkładania i zdejmowania danych. Rozważmy na przykład sytuację, gdy na czas realizacji pewnych instrukcji należy zachować zawartość rejestrów EAX I EBX. Początkujący programista mógłby zrealizować zabezpieczenie na stosie wartości rejestrów tak:

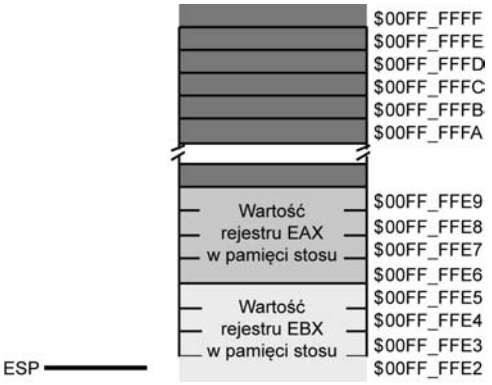
```
push( eax );
push( ebx );
// Sekwencja kodu wymagająca zwolnienia rejestrów EAX i EBX.
pop( eax );
pop( ebx );
```

Niestety, powyższy kod będzie działał niepoprawnie! Błąd zawarty w tym kodzie ilustrują rysunki 3.13 do 3.16. Problem można opisać następująco: na stos najpierw odkładany jest rejestr EAX, a po nim EBX. Wskaźnik stosu wskazuje w efekcie adres pamięci stosu, pod którym składowana jest zawartość rejestru EBX. Kiedy w ramach przywracania poprzednich wartości rejestrów wykonywana jest instrukcja `pop( eax );`, do rejestru EAX trafia wartość, która pierwotnie znajdowała się w rejestrze EBX! Z kolei następna instrukcja, `pop( ebx );`, ładuje do rejestru EBX wartość, która powinna tak naprawdę trafić do rejestru EAX! Do zamiany wartości rejestrów doszło w wyniku zastosowania niepoprawnej sekwencji zdejmowania ze stosu — dane powinny być z niego zdejmowane w kolejności odwrotnej, niż zostały nań odłożone.

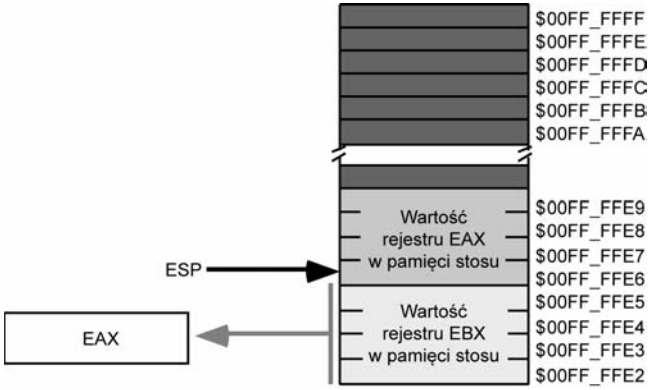
Stos, jako struktura odpowiadająca kolejce LIFO, ma tę właściwość, że to, co trafia na stos jako pierwsze, powinno z niego zostać zdjęte w ostatniej kolejności. Dla uproszczenia warto zapamiętać następującą regułę:



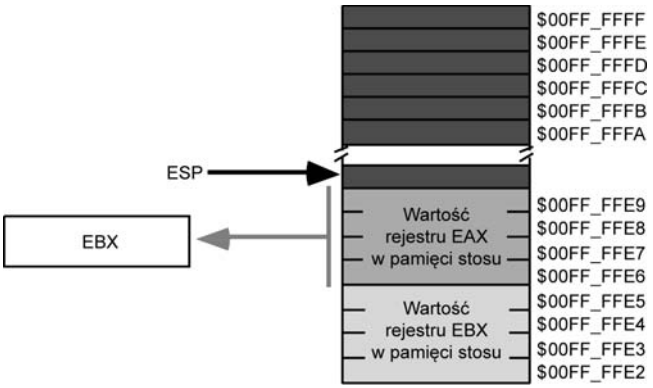
Rysunek 3.13. Obraz pamięci stosu po odłożeniu na niego zawartości rejestru EAX



Rysunek 3.14. Obraz pamięci stosu po odłożeniu na niego zawartości rejestru EBX



Rysunek 3.15. Obraz pamięci stosu po zdjęciu z niego danej do rejestru EAX



Rysunek 3.16. Obraz pamięci stosu po zdjęciu z niego danej do rejestru EBX

Dane ze stosu należy zdejmować w kolejności odwrotnej do ich odkładania.
Problematyczny kod można poprawić następująco:

```
push( eax );
push( ebx );
// Sekwencja kodu wymagająca zwolnienia rejestrów EAX i EBX.
pop( ebx );
pop( eax );
```

Jest jeszcze jedna ważna reguła, której stosowanie pozwala unikać błędów wynikających z nieodpowiedniego manipulowania stosem:

Zdejmować ze stosu należy dokładnie tyle bajtów, ile się wcześniej nań odłożyło.
Chodzi o to, aby liczba i „ciężar" danych zdejmowanych ze stosu była dokładnie równa liczbie i „ciężarowi" danych na ten stos wcześniej odkładanych. Jeśli liczba instrukcji `pop` jest zbyt mała, na stosie pozostaną osierocone dane, co może w dalszym przebiegu programu doprowadzić do błędów wykonania. Jeszcze gorsza jest sytuacja, kiedy liczba instrukcji `pop` jest zbyt duża — to niemal zawsze prowadzi do załamania programu.

