

IDŹ DO

PRZYKŁADOWY ROZDZIAŁ



SPIS TREŚCI

KATALOG KSIĄŻEK

KATALOG ONLINE

ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK

DODAJ DO KOSZYKA

CENNIK I INFORMACJE

ZAMÓW INFORMACJE
O NOWOŚCIACH

ZAMÓW CENNIK

CZYTELNIA

FRAGMENTY KSIĄŻEK ONLINE

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 assemblerze.
- 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 działał 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 assemblera procesorów 80x86 i PowerPC w zakresie niezbędnym do czytania ze zrozumieniem kodu generowanego przez kompilator.

- Assembler 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



Spis treści

Podziękowania	11
Wstęp	13
Rozdział 1. Myśl niskopoziomowo, koduj wysokopoziomowo	17
1.1. Nieporozumienia dotyczące jakości kompilatorów	18
1.2. Dlaczego nadal warto uczyć się asemblera?	18
1.3. Czemu znajomość asemblera nie jest absolutnie niezbędna?	19
1.4. Myśl niskopoziomowo	19
1.4.1. Kompilatory są tylko tak dobre, jak dobry kod do interpretacji otrzymują	20
1.4.2. Pomóżmy kompilatorowi generować lepszy kod maszynowy	20
1.4.3. Jak myśleć o asemblerze, pisząc programy w językach wysokiego poziomu?	21
1.5. Pisanie kodu wysokopoziomowego	23
1.6. Założenia	23
1.7. Niezależność od konkretnego języka	24
1.8. Cechy kodu doskonałego	24
1.9. Wymagane środowisko	25
1.10. Dodatkowe informacje	26
Rozdział 2. A może warto poznać asemblera?	27
2.1. Kłody rzucane pod nogi uczącym się asemblera	27
2.2. Tom drugi Profesjonalnego programowania spieszy z odsieczą	28
2.3. Wysokopoziomowe asemblery przychodzą z pomocą	29
2.4. Asembler wysokopoziomowy (HLA)	30
2.5. Myśl na wysokim poziomie, pisz na niskim	31
2.6. Paradigmat programowania w asemblerze (myślenie na niskim poziomie)	32
2.7. Asembler. Sztuka programowania i inne materiały	34
Rozdział 3. Asembler 80x86 dla zwykłego programisty	37
3.1. Dobrze poznać jakiś asembler, ale jeszcze lepiej poznać ich kilka	38
3.2. Składnia asemblera 80x86	38
3.3. Podstawy architektury 80x86	39
3.3.1. Rejestry	39
3.3.2. Rejestry ogólnego przeznaczenia	40
3.3.3. Rejestr EFLAGS	41
3.4. Literały stałych	42
3.4.1. Literały binarne	42
3.4.2. Literały dziesiętne	42

3.4.3.	Literały szesnastkowe	43
3.4.4.	Literały znakowe i łańcuchowe	44
3.4.5.	Liczbowe literały zmiennoprzecinkowe	45
3.5.	Stałe symboliczne w asemblerach	45
3.5.1.	Stałe symboliczne w HLA	46
3.5.2.	Stałe symboliczne w Gas	46
3.5.3.	Stałe symboliczne w MASM i TASM	46
3.6.	Tryby adresowania 80x86	47
3.6.1.	Tryby adresowania rejestrów 80x86	47
3.6.2.	Bezpośrednie podawanie wartości	48
3.6.3.	Tryb adresowania z przesunięciem	49
3.6.4.	Tryb adresowania pośredniego przez rejestr	50
3.6.5.	Indeksowany tryb adresowania	51
3.6.6.	Skalowane tryby adresowania z indeksowaniem	53
3.7.	Deklarowanie danych w językach asemblerowych	55
3.7.1.	Deklarowanie danych w HLA	55
3.7.2.	Deklarowanie danych w MASM i TASM	56
3.7.3.	Deklarowanie danych w Gas	57
3.7.4.	Dostęp do zmiennych bajtowych	57
3.8.	Określanie wielkości operandów w asemblerze	59
3.8.1.	Wymuszanie typów w HLA	60
3.8.2.	Wymuszanie typu w MASM i TASM	60
3.8.3.	Wymuszanie typu w Gas	61
3.9.	Minimalny zestaw instrukcji 80x86	61
3.10.	Dodatkowe informacje	61

