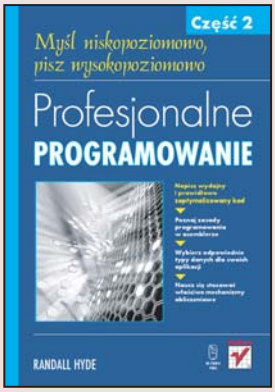
Profesjonalne programowanie. Część 2. Myśl niskopoziomowo, pisz wysokopoziomowo



Autor: Randall Hyde
Tłumaczenie: Mikołaj Szczepaniak, Tomasz Żmijewski
ISBN: 83-246-0463-4
Tytuł oryginału: Write Great Code, Volume 2: Thinking Low-Level, Writing High-Level
Format: B5, stron: 640

Napisz wydajny i prawidłowo zoptymalizowany kod

- Poznaj zasady programowania w asemblerze.
- Wybierz odpowiednie typy danych dla swoich aplikacji.
- Naucz się stosować właściwe mechanizmy obliczeniowe.

Wydajność to jedna z najważniejszych cech aplikacji tworzonych przez profesjonalistów. Należy ją uwzględniać od samego początku prac nad aplikacją. Tymczasem języki wysokiego poziomu i szybkie procesory sprawiły, że dziś programiści kładą niewielki nacisk na wydajność. Jednak źle dobrane typy danych i niewłaściwie użyte instrukcje języka wysokiego poziomu mogą spowodować, że kod maszynowy powstały w wyniku kompilacji nie będzie działać odpowiednio szybko. Utworzenie optymalnego i wydajnego programu może znacznie ułatwić wiedza o tym, jak kod wysokiego poziomu zostanie przekształcony w kod maszynowy.

W książce „Profesjonalne programowanie. Część 2. Myśl niskopoziomowo, pisz wysokopoziomowo" znajdziesz wyczerpujące informacje dotyczące wyboru typów danych i maksymalizowania wydajności aplikacji. Nauczysz się dobierać odpowiednie instrukcje języka wysokiego poziomu tak, aby kompilatory optymalizujące mogły na ich podstawie generować wydajny kod maszynowy. Poznasz także elementy asemblera procesorów 80x86 i PowerPC w zakresie niezbędnym do czytania ze zrozumieniem kodu generowanego przez kompilator.

- Asembler procesorów 80x86 i PowerPC
- Przebieg procesu kompilacji
- Formaty plików wykonywalnych
- Analiza wyników kompilacji
- Organizacja pamięci w trakcie działania programu
- Sposoby przechowywania różnych typów danych w pamięci
- Optymalizacja wyrażeń arytmetycznych
- Struktury sterujące, funkcje i procedury

Poznaj sposób działania kompilatorów i popraw wydajność swoich aplikacji

Wydawnictwo Helion
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

Spis treści

Rozdział 2.

A może warto poznać asemblera?

Wprawdzie w tej książce uczymy Czytelników pisać lepszy kod bez uczenia asemblera, ale najlepsi programiści korzystający z języków wysokiego poziomu jednak asemblera znają; właśnie dzięki temu mogą pisać kod doskonały. W publikacji podajemy 90 procent wiedzy niezbędnej do pisania doskonałego kodu wysokopoziomowego, ale pozostałe 10 procent wymaga właśnie zapoznania się z asemblerem. Wprawdzie nauka programowania w asemblerze wykracza poza zakres niniejszej książki, ale tematów tych nie możemy pominąć, poza tym obowiązkiem autora jest wskazanie Czytelnikom, gdzie szukać dalszych materiałów na ten temat. W rozdziale zajmiemy się następującymi tematami:

- czemu nauka asemblera jest problemem,
- asemblery wysokiego poziomu (HLA) i w czym ułatwiają one naukę asemblera,
- jak użyć gotowych produktów, takich jak Microsoft Macro Assembler (MASM), Borland Turbo Assembler (TASM) i HLA, do ułatwienia nauki asemblera,
- sposób myślenia programisty asemblera (czyli paradygmat programowania w asemblerze),
- zasoby, które można wykorzystać do pogłębienia swojej wiedzy o asemblerze.