Szczególną wagę należy przykładać do zrównoważenia operacji odkładania i zdejmowania realizowanych w pętli. Częstym błędem jest odkładanie danych na stos wewnątrz pętli i ich tylko jednokrotne zdejmowanie po wyjściu z pętli (bądź odwrotnie) — prowadzi to oczywiście do naruszenia spójności danych na stosie. Należy więc pamiętać, że znaczenie ma nie liczba instrukcji w kodzie źródłowym programu, ale to, ile razy zostaną one wykonane w fazie wykonania. A w fazie tej liczba instrukcji `pop` musi odpowiadać liczbie (i kolejności) instrukcji `push`.

3.9.1. Pozostałe wersje instrukcji obsługi stosu

Procesory z rodziny 80x86 udostępniają programiście szereg dodatkowych wersji instrukcji manipulujących stosem. Wśród nich są następujące instrukcje maszynowe:

- `pusha`
- `pushad`
- `pushf`
- `pushfd`
- `popa`
- `popad`
- `popf`
- `popfd`

Wykonanie instrukcji `pusha` powoduje odłożenie na stos wszystkich 16-bitowych rejestrów ogólnego przeznaczenia. Instrukcja ta wykorzystywana jest głównie w 16-bitowych systemach operacyjnych takich jak MS-DOS. W ogólności więc potrzeba jej wykorzystania jest raczej rzadka. Rejestry są na stosie odkładane w następującej kolejności:

```
ax
cx
dx
bx
sp
bp
si
di
```

Instrukcja `pushad` powoduje odłożenie na stosie wszystkich 32-bitowych rejestrów ogólnego przeznaczenia. Ich zawartość ląduje na stosie w następującej kolejności:

```
eax
ecx
edx
ebx
esp
ebp
esi
edi
```

Nie sposób nie zauważyć, że wykonanie instrukcji `pusha` (`pushad`) powoduje zmodyfikowanie wartości wskaźnika stosu SP (ESP). Powstaje więc pytanie, po co w ogóle ów rejestr jest odkładany na stosie? Prawdopodobnie odpowiedź na to pytanie wynika z tego, że ze względów technicznych łatwiejsze jest zapewne odłożenie na stos wszystkich rejestrów naraz, bez czynienia wyjątku dla nieaktualnego w chwili odkładania na stos rejestru SP (ESP).

Instrukcje `popa` i `popad` to odpowiadające instrukcjom `pusha` i `pushad` instrukcje zdejmowania ze stosu całych grup wartości do rejestrów ogólnego przeznaczenia. Naturalnie instrukcje te zachowują właściwy porządek zdejmowania ze stosu zawartości poszczególnych rejestrów, odwrotny do kolejności ich odkładania.

Mimo że stosowanie zbiorczych instrukcji `pusha` (`pushad`) oraz `popa` (`popad`) jest bardzo wygodne, ich realizacja przebiega nieco dłużej, niż gdyby w ich miejsce zastosować stosowną sekwencję instrukcji `push` i `pop`. Nie jest to specjalnym problemem, jako że rzadko zachodzi

potrzeba odkładania na stos zawartości większej liczby rejestrów[11]. Jeśli więc w programie chodzi o maksymalną wydajność przetwarzania, należy każdorazowo przeanalizować sensowność wykonania instrukcji zbiorczego odkładania rejestrów na stos.

Instrukcje `pushf`, `pushfd`, `popf` i `popfd` powodują, odpowiednio: umieszczenie i zdjęcie ze stosu rejestru znaczników EFLAGS. Instrukcje te pozwalają na zachowanie słowa stanu programu na czas wykonania pewnej sekwencji instrukcji. Niestety, trudniej jest zachować wartości pojedynczych znaczników. Instrukcją `pushf(d)` i `popf(d)` można zachowywać na stosie jedynie wszystkie znaczniki naraz; bardziej bolesne jest jednak to, że rejestr znaczników również przywrócić można tylko w całości.

Przy zachowywaniu i przywracaniu wartości rejestru znaczników należy korzystać z 32-bitowej wersji instrukcji, czyli `pushfd` i `popfd`. Co prawda dodatkowe 16 bitów odłożonych na stosie nie jest w typowych aplikacjach nijak wykorzystywane, ale przynajmniej zachowuje się w ten sposób wyrównanie stosu, którego wskaźnik powinien być zawsze liczbą podzielną bez reszty przez cztery.