Rozdział 4. Asembler PowerPC dla zwykłego programisty 63

4.1.	Dobrze poznać jakiś asembler, ale jeszcze lepiej poznać ich kilka	64
4.2.	Składnia asemblera	64
4.3.	Podstawy architektury PowerPC	64
4.3.1.	Rejestry ogólnego przeznaczenia	65
4.3.2.	Zmiennoprzecinkowe rejestry ogólnego przeznaczenia	65
4.3.3.	Rejestry ogólnego przeznaczenia dostępne w trybie użytkownika	65
4.4.	Literały stałych	68
4.4.1.	Literały binarne	68
4.4.2.	Literały dziesiętne	69
4.4.3.	Literały szesnastkowe	69
4.4.4.	Literały znakowe i łańcuchowe	69
4.4.5.	Literały zmiennoprzecinkowe	69
4.5.	Stałe symboliczne w asemblerze	70
4.6.	Tryby adresowania PowerPC	70
4.6.1.	Tryby adresowania rejestrów PowerPC	70
4.6.2.	Bezpośrednie podawanie wartości	70
4.6.3.	Tryby adresowania pamięci PowerPC	71
4.7.	Deklarowanie danych w językach asemblerowych	72
4.8.	Określanie wielkości operandów w asemblerze	74
4.9.	Minimalny zestaw instrukcji	75
4.10.	Dodatkowe informacje	75

Rozdział 5. Narzędzia do analizy wyników kompilacji 77

5.1.	Typy plików używanych w językach programowania	78
5.2.	Pliki z kodem źródłowym	78
5.2.1.	Pliki źródłowe podzielone na elementy	78
5.2.2.	Specjalne formaty kodu źródłowego	79

5.3. Rodzaje procesorów języków komputerowych	80
5.3.1. Czyste interpretery	80
5.3.2. Interpretery	80
5.3.3. Kompilatory	80
5.3.4. Kompilatory przyrostowe	81
5.4. Proces translacji	82
5.4.1. Analiza leksykalna i tokeny	84
5.4.2. Parsowanie (analiza składniowa)	85
5.4.3. Generowanie kodu pośredniego	86
5.4.4. Optymalizacja	87
5.4.5. Porównanie optymalizacji różnych kompilatorów	97
5.4.6. Generowanie kodu natywnego	97
5.5. Wyniki kompilacji	97
5.5.1. Generowanie przez kompilator kodu źródłowego	98
5.5.2. Generowanie przez kompilator kodu asemblera	99
5.5.3. Generowanie przez kompilator kodu pośredniego	100
5.5.4. Generowanie przez kompilator plików wykonywalnych	101
5.6. Formaty plików z kodem pośrednim	101
5.6.1. Nagłówek pliku COFF	102
5.6.2. Nagłówek opcjonalny COFF	104
5.6.3. Nagłówki sekcji COFF	107
5.6.4. Sekcje COFF	109
5.6.5. Sekcja relokacji	109
5.6.6. Informacje o symbolach i dla programu uruchomieniowego	110
5.6.7. Więcej o formatach plików z kodem pośrednim	110
5.7. Formaty plików wykonywalnych	110
5.7.1. Strony, segmenty i wielkość pliku	111
5.7.2. Fragmentacja wewnętrzna	113
5.7.3. Po co zatem w ogóle oszczędzać miejsce?	113
5.8. Wyrównanie danych i kodu w pliku z kodem pośrednim	115
5.8.1. Dobór wielkości wyrównania sekcji	116
5.8.2. Łączenie sekcji	117
5.8.3. Sterowanie wyrównaniem sekcji	117
5.8.4. Wyrównanie sekcji a moduły biblioteczne	118
5.9. Konsolidatory i ich wpływ na kod	125
5.10. Dodatkowe informacje	128