2.1. Kłody rzucane pod nogi uczącym się asemblera

Gruntowne poznanie asemblera ma dwie zalety. Po pierwsze, można *dokładnie* zrozumieć kod maszynowy generowany przez kompilator. Jeśli znamy asemblera, mamy wspomniane powyżej 100 procent wiedzy pozwalające nam pisać lepszy kod w językach wysokiego poziomu. Po drugie, zawsze możemy odwołać się do asemblera i zakodować

w nim te części programu, których nasz kompilator nie byłby w stanie zoptymalizować. Kiedy zatem wykorzystamy następne rozdziały do podszlifowania swoich umiejętności programowania wysokopoziomowego, rozsądnym następnym krokiem będzie nauczenie się samego asemblera.

Jednak z uczeniem się asemblera wiąże się pewna pułapka: dawniej nauka asemblera była długim, trudnym i frustrującym zadaniem. Paradygmat programowania w asemblerze jest na tyle różny od programowania w językach wysokiego poziomu, że dla większości ludzi nauka asemblera wydaje się uczeniem się programowania od zera. Frustrujące jest, kiedy doskonale wiemy, jak coś zapisać w C/C++, Javie, Pascalu czy Visual Basicu, a nie potrafimy tego samego zapisać w asemblerze.

Większość programistów, zamiast uczyć się całkiem nowych rzeczy, woli stosować swoją dotychczasową wiedzę. Niestety, tradycyjne metody nauki asemblera oznaczają konieczność porzucenia przyzwyczajeń z języków wysokiego poziomu. Nie jest to, rzecz jasna, najbardziej wydajna metoda nauki. Wobec tego konieczna była metoda, która pozwoliłaby poszerzyć zdobytą dotychczas wiedzę.

2.2. Tom drugi Profesjonalnego programowania spieszy z odsieczą

Po przeczytaniu tej książki nauka asemblera stanie się znacznie łatwiejsza z trzech powodów:

- Wiedząc, jak bardzo znajomość asemblera może pomóc w pisaniu kodu doskonałego, Czytelnik będzie bardziej zmotywowany do nauki.
- W niniejszej książce umieszczono dwa krótkie wprowadzenia do asemblera (jedno na temat asemblera 80x86, drugie na temat asemblera PowerPC), więc nawet osoby, które nie zetknęły się dotąd z asemblerem, poznają przynajmniej podstawy tego języka.
- Kiedy Czytelnik będzie wiedział, jak kompilator zamienia typowe instrukcje sterujące i struktury danych na kod maszynowy, automatycznie opanuje jedno z najtrudniejszych zagadnień przy nauce asemblera: jak zrobić w asemblerze to, co tak prosto robi się w językach wysokiego poziomu.

Niniejsza książka z nikogo nie zrobi asemblerowego eksperta, ale liczne przykłady pokazujące translację kodu wysokiego poziomu na kod maszynowy pozwolą zapoznać się z podstawowymi technikami programowania w asemblerze. Czas na naukę asemblera przyjdzie po przeczytaniu tego tomu.

Oczywiście książka ta będzie łatwiejsza do zrozumienia dla osób znających już asemblera. Z drugiej strony, poznanie asemblera będzie łatwiejsze po przeczytaniu tej książki. Jako że nauka asemblera prawdopodobnie jest bardziej czasochłonna od przeczytania tej pozycji, lepiej zacząć właśnie od czytania.

2.3. Wysokopoziomowe asemblery przychodzą z pomocą

W 1995 roku autor dyskutował z dziekanem wydziału informatyki UC Riverside, narzekając na to, ile czasu zajmuje studentom nauka asemblera i ilu rzeczy muszą się oni przy tej okazji uczyć na nowo. W toku dyskusji oczywistym się stało, że problemem nie jest asembler jako taki, ale pewne aspekty składniowe konkretnych asemblerów, takich jak Microsoft Macro Assembler (MASM). Nauka asemblera to coś znacznie więcej niż nauka kilku instrukcji maszynowych: trzeba też nauczyć się zestawiać te instrukcje w konkretne programy. I *to* właśnie jest w nauce asemblera najtrudniejsze.