3.9.2. Usuwanie danych ze stosu bez ich zdejmowania

Okazjonalnie może pojawić się kwestia następująca: na stos odłożone zostały pewne dane, które jednak już dalej w programie nie będą wykorzystywane. Można co prawda zdjąć te dane ze stosu instrukcją `pop`, umieszczając je w nieużywanym akurat rejestrze, ale można to również zrobić metodą prostszą, mianowicie ingerując w wartość rejestru wskaźnika stosu.

Niech ilustracją tego zagadnienia będzie następujący kod:

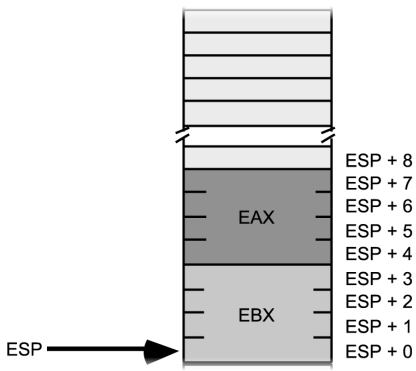
```
push( eax );
push( ebx );

// Kod kończący obliczenia na rejestrach EAX i EBX.

if( Calculation_was_performed ) then

    // Hm... Jest już wynik i odłożone na stos wartości nie będą w takim razie potrzebne.
    // Co z nimi zrobić?

else

    // Konieczne dalsze obliczenia; przywróć zawartość rejestrów.

    pop( ebx );
    pop( eax );

endif;
```

[11] Na przykład bardzo rzadko zachodzi potrzeba odłożenia na stos (albo zdjęcia ze stosu) zawartości rejestru ESP w ramach sekwencji instrukcji `pushad-popad`.

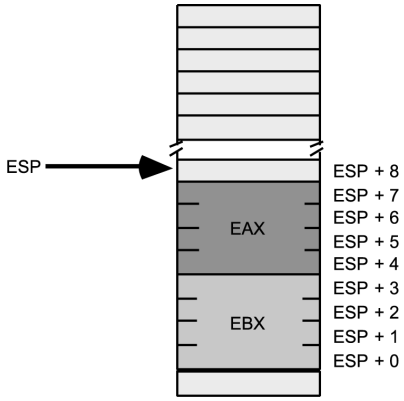
W ramach klauzuli `then` instrukcji `if` należałoby usunąć ze stosu poprzednie wartości rejestrów EAX i EBX, ale bez wpływania na zawartość pozostałych rejestrów czy zmiennych. Jak to zrobić?

Można wykorzystać fakt, że rejestr ESP przechowuje wprost wartość wskaźnika stosu, czyli szczytowego elementu stosu; wystarczy więc dostosować tę wartość tak, aby wskaźnik stosu wskazywał na niższy, kolejny element stosu. W prezentowanym przykładzie ze szczytu stosu należało usunąć dwie wartości o rozmiarze podwójnego słowa. Efekt usunięcia ich ze stosu można osiągnąć, dodając do wskaźnika stosu liczbę osiem (takie „usuwanie" danych ze stosu ilustrują rysunki 3.17 oraz 3.18):

```
push( eax );
push( ebx );

// Kod kończący obliczenia na rejestrach EAX i EBX.

if( Calculation_was_performed ) then

    add( 8, ESP );          // Usuń niepotrzebne dane ze stosu.

else

    // Konieczne dalsze obliczenia; przywróć zawartość rejestrów.

    pop( ebx );
    pop( eax );

endif;
```



Rysunek 3.17. Usuwanie danych ze stosu; obraz pamięci stosu przed wykonaniem instrukcji add(8, ESP)



Rysunek 3.18. Usuwanie danych ze stosu; obraz pamięci stosu po wykonaniu instrukcji add(8, ESP)

W ten sposób można „zdjąć" dane ze stosu bez umieszczania ich w jakimkolwiek operandzie docelowym. zwiększenie wskaźnika stosu jest też szybsze niż wykonanie sekwencji sztucznych instrukcji `pop`, ponieważ w pojedynczej instrukcji `add` możemy zwiększyć wskaźnik stosu o większą liczbę podwójnych słów.

Ostrzeżenie

Przy „usuwaniu" danych ze stosu nie wolno zapominać o zachowaniu wyrównania stosu. Rejestr wskaźnika stosu ESP należy każdorazowo modyfikować o liczbę będącą całkowitą wielokrotnością liczby cztery.