Rozdział 6. Narzędzia do analizy wyników kompilacji 129

6.1. Tytułem wstępu	130
6.2. Nakazywanie kompilatorowi generowania kodu asemblerowego	131
6.2.1. Asembler z kompilatorów GNU i Borlanda	131
6.2.2. Kod asemblerowy z Visual C++	132
6.2.3. Przykładowy kod asemblerowy	132
6.2.4. Analiza asemblerowych wyników kompilacji	141
6.3. Analiza wyników kompilacji za pomocą narzędzi do kodu pośredniego	142
6.3.1. Narzędzie dumpbin.exe Microsoftu	142
6.3.2. Program FSF/GNU objdump.exe	154
6.4. Użycie deassemblerów do analizy wyników kompilacji	158
6.5. Użycie programu uruchomieniowego do analizy wyników kompilacji	161
6.5.1. Użycie programu uruchomieniowego z IDE	161
6.5.2. Użycie samodzielnego programu uruchomieniowego	162
6.6. Porównywanie wyników dwóch kompilacji	164
6.6.1. Porównywanie wersji za pomocą narzędzia diff	164
6.6.2. Porównywanie ręczne	173
6.7. Dodatkowe informacje	174

Rozdział 7. Stałe a języki wysokiego poziomu	175
7.1. Literały stałych a wydajność programu	176
7.2. Literały stałych a stałe deklarowane	178
7.3. Wyrażenia stałe	179
7.4. Stałe deklarowane a obiekty w pamięci tylko do odczytu	181
7.5. Typy wyliczeniowe	182
7.6. Stałe logiczne	184
7.7. Stałe zmiennoprzecinkowe	186
7.8. Stałe łańcuchowe	191
7.9. Stałe typów złożonych	195
7.10. Dodatkowe informacje	196
Rozdział 8. Zmienne w językach wysokiego poziomu	197
8.1. Organizacja pamięci w trakcie działania programu	197
8.1.1. Sekcje kodu, stałych i danych tylko do odczytu	198
8.1.2. Sekcja zmiennych statycznych	200
8.1.3. Sekcja BSS	201
8.1.4. Sekcja stosu	202
8.1.5. Sekcja serty i dynamiczna alokacja pamięci	203
8.2. Czym jest zmienna?	204
8.2.1. Atrybuty	204
8.2.2. Wiązanie	204
8.2.3. Obiekty statyczne	204
8.2.4. Obiekty dynamiczne	205
8.2.5. Zakres	205
8.2.6. Czas życia	205
8.2.7. Czym zatem jest zmienna?	206
8.3. Zmienne w pamięci	206
8.3.1. Wiązanie statyczne i statyczne zmienne	206
8.3.2. Wiązanie pseudostatyczne i zmienne automatyczne	210
8.3.3. Wiązanie dynamiczne i zmienne dynamiczne	213
8.4. Typowe elementarne typy danych	217
8.4.1. Zmienne całkowitoliczbowe	217
8.4.2. Zmienne zmiennoprzecinkowe, czyli rzeczywiste	220
8.4.3. Zmienne znakowe	221
8.4.4. Zmienne logiczne	222
8.5. Adresy zmiennych a języki wysokiego poziomu	222
8.5.1. Alokacja pamięci na zmienne globalne i statyczne	223
8.5.2. Użycie zmiennych automatycznych w celu zmniejszenia przesunięć	224
8.5.3. Alokacja pamięci na zmienne pośrednie	230
8.5.4. Alokacja pamięci na zmienne dynamiczne i wskaźniki	231
8.5.5. Użycie rekordów (struktur) do zmniejszenia przesunięć	233
8.5.6. Zmienne rejestrowe	235
8.6. Wyrównanie zmiennych w pamięci	236
8.6.1. Rekordy i wyrównanie	241
8.7. Dodatkowe informacje	246
Rozdział 9. Tablicowe typy danych	249
9.1. Czym jest tablica?	249
9.1.1. Deklarowanie tablic	250
9.1.2. Zapis tablic w pamięci	254
9.1.3. Dostęp do elementów tablicy	257
9.1.4. Dopelnianie a kodowanie	259
9.1.5. Tablice wielowymiarowe	262
9.1.6. Tablice dynamiczne a tablice statyczne	275
9.2. Dodatkowe informacje	284