Kolejny problem polega na tym, że w *czystym* asemblerze nie można po prostu wybrać kilku instrukcji, aby poprawić wydajność programu. Napisanie choćby najprostszego programu wymaga dużej wiedzy i znajomości kilkudziesięciu instrukcji maszynowych. Kiedy dodamy do tego znajomość budowy komputera, okazuje się, że trzeba przynajmniej kilku tygodni nauki, aby w asemblerze cokolwiek sensownego napisać.

Ważną cechą asemblera MASM z roku 1995 była możliwość użycia instrukcji podobnych do instrukcji wysokopoziomowych, jak `.if`, `.while` i tak dalej. Wprawdzie nie są to instrukcje maszynowe, ale pozwalają one studentom korzystać ze znajomych konstrukcji, a nauką ich niskopoziomowych odpowiedników można zająć się później. Dzięki temu można skupić uwagę studentów na wybranych aspektach asemblera, bez konieczności uczenia od razu bardzo obszernego materiału. Dzięki temu możliwe jest stosunkowo szybkie rozpoczęcie pisania kodu, a w konsekwencji w ciągu kursu można zdobyć większą wiedzę.

Asembler udostępniający instrukcje sterujące podobne do instrukcji wysokopoziomowych nazywany jest *asemblerem wysokiego poziomu*. MASM Microsoftu (wersja 6.0 i nowsze) oraz TASM Borlanda (wersja 5.0 i nowsze) są dobrymi przykładami takiego podejścia. Teoretycznie, mając odpowiedni podręcznik o programowaniu w takich wysokopoziomowych asemblerach, studenci mogliby pisać proste programy w ciągu pierwszego tygodnia kursu.

Jedyny problem związany z asemblerami takimi jak MASM czy TASM polega na tym, że dostępnych jest stosunkowo niewiele wysokopoziomowych instrukcji sterujących i struktur danych. Niemalże wszystko inne jest dla programistów wysokiego poziomu obce. Na przykład deklarowanie danych w MASM i TASM wygląda całkiem inaczej niż ich deklarowanie w językach wysokiego poziomu. Początkujący programiści asemblera nadal muszą posiąść od razu dość dużo wiedzy, mimo istnienia wysokopoziomowych struktur sterujących.

2.4. Asembler wysokopoziomowy (HLA)

Krótko po wspomnianej wcześniej dyskusji z dziekanem autor stwierdził, że właściwie nie ma przeciwwskazań, aby w asemblerze zastosować pewne elementy składniowe wyższego poziomu bez zmiany asemblerowej semantyki. Weźmy na przykład pod uwagę następujące instrukcje C/C++ czy Pascala, w których deklaruje się zmienną tablicową:

```
int intVar[8];  // C/C++

var intVar: array[0..7] of integer;  (* Pascal *)
```

Oto deklaracja takiego samego obiektu w asemblerze MASM:

```
intVar sdword 8 dup (?)  ; MASM
```

Deklaracje C/C++ i Pascala różnią się między sobą, ale deklaracja asemblerowa jest całkowicie odmienna. Programista C/C++ zapewne będzie w stanie zrozumieć deklarację pascalową, nawet jeśli Pascala nie zna. To samo można powiedzieć o programiście pascalowym. Jednak prawdopodobnie żaden z nich nie będzie w stanie zorientować się, o co chodzi w deklaracji asemblerowej. To tylko jeden przykład problemów, przed jakimi stają programiści używający języków wysokiego poziomu chcący nauczyć się asemblera.