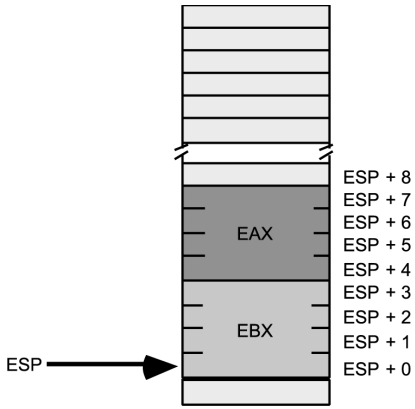
3.10. Odwoływanie się do danych na stosie bez ich zdejmowania

Czasami zdarza się, że do danych odłożonych na stosie trzeba się odwołać, ale ich ze stosu nie zdejmować — może na przykład chodzić o czasowe przywrócenie odłożonej wartości i być może nawet jej modyfikowanie, z zachowaniem rezerwy pierwotnej wartości na stosie, celem ich późniejszego zdjęcia. Otóż można to zrobić, korzystając z adresowania postaci [$rejestr_{32}$ + *przesunięcie*].

Rozważmy obraz pamięci stosu (rysunek 3.19) po wykonaniu dwóch poniższych instrukcji:

```
push( eax );
push( ebx );
```

Jeśli zachodzi teraz potrzeba odwołania się do poprzedniej zawartości rejestru EBX bez zdejmowania go ze stosu, można by spróbować małego oszustwa: zdjąć daną ze stosu do rejestru EBX i natychmiast ją z powrotem odłożyć na stos. Gorzej, kiedy będzie trzeba odwołać się



Rysunek 3.19. Pamięć stosu po odłożeniu nań zawartości rejestrów EAX i EBX

do poprzedniej wartości rejestru EAX albo innej wartości, odłożonej na stos jeszcze wcześniej. Zdejmowanie ze stosu wszystkich zasłaniających ją danych (a następnie ich umieszczenie z powrotem na stosie) byłoby w najlepszym razie problematyczne, a w najgorszym — niemożliwe do wykonania. Na rysunku 3.19 widać jednakże, że każda z wartości odłożonych na stos znajduje się w pamięci obszaru stosu pod adresem odległym od bieżącej wartości wskaźnika stosu o określoną wartość przesunięcia, dlatego można skorzystać z odwołania postaci `[ ESP + przesunięcie ]` i odwołać się do pożądanej wartości bezpośrednio w pamięci stosu. W powyższym przykładzie można, na przykład, przywrócić poprzednią zawartość rejestru EAX, wykonując instrukcję:

```
mov( [esp + 4], eax );
```

Wykonanie tej instrukcji spowoduje skopiowanie do rejestru EAX wartości znajdującej się pod adresem ESP+4. Adres ten określa daną znajdującą się bezpośrednio pod szczytem stosu. Technikę tę można jednak z powodzeniem stosować również do danych znajdujących się głębiej.

Ostrzeżenie

Nie wolno zapominać, że przesunięcia konkretnych elementów w pamięci stosu zmieniają się w wyniku wykonania każdej instrukcji `push` i `pop`. Pominięcie tego faktu może doprowadzić do stworzenia trudnego do modyfikowania kodu źródłowego. Opieranie się na założeniu, że przesunięcie jest stałe pomiędzy punktem w programie, w którym dane zostały na stos odłożone, a punktem, w którym programista zdecydował się do nich odwołać, może uniemożliwiać bądź utrudniać uzupełnianie kodu, zwłaszcza jeśli uzupełnienie będzie zawierać instrukcje manipulujące stosem.

W poprzednim punkcie pokazany został sposób usuwania danych ze stosu polegający na modyfikowaniu wartości rejestru wskaźnika stosu. Prezentowany przy tej okazji kod można by jeszcze ulepszyć, zapisując go następująco:

```
push( eax );
push( ebx );

// Kod kończący obliczenia na rejestrach EAX i EBX.

if( Calculation_was_performed ) then

    // Nadpisz wartości przechowywane na stosie nowymi wartościami EAX i EBX, tak aby
    // można było bezpiecznie zdjąć je ze stosu, nie ryzykując utraty bieżącej zawartości rejestrów.
mov( eax, [esp + 4] );
    mov( ebx, [esp] );

endif;

    pop( eax );
    pop( ebx );
```

W powyższej sekwencji kodu wynik pewnych obliczeń został zapisany w miejscu poprzednich wartości rejestrów EAX i EBX. Kiedy później wykonane zostaną instrukcje zdjęcia ze stosu, rejestry EAX i EBX pozostaną niezmienione — wciąż będą zawierać obliczone i uznane w instrukcji `if` za ostateczne — wartości.