Rozdział 10. Łańcuchowe typy danych	285
10.1. Formaty łańcuchów znakowych	286
10.1.1. Łańcuchy zakończone zerem	286
10.1.2. Łańcuchy poprzedzone długością	302
10.1.3. Łańcuchy 7-bitowe	304
10.1.4. Łańcuchy HLA	305
10.1.5. Łańcuchy oparte na deskryptorach	308
10.2. Łańcuchy statyczne, pseudodynamiczne i dynamiczne	309
10.2.1. Łańcuchy statyczne	309
10.2.2. Łańcuchy pseudodynamiczne	310
10.2.3. Łańcuchy dynamiczne	310
10.3. Zliczanie odwołań do łańcuchów	311
10.4. Łańcuchy w Delphi i Kylixie	311
10.5. Korzystanie z łańcuchów w językach wysokiego poziomu	312
10.6. Dane znakowe w łańcuchach	314
10.7. Dodatkowe informacje	315
Rozdział 11. Wskaźnikowe typy danych	317
11.1. Definiowanie i odkrywanie tajemnic wskaźników	318
11.2. Implementacje wskaźników w językach wysokopoziomowych	319
11.3. Wskaźniki i dynamiczny przydział pamięci	322
11.4. Operacje na wskaźnikach i arytmetyka wskaźników	323
11.4.1. Dodawanie liczby całkowitej do wskaźnika	324
11.4.2. Odejmowanie liczby całkowitej od wskaźnika	326
11.4.3. Odejmowanie wskaźnika od wskaźnika	327
11.4.4. Porównywanie wskaźników	328
11.4.5. Stosowanie logicznych operatorów AND i OR dla wskaźników	330
11.4.6. Pozostałe operatory wskaźnikowe	331
11.5. Prosty przykład alokatora pamięci	333
11.6. Odzyskiwanie pamięci	336
11.7. Przydział pamięci na poziomie systemu operacyjnego	337
11.8. Dodatkowa złożoność pamięciowa menedżera sterty	338
11.9. Typowe problemy związane ze wskaźnikami	341
11.9.1. Używanie wskaźnika niezainicjalizowanego	341
11.9.2. Używanie wskaźnika zawierającego nieprawidłową wartość	342
11.9.3. Korzystanie z bloku pamięci już po jej zwolnieniu	343
11.9.4. Zaniedbywanie zwalniania niepotrzebnej pamięci	344
11.9.5. Uzyskiwanie dostępu do danych za pośrednictwem danych błędnego typu	345
11.10. Dodatkowe informacje	346
Rozdział 12. Rekordy, unie i klasowe typy danych	347
12.1. Rekordy	348
12.1.1. Deklaracje rekordów w różnych językach programowania	349
12.1.2. Tworzenie egzemplarza rekordu	350
12.1.3. Inicjalizacja danych rekordu w czasie kompilacji	357
12.1.4. Pamięciowe reprezentacje rekordów	361
12.1.5. Podnoszenie efektywności pamięciowej za pomocą rekordów	364
12.1.6. Dynamiczne typy rekordowe i bazy danych	366
12.2. Unie wyróżnikowe	367
12.3. Deklaracje unii w różnych językach programowania	368
12.3.1. Deklaracje unii w języku C/C++	368
12.3.2. Deklaracje unii w języku Pascal (Delphi, Kylix)	368
12.3.3. Deklaracje unii w języku HLA	369