Najbardziej irytujące w tym wszystkim jest to, że tak naprawdę nie ma powodu, dla którego deklaracje asemblerowe musiałyby się aż tak bardzo różnić od typowych deklaracji wysokiego poziomu. W ostatecznie uzyskiwanym pliku wykonywalnym to, jakiej składni użyto, nie będzie miało już żadnego znaczenia. Skoro tak, to czemu nie zaadaptować do asemblera pewnych elementów składni języków wysokopoziomowych, aby ułatwić naukę nowym programistom? Właśnie o tym autor dyskutował w 1996 roku ze swoim dziekanem. I te przemyślenia stały się impulsem do stworzenia nowego asemblera, dedykowanego przede wszystkim dla osób znających już jakiś język programowania wysokiego poziomu: HLA, czyli *High-Level Assembler*, asemblera wysokopoziomowego. W HLA pokazana powyżej deklaracja tablicy wygląda następująco:

```
var intVar:int32[8];  // HLA
```

Wprawdzie składnia ta jest nieco inna niż w przypadku C/C++ i Pascala (właściwie jest mieszanką tych ostatnich), ale większość programistów nieznających jeszcze asemblera domyśli się znaczenia pokazanej instrukcji.

Podstawowym założeniem związanym z tworzeniem HLA było zbudowanie takiego środowiska programowania w asemblerze, które byłoby znajome dla programistów znających języki proceduralne, i to bez rezygnowania z możliwości pisania *prawdziwych* programów asemblerowych. Te elementy języka, które nie mają nic wspólnego z instrukcjami maszynowymi, wykorzystują dobrze znaną składnię wysokopoziomową, natomiast instrukcje maszynowe nadal odpowiadają instrukcjom maszynowym procesorów 80x86 jeden do jednego.

Dzięki upodobnieniu w miarę możliwości HLA do języków wysokiego poziomu, uczący się nie muszą poświęcać tyle czasu na zapoznanie się z całkowicie odmienną składnią. Zamiast tego mogą wykorzystać posiadane już umiejętności programistyczne, dzięki czemu nauka jest prostsza i szybsza.

Jednak wygodna składnia deklaracji oraz kilka instrukcji sterujących podobnych do ich wysokopoziomowych odpowiedników to nie wszystko, co można zrobić w celu ułatwienia nauki programowania w asemblerze. Bardzo wiele osób narzeka, że w asemblerach bardzo niewiele zrobiono, aby pomóc programiście: pisząc kod asemblerowy, co i rusz trzeba wyważać otwarte drzwi. Jeśli, na przykład, piszemy cokolwiek w asemblerze MASM czy TASM, zaraz się przekonamy, że środowisko to nie daje nam żadnych narzędzi do obsługi wejścia-wyjścia, choćby do wypisywania liczb całkowitych jako napisów na konsoli. Programista asemblerowy musi takie procedury stworzyć sam. Niestety, napisanie sensownych procedur obsługi wejścia-wyjścia wymaga dużych umiejętności programistycznych w asemblerze. Jedynym sposobem ich pozyskania jest pisanie kodu, a pisanie programów bez procedur wejścia-wyjścia jest trudne. Wobec tego kolejnym składnikiem dobrego środowiska do programowania w asemblerze powinien być zestaw procedur do obsługi wejścia wyjścia, tak aby początkujący programiści mogli bezproblemowo wczytywać na przykład liczby całkowite z konsoli. Na pisanie tego typu procedur samodzielnie czas nadejdzie później, po zdobyciu niezbędnego doświadczenia. W HLA tego typu funkcjonalność jest dostępna jako *biblioteka standardowa HLA*. Biblioteka ta jest zbiorem procedur i makr, które znakomicie ułatwiają pisanie złożonych aplikacji, gdyż wystarczy tylko odpowiednie procedury wywołać.

Z uwagi na stale rosnącą popularność asemblera HLA oraz na to, że HLA jest dostępny za darmo, wraz z kodem źródłowym, że działa w systemach Windows i Linux, w tej książce kod asemblerowy umieszczany w przykładach jest zapisywany zgodnie ze składnią tego właśnie asemblera.