3.11. Dynamiczny przydział pamięci — obszar pamięci sterty

Potrzeby pamięciowe co prostszych programów mogą być skutecznie zaspokajane deklaracjami zmiennych statycznych i automatycznych. Jednak bardziej zaawansowane zastosowania wymagają możliwości przydziału i zwalniania pamięci w sposób dynamiczny, kiedy decyzje o potrzebie przydziału podejmowane są nie na etapie pisania kodu, a w fazie wykonania programu. W języku C do dynamicznego przydzielania pamięci służy funkcja `malloc`, a do jej zwalniania — funkcja `free`. Język C++ przewiduje wykorzystanie do tych samych celów operatorów `new` oraz `delete`. W Pascalu mamy funkcje `new` i `dispose`. Analogiczne mechanizmy dostępne są też w innych językach programowania wysokiego poziomu. Wszystkie one dzielą następujące cechy: pozwalają programiście na określenie rozmiaru przydzielanej pamięci, zwracają **wskaźnik** do początku obszaru przydzielonej pamięci i umożliwiają zwrócenie pamięci do systemu, kiedy nie będzie już potrzebna. Jak można się domyślać, również w języku HLA — a konkretnie w ramach biblioteki standardowej HLA — dostępne są procedury realizujące przydział i zwalnianie pamięci.

Przydział pamięci jest w języku HLA realizowany za pośrednictwem procedury bibliotecznej `mem.alloc`, jej zwalnianie odbywa się zaś za pośrednictwem procedury `mem.free`. Procedura `mem.alloc` wywoływana jest następująco:

```
mem.alloc( liczba-bajtów );
```

Jedyny argument wywołania procedury `mem.alloc` to wartość o rozmiarze podwójnego słowa, określająca liczbę bajtów, jaka ma zostać przydzielona do programu. Stosownej wielkości pamięć przydzielana jest w obszarze pamięci sterty. Wywołanie funkcji powoduje przydzielenie wolnego bloku tej pamięci i oznaczenie tego bloku jako „zajętego", co pozwala na ochronę pamięci przed wielokrotnym przydziałem. Po oznaczeniu bloku pamięci jako „zajętego" procedura zwraca za pośrednictwem rejestru EAX wskaźnik na pierwszy bajt przydzielonego obszaru.

W przypadku większości obiektów liczba bajtów niezbędna do prawidłowego zachowania obiektu w pamięci jest programiście znana. Na przykład chcąc dynamicznie przydzielić pamięć dla zmiennej typu `uns32`, można skorzystać z następującego wywołania:

```
mem.alloc( 4 );
```

Jak widać, w wywołaniu procedury `mem.alloc` można skutecznie umieszczać literały liczbowe, ale w ogólnym przypadku lepiej jest skorzystać z dostępnej w HLA **funkcji czasu kompilacji**[12] o nazwie `@size`. Wywołanie tej funkcji jest zastępowane obliczonym przez kompilator rozmiarem danych. Składnia wywołania `@size` jest następująca:

```
@size( nazwa-zmiennej-bądź-typu )
```

Wywołanie funkcji `@size` zastępowane jest stałą liczbą całkowitą równą rozmiarowi parametru wywołania, określonemu w bajtach. Poprzednie wywołanie procedury przydziału `mem.alloc` można więc zapisać następująco:

```
mem.alloc( @size( uns32 ) );
```

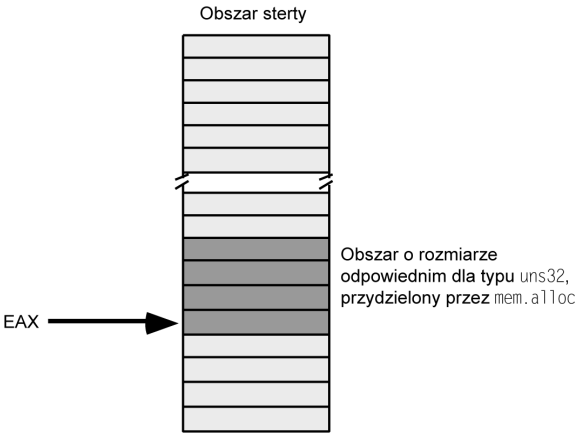
Powyższe wywołanie spowoduje przydzielenie w obszarze pamięci sterty obszaru odpowiedniego do przechowywania obiektu zadanego typu. Co prawda nie należy się spodziewać, aby rozmiar typu danych `uns32` został kiedykolwiek zmieniony, jednak w przypadku innych typów danych (zwłaszcza tych definiowanych przez użytkownika) stałość rozmiaru nie jest już taka pewna, więc warto wyrobić sobie nawyk stosowania w miejsce literałów liczbowych wywołania funkcji `@size`.

Po zakończeniu wykonywania kodu procedury `mem.alloc` w rejestrze EAX powinien znajdować się wskaźnik na przydzielony obszar pamięci — patrz rysunek 3.20.

Aby odwołać się do pamięci przydzielonej w wyniku wywołania procedury `mem.alloc`, należy skorzystać z adresowania pośredniego przez rejestr. Oto przykład przypisania wartości 1234 do zmiennej typu `uns32` przydzielonej w pamięci sterty:

```
mem.alloc( @size( uns32 ) );
mov( 1234, (type uns32 [eax] ) );
```

[12] Funkcja czasu kompilacji to taka, której wartość jest obliczana nie w czasie wykonania programu, a już na etapie kompilacji.