12.4. Pamięciowa reprezentacja unii	370
12.5. Pozostałe zastosowania unii	371
12.6. Typy wariantowe	372
12.7. Przestrzenie nazw	377
12.8. Klasy i obiekty	379
12.8.1. Klasy kontra obiekty	380
12.8.2. Prosta deklaracja klasy w języku C++	380
12.8.3. Tabele metod wirtualnych	381
12.8.4. Współdzielenie tabel metod wirtualnych	385
12.8.5. Dziedziczenie w klasach	386
12.8.6. Polimorfizm w klasach	389
12.8.7. Klasy, obiekty i problem wydajności	390
12.9. Dodatkowe informacje	391

Rozdział 13. Wyrażenia arytmetyczne i logiczne 393

13.1. Wyrażenia arytmetyczne w kontekście architektury komputera	394
13.1.1. Maszyny stosowe	394
13.1.2. Maszyny akumulatorowe	399
13.1.3. Maszyny rejestrowe	401
13.1.4. Typowe formy zapisu wyrażeń arytmetycznych	403
13.1.5. Architektury trójadresowe	403
13.1.6. Architektury dwuadresowe	404
13.1.7. Różnice architektoniczne w kontekście Twojego kodu	404
13.1.8. Obsługa wyrażeń złożonych	405
13.2. Optymalizacja wyrażeń arytmetycznych	406
13.2.1. Składanie stałych	407
13.2.2. Propagacja stałych	408
13.2.3. Eliminacja martwego kodu	409
13.2.4. Eliminacja powtarzających się podwyrażeń	411
13.2.5. Redukcja złożoności wyrażeń	415
13.2.6. Indukcja	420
13.2.7. Niezmienne wyrażenia w pętlach	422
13.2.8. Mechanizmy optymalizujące i programiści	425
13.3. Skutki uboczne stosowania wyrażeń arytmetycznych	427
13.4. Skutki uboczne zawierania — punkty sekwencji	432
13.5. Eliminowanie problemów wynikających z występowania skutków ubocznych	436
13.6. Wymuszanie określonego porządku przetwarzania kodu	437
13.7. Skrócone wyznaczanie wartości wyrażeń arytmetycznych	439
13.7.1. Skrócone wyznaczanie wartości wyrażeń logicznych	440
13.7.2. Wymuszanie skróconego lub pełnego wyznaczania wartości wyrażeń logicznych	443
13.7.3. Skrócone wyznaczanie wartości wyrażeń a kwestia wydajności	444
13.8. Względne koszty operacji arytmetycznych	449
13.9. Dodatkowe informacje	450

Rozdział 14. Struktury sterujące i decyzje programowe 453

14.1. Struktury sterujące są wolniejsze od wyrażeń arytmetycznych!	454
14.2. Wprowadzenie do niskopoziomowych struktur sterujących	454
14.3. Rozkaz goto	458
14.4. Wyrażenia break, continue, next, return i inne ograniczone formy wyrażenia goto	462