2.5. Myśl na wysokim poziomie, pisz na niskim

HLA stworzono po to, aby początkujący programiści asemblerowi mogli pisać kod niskopoziomowy, myśląc w kategoriach języków wysokiego poziomu — czyli dokładnie odwrotnie, niż uczymy tego w naszej książce. W końcu, oczywiście, każdy programista asemblera zacznie myśleć na niskim poziomie ogólności, ale przy pierwszym zetknięciu z asemblerem możliwość pozostania na wysokim poziomie ogólności stanowi prawdziwe błogosławieństwo: można stosować techniki znane z innych języków.

Wcześniej czy później uczący się rezygnuje z wysokopoziomowych struktur sterujących i zaczyna używać ich odpowiedników niskopoziomowych, jednak pierwszy etap nauki, z wykorzystaniem struktur wysokopoziomowych, pozwala przyswoić sobie inne pojęcia niskopoziomowe; nowe zagadnienia trzeba opanowywać stopniowo, a nie od razu, dzięki czemu nauka odbywa się szybciej.

Ostatecznie jednak celem nauki jest opanowanie paradygmatu programowania niskopoziomowego. Oznacza to rozstanie się z wysokopoziomowymi strukturami sterującymi i pisanie czystego kodu niskiego poziomu. Właśnie to jest myślenie na niskim poziomie i pisanie na niskim poziomie. Jednak bardzo dobrze jest móc zacząć naukę programowania w asemblerze od myślenia na wysokim poziomie przy pisaniu na niskim poziomie. Jest to działanie podobne jak rzucanie palenia z wykorzystaniem plastrów o różnej zawartości nikotyny: ilość nikotyny dostarczanej organizmowi stopniowo się zmniejsza. Tak samo asembler wysokopoziomowy pozwala programiście stopniowo odzwyczajać się od schematów wysokopoziomowych. Rozwiązanie to w przypadku nauki asemblera jest równie skuteczne jak skuteczne jest stopniowe odzwyczajanie się od palenia.

2.6. Paradygmat programowania w asemblerze (myślenie na niskim poziomie)

Programowanie w asemblerze znacząco różni się od programowania w typowych językach wysokiego poziomu. Dlatego właśnie dla wielu programistów uczenie się programowania w asemblerze jest tak trudne. Na szczęście dla autora tej książki, aby ze zrozumieniem czytać wyniki kompilacji, wystarczy rozumieć asemblera; nie trzeba umieć pisać w nim programów. Oznacza to, że nie musimy opanowywać całej sztuki programowania; samo zrozumienie programów asemblerowych pozwoli zrozumieć powody, dla których kompilator generuje takie, a nie inne sekwencje kodu. Za to opiszemy, jak programiści asemblera (i kompilatory) „myślą".

Najważniejszym aspektem asemblerowego paradygmatu programowania[1] jest to, że potrzebne działania trzeba rozbijać na bardzo drobne fragmenty, zrozumiałe dla maszyny. Procesor może w zasadzie wykonywać naraz tylko jedno, bardzo niewielkie zadanie (i dotyczy to nawet procesorów CISC). Wobec tego złożone działania, takie jak instrukcje znane z języków wysokiego poziomu, muszą być rozbijane na mniejsze fragmenty, które da się wykonywać bezpośrednio. Spójrzmy na poniższy przykład — instrukcję przypisania w Visual Basicu:

```
profits = sales - costOfGoods - overhead - commissions
```

Żaden procesor nie umożliwi wykonania całej takiej instrukcji VB w jednej instrukcji maszynowej. Całe to wyrażenie trzeba będzie rozbić na ciąg instrukcji maszynowych wyliczających poszczególne składniki całego przypisania. Na przykład wiele procesorów ma instrukcję odejmowania (*subtract*), która pozwala odejmować od zawartości rejestru jedną wartość. Instrukcja przypisania z naszego przykładu zawiera trzy odejmowania, więc całe przypisanie musimy podzielić na przynajmniej trzy odrębne instrukcje odejmowania.

[1] Przez *paradygmat* rozumiemy model. Paradygmat programowania to model programowania. Wobec tego paradygmat programowania w asemblerze to opis, jak programuje się w asemblerze.