Rysunek 3.20. Wywołanie procedury mem.alloc zwraca w rejestrze EAX wskaźnik na przydzielony obszar

Warto zwrócić uwagę na zastosowanie w powyższym kodzie operatora koercji typu rejestru. Otóż jest on tu niezbędny, ponieważ zmienne anonimowe nie mają żadnego typu, więc kompilator nie mógłby stwierdzić zgodności rozmiarów operandów — w końcu wartość 1234 da się też zapisać zarówno w zmiennej o rozmiarze słowa, jak i w zmiennej o rozmiarze podwójnego słowa. Zastosowanie operatora koercji typu pozwala na rozstrzygnięcie niejednoznaczności.

Przydział pamięci za pośrednictwem procedury `mem.alloc` nie zawsze jest skuteczny. Jeśli na przykład w obszarze pamięci sterty nie istnieje odpowiednio duży ciągły obszar wolnej pamięci, wywołanie `mem.alloc` sprowokuje wyjątek `ex.MemoryAllocationFailure`. Jeśli wywołanie nie zostanie osadzone w bloku kodu chronionego instrukcji `try`, błąd przydziału pamięci spowoduje awaryjne zatrzymanie wykonania programu. Jako że większość programów nie przydziela jakichś gigantycznych obszarów pamięci, wyjątek ten zgłaszany jest stosunkowo rzadko. Niemniej jednak nie powinno się zakładać, że przydział pamięci będzie zawsze skuteczny.

Kiedy operacje na obiektach danych przydzielonych w pamięci sterty zostaną zakończone, można zajmowaną przez te obiekty pamięć zwolnić do systemu operacyjnego, czyli oznaczyć jako „wolną". Służy do tego procedura `mem.free`. Procedura ta przyjmuje pojedynczy argument, którym musi być adres zwrócony podczas odpowiedniego wywołania przydzielającego pamięć. Dodatkowo nie może to być adres pamięci raz już zwolnionej. Sposób wykorzystywania pary instrukcji `mem.alloc` i `mem.free` ilustruje następujący przykład:

```
mem.alloc( @size( uns32 ) );

    // Manipulowanie obiektami w pamięci o adresie zwróconym przez rejestr EAX.
    // Uwaga: ten kod nie może modyfikować zawartości EAX.

mem.free( eax );
```

Niniejszy kod ilustruje bardzo ważną zależność — aby skutecznie zwolnić pamięć przydzieloną wywołaniem `mem.alloc`, należy zachować wskaźnik zwracany przez to wywołanie. Jeśli

rejestr EAX jest na czas wykorzystywania pamięci dynamicznej potrzebny do innych celów, można ów wskaźnik zachować na stosie albo po prostu skopiować go do zmiennej w pamięci.

Zwolnione obszary pamięci są dostępne dla następnych operacji przydziału, realizowanych za pośrednictwem procedury `mem.alloc`. Możliwość przydzielania pamięci do obiektów i jej zwalniania w razie potrzeby znakomicie zwiększa efektywność wykorzystania pamięci. Zwalniając niepotrzebną już pamięć dynamiczną, można ją udostępnić dla innych celów, zmniejszając zajętość pamięci w porównaniu z sytuacją, w której pamięć dla takich tymczasowych danych przydzielana była statycznie.

Z wykorzystaniem wskaźników wiąże się kilka problemów. Często powodują one u niedoświadczonych programistów następujące błędy nieprawidłowej obsługi pamięci dynamicznej:

- Odwoływanie się do zwolnionych wcześniej obszarów pamięci. Po zwróceniu pamięci do systemu (wywołaniem procedury `mem.free`) nie można już odwoływać się do tej pamięci. Odwołania takie mogą doprowadzić do sprowokowania błędu ochrony pamięci albo — co gorsze, bo trudniejsze do wykrycia — nadpisanie innych danych przydzielanych później dynamicznie w zwolnionym obszarze pamięci.
- Dwukrotne wywoływanie procedury `mem.free` w odniesieniu do tego samego obszaru pamięci. Powtórne wywołanie procedury `mem.free` może doprowadzić do nieumyślnego zwolnienia innego obszaru pamięci albo wręcz naruszyć spójność tablic podsystemu zarządzania pamięcią.

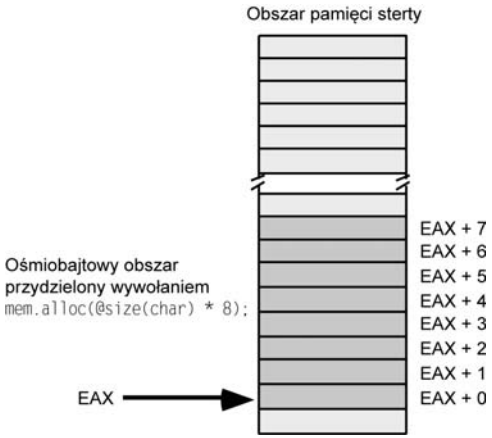
W rozdziale 4. omówionych zostanie jeszcze kilka innych problemów związanych z obsługą pamięci dynamicznej.