14.5. Wyrażenie if	463
14.5.1. Optymalizacja niektórych konstrukcji if-else	466
14.5.2. Wymuszanie pełnego wyznaczania wartości wyrażeń logicznych stosowanych w wyrażeniach if	469
14.5.3. Wymuszanie skróconego wyznaczania wartości wyrażeń logicznych stosowanych w wyrażeniach if	474
14.6. Wyrażenie switch (case)	480
14.6.1. Semantyka wyrażenia switch (case)	482
14.6.2. Tabele skoków kontra sekwencje porównań	482
14.6.3. Pozostałe implementacje konstrukcji switch (case)	490
14.6.4. Kod maszynowy generowany przez kompilatory na podstawie wyrażeń switch	501
14.7. Dodatkowe informacje	502
Rozdział 15. Iteracyjne struktury sterujące	505
15.1. Pętla while	505
15.1.1. Wymuszanie pełnego wyznaczania wartości wyrażeń logicznych w pętli while	509
15.1.2. Wymuszanie skróconego wyznaczania wartości wyrażeń logicznych w pętli while	518
15.2. Pętla repeat..until (do..until lub do..while)	521
15.2.1. Wymuszanie pełnego wyznaczania wartości wyrażeń logicznych w pętli repeat..until	524
15.2.2. Wymuszanie skróconego wyznaczania wartości wyrażeń logicznych w pętli repeat..until	527
15.3. Pętla forever..endfor	532
15.3.1. Wymuszanie pełnego przetwarzania wyrażenia logicznego wykorzystywanego w pętli forever	535
15.3.2. Wymuszanie skróconego przetwarzania wyrażenia logicznego wykorzystywanego w pętli forever	535
15.4. Pętla określona (pętla for)	535
15.5. Dodatkowe informacje	537
Rozdział 16. Funkcje i procedury	539
16.1. Proste wywołania funkcji i procedur	540
16.1.1. Składowanie adresu zwracanej wartości	543
16.1.2. Pozostałe źródła opóźnień	547
16.2. Funkcje-liście i procedury-liście	548
16.3. Makra i funkcje wbudowane	553
16.4. Przekazywanie parametrów do funkcji lub procedury	559
16.5. Rekordy aktywacji i stos	566
16.5.1. Struktura rekordu aktywacji	569
16.5.2. Przypisywanie przesunięć zmiennym lokalnym	572
16.5.3. Wiązanie przesunięć z parametrami	575
16.5.4. Uzyskiwanie dostępu do parametrów i zmiennych lokalnych	580
16.6. Mechanizmy przekazywania parametrów	589
16.6.1. Przekazywanie przez wartość	590
16.6.2. Przekazywanie przez referencję	590
16.7. Wartości zwracane przez funkcje	592
16.8. Dodatkowe informacje	599

Dodatek A Inżynieria oprogramowania 601

**Dodatek B Krótkie zestawienie informacji o rodzinach procesorów 80x86
oraz PowerPC 603**

B.1. Różnice architektoniczne pomiędzy technologiami RISC i CISC	604
B.1.1. Zakres zadań realizowanych przez pojedyncze rozkazy	604
B.1.2. Rozmiar rozkazu	605
B.1.3. Częstotliwość taktowania zegara i współczynnik liczby cykli na rozkaz ...	606
B.1.4. Dostęp do pamięci i tryby adresowania	607
B.1.5. Rejestry	608
B.1.6. Operandy bezpośrednie (stałe)	608
B.1.7. Stosy	609
B.2. Kompilatory i interfejs ABI	609
B.3. Profesjonalne programowanie dla obu architektur	610

Dodatek C Dodatki internetowe 613

Skorowidz 615

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 nauczanie 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 przyzwyczajień 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 robi 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 *assemblerem 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 pojąć 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 mieszkanką 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 assemblerze. Bardzo wiele osób narzeka, że w assemblerach bardzo niewiele zrobiono, aby pomóc programiście: pisać kod assemblerowy, co i rusz trzeba wywierać otwarte drzwi. Jeśli, na przykład, piszemy cokolwiek w assemblerze 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 assemblerowy musi takie procedury stworzyć sam. Niestety, napisanie sensownych procedur obsługi wejścia-wyjścia wymaga dużych umiejętności programistycznych w assemblerze. 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 assemblerze 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 odpowiednio 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 assemblerowy 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 assemblerowi 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 assemblerem 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 assemblerze 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 assembler wysokopoziomowy pozwala programiście stopniowo odzwyczajając się od schematów wysokopoziomowych. Rozwiązanie to w przypadku nauki assemblera jest równie skuteczne jak skuteczne jest stopniowe odzwyczajanie się od palenia.

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

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

Najważniejszym aspektem assemblerowego paradygmatu programowania¹ 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 assemblerze to opis, jak programuje się w assemblerze.

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 assemblerze polega na umiejętności rozbiwania 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 pascelową 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 assemblerze. *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 assemblerze 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, <http://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 assemblerze. 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 assemblerze, w tym kilka dostępnych *online* artykułów. Osoby zainteresowane nauką asemblera zdecydowanie powinny tę witrynę znać.

Można znaleźć jeszcze wiele innych materiałów poświęconych programowaniu w assemblerze. 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.