Rodzina procesorów 80x86 zawiera całkiem elastyczną instrukcję odejmowania, `sub`. Instrukcja ta ma następujące postaci (składnia HLA):

```
sub( stała, rejestr );      // rejestr = rejestr - stała
sub( stała, pamięć );       // pamięć = pamięć - stała
sub( rejestr1, rejestr2);   // rejestr2 = rejestr2 - rejestr1
sub( pamięć, rejestr);      // rejestr = rejestr - pamięć
sub( rejestr, pamięć );     // pamięć = pamięć - rejestr
```

Zakładając, że wszystkie identyfikatory z pierwotnego kodu Visual Basica to zmienne, możemy za pomocą instrukcji 80x86 `sub` i `mov` zapisać te same działania jako kod HLA:

```
// Pobierz wartość sales do rejestru EAX:

mov( sales, eax );

// Wylicz sales - costOfGoods  (EAX := EAX - costOfGoods)

sub( costOfGoods, eax );

// Wylicz (sales-costOfGoods) - overhead
// (uwaga: EAX zawiera teraz sales-costOfGoods)

sub( overhead, eax );

// Wylicz (sales-costOfGoods-overhead) - commissions
// (uwaga: EAX zawiera sales-costOfGoods-overhead)

sub( commissions, eax );

// Zapisz wynik (z EAX) w zmiennej profits:

mov( eax, profits );
```

Istotne jest tutaj to, że pojedyncza instrukcja Visual Basica została podzielona na pięć różnych instrukcji HLA, z których każda wykonuje niewielki fragment obliczeń. Cała tajemnica paradygmatu programowania w asemblerze polega na umiejętności rozbijania złożonych działań na proste ciągi instrukcji maszynowych. Zajmiemy się tym ponownie w rozdziale 13.

Instrukcje sterujące języków wysokiego poziomu to kolejna dziedzina, gdzie złożone działania są rozbijane na ciągi prostszych instrukcji. Weźmy na przykład następującą pascalową instrukcję `if`:

```
if( i = j ) then begin

    writeln( "i jest równe j" );

end;
```

W zestawie instrukcji maszynowych obsługiwanych przez procesory nie ma instrukcji `if`. Zamiast tego porównuje się dwie wartości, ustawiając *flagi warunków*, a następnie wykorzystuje się ustawienia tych flag w instrukcjach *skoków warunkowych*. Typowym sposobem przekształcenia powyższej instrukcji `if` na asemblera jest zbadanie warunku przeciwnego (czyli `i <> j`), a następnie przeskoczenie instrukcji, które wykonywane

byłyby w przypadku spełnienia pierwotnego warunku, `i = j`. Oto przykład translacji powyższej instrukcji Pascala na kod HLA (instrukcje *czysto* asemblerowe, czyli bez konstrukcji będących imitacjami konstrukcji wysokopoziomowych):

```
mov( i, eax );      // Pobieraj wartości i
cmp( eax, j );      // Porównuj z wartościami j
jne skipIfBody;     // Pomiń treść instrukcji if - o ile i <> j
<< kod pokazujący napis >>

skipIfBody:
```

W miarę jak wyrażenia używane w strukturach sterujących wysokiego poziomu komplikują się, liczba odpowiadających im instrukcji maszynowych także się zwiększa, ale sama zasada się nie zmienia. Potem (w rozdziałach 14. i 15.) zajmiemy się translacją wysokopoziomowych instrukcji sterujących na asemblera.

Przekazywanie parametrów do procedury czy funkcji, odczyt tych parametrów oraz dostęp do innych danych lokalnych w ramach tych procedur i funkcji — to kolejne przykłady operacji, które w asemblerze jest znacznie trudniej wykonać niż w typowych językach wysokiego poziomu. W tej chwili nie mamy dostatecznej wiedzy, aby pokazać tutaj jakiś przykład (zresztą tworzenie prostych przykładów też nie ma większego sensu), ale zagadnienie to omówimy dalej, w rozdziale 16.