Wszystkie prezentowane dotychczas przykłady pokazywały przydział i zwalnianie pamięci dla pojedynczych zmiennych określonego typu — 32-bitowej zmiennej bez znaku. Tymczasem naturalnie przydział może dotyczyć dowolnego typu danych, określonego w wywołaniu procedury `mem.alloc` nazwą typu albo po prostu liczbą potrzebnych bajtów. Można w ten sposób przydzielać pamięć dla całych sekwencji obiektów. Na przykład poniższe wywołanie realizuje przydział pamięci dla ośmiu znaków:

```
mem.alloc( @size( char ) * 8 );
```

W powyższej instrukcji uwagę zwraca zastosowanie wyrażenia stałowartościowego w celu obliczenia liczby bajtów wymaganych do przechowywania ośmioznakowej sekwencji. Jako że funkcja `@size(char)` zwraca zawsze rozmiar (w bajtach) pojedynczego znaku, to przydział pamięci dla ośmiu znaków należy zasygnalizować osiem razy większym argumentem wywołania; wyrażenie stałowartościowe, nawet najbardziej złożone, jest obliczane przez kompilator i nie powoduje wstawienia do kodu maszynowego żadnych dodatkowych instrukcji.

Wywołanie procedury `mem.alloc` dla liczby bajtów większej niż jeden powoduje zawsze przydział ciągłego obszaru pamięci o zadanym rozmiarze. Stąd dla prezentowanego wcześniej wywołania w pamięci sterty zarezerwowana zostanie ośmiobajtowa porcja pamięci, jak zostało to pokazane na rysunku 3.21.



Rysunek 3.21. Przydział pamięci dla sekwencji znaków

Do kolejnych znaków sekwencji można się odwoływać, określając ich przesunięcie względem adresu bazowego sekwencji zwracanego przez rejestr EAX. Na przykładu, aby zapisać w trzecim znaku sekwencji wartość przechowywaną w rejestrze CH, należy skorzystać z instrukcji `mov( CH, [eax + 2] );`. Można też, na przykład, skorzystać z adresowania `[eax + ebx]` i wtedy przesunięcie odwołania określać zawartością rejestru EBX, odpowiednio manipulując jego wartością. Na przykład poniższy kod ustawia wszystkie znaki 128-znakowej sekwencji na wartość NUL (wartość #0):

```
mem.alloc( 128 );
for( mov( 0, ebx ); ebx < 128; add( 1, ebx ) ) do

    mov( 0, ( type byte [eax + ebx] ) );

endfor;
```

W rozdziale 4., gdzie będą omawiane złożone struktury danych (w tym tablice elementów), zaprezentowane zostaną jeszcze inne sposoby odwoływania się do obszarów pamięci zawierających sekwencje obiektów.

Należy jeszcze podkreślić, że wywołanie procedury `mem.alloc` powoduje każdorazowo przydzielenie obszaru nieco większego niż żądany. Bloki pamięci dynamicznej mają pewne określone rozmiary minimalne (często są to rozmiary równe kolejnym potęgom dwójki w zakresie od 2 do 16; jest to zależne wyłącznie od architektury systemu operacyjnego). Dalej, wykonanie przydziału wymaga również zarezerwowania kilku dodatkowych bajtów pomocniczych (jest ich zwykle od 8 do 16), aby możliwe było utrzymywanie informacji o blokach zajętych i wolnych. Niekiedy ów narzut pamięciowy jest większy od żądanego rozmiaru przydziału, dlatego procedura `mem.alloc` wywoływana jest raczej celem przydziału pamięci dla dużych obiektów, jak tablice i złożone struktury danych — jej wykorzystywanie do przydziału pojedynczych bajtów jest nieefektywne.

3.12. Instrukcje inc oraz dec

Przykład z poprzedniego podrozdziału uwidaczniał, że jedną z częstszych operacji w języku asemblerowym jest zwiększanie bądź zmniejszanie o jeden wartości jakiegoś rejestru czy zmiennej w pamięci. Częstotliwość występowania tej operacji całkowicie usprawiedliwia obecność w zestawie instrukcji maszynowych procesorów 80x86 pary instrukcji, które taką operację implementują: `inc` (dla zwiększenia o jeden) oraz `dec` (dla zmniejszenia o jeden).

Instrukcje te mają następującą składnię:

```
inc( rej/pam );
dec( rej/pam );
```

Operandem instrukcji może być dowolny rejestr 8-bitowy, 16-bitowy bądź 32-bitowy albo dowolny operand pamięciowy. Instrukcja `inc` powoduje zwiększenie wartości operandu o jeden; instrukcja `dec` zmniejsza wartość operandu o jeden.

Niniejsze instrukcje są realizowane nieco szybciej niż odpowiadające im instrukcje `add` czy `sub` (instrukcje te są kodowane na mniejszej liczbie bajtów). Ich zapis w kodzie maszynowym również jest bardziej oszczędny (w końcu występuje tu tylko jeden operand). Ale to nie koniec różnic pomiędzy parą `inc-dec` a parą `add-sub` — manipulowanie wartością operandu za pośrednictwem instrukcji `inc` i `dec` nie wpływa bowiem na wartość znacznika przeniesienia.

Przykładem zastosowania instrukcji `inc` może być przykład pętli wykorzystany w poprzednim podrozdziale:

```
mem.alloc( 128 );
for( mov( 0, ebx ); ebx < 128; inc( ebx ) ) do

    mov( 0, ( type byte [eax + ebx] ) );

endfor;
```

3.13. Pobieranie adresu obiektu

W podpunkcie 3.1.2.2 omawiane było zastosowanie operatora pobrania adresu (`&`), który zwracał adres zmiennej statycznej[13]. Niestety, operatora tego nie można stosować w odniesieniu do zmiennych automatycznych (deklarowanych w sekcji `var`) ani zmiennych anonimowych; operator ten nie nadaje się też do pobrania adresu odwołania do pamięci realizowanego w trybie indeksowym albo indeksowym skalowanym (nawet jeśli częścią wyrażenia adresowego jest zmienna statyczna). Operator pobrania adresu (`&`) nadaje się więc wyłącznie do określania adresów prostych obiektów statycznych. Tymczasem niejednokrotnie zachodzi potrzeba określenia

[13] Zmienna statyczna to zmienna deklarowana w kodzie źródłowym programu, dla której przydział pamięci odbywa się na etapie kompilacji czy konsolidacji, czyli zmienna deklarowana w sekcjach `static`, `readonly` i `storage`.

adresu również obiektów innych kategorii. Na szczęście w zestawie instrukcji procesorów z rodziny 80x86 przewidziana jest instrukcja załadowania adresu efektywnego `lea` (od *load effective adres*).

Składnia instrukcji `lea` prezentuje się następująco:

```
lea( rejestr32, operand-pamięciowy );
```

Pierwszym z operandów musi być 32-bitowy rejestr. Operand drugi może być dowolnym dozwolonym odwołaniem do pamięci przy użyciu dowolnego z dostępnych trybów adresowania. Wykonanie instrukcji powoduje załadowanie określonego rejestru obliczonym adresem efektywnym. Instrukcja nie wpływa przy tym w żaden sposób na wartość operandu znajdującego się pod obliczonym adresem.

Po załadowaniu adresu efektywnego do 32-bitowego rejestru ogólnego przeznaczenia można wykorzystać adresowanie pośrednie przez rejestr, adresowanie indeksowe, indeksowe skalowane, celem odwołania się do obiektu okupującego określony adres. Spójrzmy na następujący przykład:

```
static
    b:    byte; @nostorage;
          byte 7, 0, 6, 1, 5, 2, 4, 3;
…
    lea( ebx, b );
    for( mov( 0, ecx ); ecx < 8; inc( ecx ) ) do

        stdout.put( "[ebx+ecx]=", (type byte [ebx + ecx]), nl );

    endfor;
```

Powyższy kod inicjuje pętlę, w ramach której następuje wyświetlenie wartości wszystkich kolejnych bajtów nieetykietowanych, począwszy od bajta znajdującego się pod adresem zmiennej `b`. W odwołaniach zastosowany został tryb adresowania `[ebx + ecx]`. Rejestr EBX przechowuje tu adres bazowy sekwencji bajtów (adres pierwszego z bajtów sekwencji), a rejestr ECX definiuje przesunięcie adresu efektywnego, stanowiąc indeks sekwencji.

3.14. Źródła informacji dodatkowych

Pod adresem *http://webster.cs.ucr.edu/* dostępne jest starsze wydanie niniejszej książki, pisane pod kątem procesorów 16-bitowych. Można tam znaleźć informacje o 16-bitowych trybach adresowania procesorów 80x86 i o segmentacji pamięci. Więcej informacji o funkcjach `mem.alloc` i `mem.free` z biblioteki standardowej można znaleźć w podręczniku *HLA Standard Library Manual*, również dostępnym w witrynie Webster pod adresem *http://webster.cs.ucr.edu/*, ewentualnie na stronie WWW pod adresem *http://artofasm.com/*. Oczywiście, znakomitym źródłem informacji na ten temat jest dokumentacja procesorów x86 firmy Intel (do poszukania w witrynie *http://www.intel.com/*), gdzie znajduje się komplet informacji o trybach adresowania i o kodowaniu instrukcji maszynowych.