Wniosek z powyższych dywagacji pozostaje niezmienny: kiedy zamieniamy jakiś algorytm zapisany w języku wysokiego poziomu, musimy podzielić dany problem na znacznie mniejsze fragmenty, dające się zakodować w asemblerze. Jak wspomnieliśmy wcześniej, nasi Czytelnicy mają znacznie łatwiejsze zadanie niż zwykli programiści asemblerowi — nie muszą się zastanawiać, jakiej instrukcji maszynowej należy użyć w danej chwili, gdyż to kompilator (albo programista asemblera) tworzy kod, na którym my będziemy pracować. Wystarczy nam znalezienie powiązań między kodem wysokopoziomowym a kodem asemblera. A wiedzę do tego potrzebną będziemy zdobywali w dalszej części niniejszej książki.

2.7. Asembler. Sztuka programowania i inne materiały

Wprawdzie HLA jest doskonałym narzędziem do nauki asemblera, ale jest to narzędzie jeszcze niewystarczające. Aby nauczyć się asemblera na przykładzie HLA, trzeba dysponować dodatkowym zestawem materiałów edukacyjnych. Na szczęście materiały takie istnieją — wynika to z prostego faktu, że HLA stworzono właśnie jako narzędzie do wykorzystania owych materiałów, a nie odwrotnie. Podstawowy zasób, z którego można korzystać, ucząc się asemblera na HLA, jest książka *Asembler. Sztuka programowania* (Helion, 2004). Pozycja ta była pisana z myślą o stopniowej nauce, dokładniej takiej, o jakiej mowa jest w tym rozdziale. Przyjęte założenia sprawdziły się, co mogą poświadczyć tysiące studentów i zwykłych czytelników. Każdy, kto zna jakiś język wysokiego poziomu i chciałby opanować asemblera, powinien na tę książkę zwrócić uwagę.

Oczywiście *Asembler. Sztuka programowania* to nie jedyna pozycja na temat programowania w asemblerze. *Assembly Language Step-by-Step* Jeffa Duntemanna (Wiley, 2000) to książka przeznaczona dla osób, które naukę programowania zaczynają właśnie od asemblera (czyli dla osób niemających praktyki w językach programowania wysokiego poziomu). Wprawdzie Czytelnicy serii *Profesjonalne programowanie* nie należą do tej kategorii, ale dla niektórych takie właśnie podejście może być skuteczniejszą metodą nauki asemblera.

Programming from the Ground Up Jonathona Barletta (Bartlett Publishing, 2004) jest podręcznikiem programowania w asemblerze opartym na implementacji GNU Gas. Książka ta jest szczególnie cenna dla osób, które będą analizowały kod generowany przez kompilator GCC. Starszą, darmową wersję tej książki można znaleźć w Internecie na stronie Webstera, *hhtp://webster.cs.ucr.edu/AsmTools/Gas/index.html*.

Professional Assembly Language (Programmer to Programmer) autorstwa Richarda Bluma (Wrox, 2005) to kolejna książka wykorzystująca asembler GNU na procesory 80x86 i powinna spotkać się z zainteresowaniem osób korzystających z kodu Gas generowanego przez GCC.

Dr Paul Carter jest autorem ciekawego podręcznika *online* programowania w asemblerze. Hiperłącza do aktualnej wersji książki Cartera znaleźć można w witrynie Webstera, *http://webster.cs.ucr.edu/links.htm*.

Oczywiście witryna Webstera oferuje o wiele bogatszy zbiór materiałów poświęconych programowaniu w asemblerze, w tym kilka dostępnych *online* artykułów. Osoby zainteresowane nauką asemblera zdecydowanie powinny tę witrynę znać.

Można znaleźć jeszcze wiele innych materiałów poświęconych programowaniu w asemblerze. Chwila szukania w ulubionej wyszukiwarce internetowej zwróci tysiące stron na ten temat. Istnieją też grupy dyskusyjne, jak *alt.lang.asm* czy *comp.lang.asm.x86*, gdzie można uzyskać odpowiedzi na wiele pytań z dziedziny asemblera. Dostępne są też fora dyskusyjne poświęcone asemblerowi. Chwila pracy z wyszukiwarką dostarczy więcej materiałów niż da się w rozsądnym czasie przeczytać.