Praktyczne rozwiązania
dla zaawansowanych programistów

UCZTA PROGRAMISTÓW

Henry S. Warren, Jr.

$1111^2 = 11100001$

$$\mathrm{pop}(x) = -\sum_{i=0}^{31} (x \overset{rot}{\ll} i)$$

George Boole
1815 - 1864

$$n = -2^{31}b_{31} + 2^{30}b_{30} + 2^{29}b_{29} + \ldots + 2^0 b_0$$

$$\frac{1}{3} = 0.01010101\ldots$$

$$x \oplus y = (x \mid y) - (x \,\&\, y)$$

$$x + y = (x \mid y) + (x \,\&\, y)$$

$$x - y = x + \bar{y} + 1$$

$$\lceil x \rceil = -\lfloor -x \rfloor$$

$$-\bar{x} = x + 1$$

Num factors of 2 in x = $\log_2(x \,\&\, (-x)),\ x \neq 0$

$$2^{2^5} + 1 = 641 \cdot 6700417$$

$$2^{2^6} + 1 = 274177 \cdot 67280421310721$$

$$\lfloor a \rfloor + \lfloor b \rfloor \leq \lfloor a + b \rfloor \leq \lfloor a \rfloor + \lfloor b \rfloor + 1$$

$$\left\lfloor \sqrt{11111111} \right\rfloor = 1111$$

$$p_n = 1 + \sum_{m=1}^{2^n} \left\lfloor \sqrt[n]{n} \left(\sum_{x=1}^{m} \left\lfloor \cos^2 \pi \frac{(x-1)! + 1}{x} \right\rfloor \right)^{-1/n} \right\rfloor$$

ADDISON-WESLEY

Helion

Podziękowania

W pierwszej kolejności dziękuję Bruce'owi Shiverowi oraz Dennisowi Allisonowi za zachęcenie mnie do opublikowania tej książki. Jestem również wdzięczny wielu kolegom z IBM, prace niektórych z nich zostały wspomniane w bibliografii. Jeden z nich szczególnie zasłużył na wzmiankę. Jest to Martin E. Hopkins, którego nazywam „pan kompilator". Był niestrudzonym poszukiwaczem każdego zaoszczędzonego cyklu maszynowego. Jestem pewien, że jego postawa miała na mnie wielki wpływ.

Ulepszyć tę książkę bardzo pomogli redaktorzy z Addison-Wesley. Większość z nazwisk tych ludzi nie jest mi znana, lecz uwagi jednego z nich, którego nazwisko zdołałem poznać, były niesamowite. Guy L. Steele Jr. przedstawił 50-stronicowe opracowanie, zawierające nowe zagadnienia, którymi warto się zająć, takie jak mieszanie (ang. *shuffling*) i porządkowanie (ang. *unshuffling*) bitów, operacje typu „owce i kozły" (ang. *sheep and goats*) oraz wiele innych, które muszą zaczekać do drugiego wydania. Zasugerował algorytmy, które na głowę biły te zaproponowane przeze mnie. Był niesamowicie dociekliwy. Na przykład popełniłem błąd w rozpisaniu na czynniki pierwsze liczby szesnastkowej AAAAAAAA, którą rozpisałem jako następujący iloczyn 2 * 3 * 17 * 257 * 65537. Guy poprawił mnie, że zamiast liczby 3 w powyższym działaniu powinna być liczba 5. Zasugerował mnóstwo poprawek odnośnie stylu i nie wzdragał się nawet przed najdrobniejszymi szczegółami.

H. S. Warren, Jr.

Yorktown, New York

Luty 2002

Spis treści

Przedmowa **9**

Wstęp **11**

Rozdział 1. Wprowadzenie **13**
1.1. Notacja 13
1.2. Zestaw instrukcji i model wykonawczy 17

Rozdział 2. Podstawy **23**
2.1. Manipulowanie prawostronnymi bitami 23
2.2. Łączenie dodawania z operacjami logicznymi 27
2.3. Nierówności w wyrażeniach logicznych i arytmetycznych 29
2.4. Wartość bezwzględna 30
2.5. Rozszerzenie o znak 31
2.6. Przesunięcie w prawo ze znakiem za pomocą instrukcji przesunięcia bez znaku 32
2.7. Funkcja signum 32
2.8. Funkcja porównania trójwartościowego 33
2.9. Przeniesienie znaku 34
2.10. Dekodowanie pola „zero oznacza 2^n" 34
2.11. Predykaty porównań 35
2.12. Wykrywanie przepełnienia 40
2.13. Kod warunkowy operacji dodawania, odejmowania i mnożenia 49
2.14. Przesunięcia cykliczne 50
2.15. Dodawanie i odejmowanie liczb o podwójnej długości 51
2.16. Przesunięcia liczb o podwójnej długości 52
2.17. Operacje dodawania, odejmowania i wyznaczania wartości bezwzględnej na wartościach wielobajtowych 53
2.18. Doz, Max oraz Min 54
2.19. Wymiana wartości między rejestrami 56
2.20. Wymiana dwóch lub większej liczby wartości 59

Rozdział 3. Ograniczenia potęg dwójki **63**
3.1. Zaokrąglanie do wielokrotności znanych potęg liczby 2 63
3.2. Zaokrąglanie w górę lub w dół do następnej potęgi liczby 2 64
3.3. Wykrywanie przekroczenia ograniczeń potęgi dwójki 67

Rozdział 4. Ograniczenia arytmetyczne **71**
4.1. Kontrola ograniczeń liczb całkowitych 71
4.2. Ograniczenia zakresów w operacjach sumy i różnicy 74
4.3. Ograniczenia zakresów w operacjach logicznych 78

Rozdział 5. Zliczanie bitów ... 85
5.1. Zliczanie jedynek ... 85
5.2. Parzystość ... 94
5.3. Zliczanie zer wiodących ... 97
5.4. Zliczanie zer końcowych ... 104

Rozdział 6. Przeszukiwanie słów ... 111
6.1 Wyszukiwanie pierwszego bajtu o wartości 0 ... 111
6.2. Wyszukiwanie pierwszego ciągu jedynek o zadanej długości ... 117

Rozdział 7. Manipulacja bitami i bajtami ... 121
7.1. Odwracanie kolejności bitów i bajtów ... 121
7.2. Tasowanie bitów ... 126
7.3. Transponowanie macierzy bitów ... 128
7.4. Kompresja lub uogólniona ekstrakcja ... 137
7.5. Uogólnione permutacje, operacja typu „owce i kozły” ... 143
7.6. Zmiana kolejności oraz transformacje oparte na indeksach ... 148

Rozdział 8. Mnożenie ... 151
8.1. Mnożenie czynników wieloelementowych ... 151
8.2. Bardziej znacząca połowa 64-bitowego iloczynu ... 154
8.3. Konwersje między bardziej znaczącą połową 64-bitowego iloczynu ze znakiem i bez znaku ... 155
8.4. Mnożenie przez stałe ... 156

Rozdział 9. Dzielenie całkowitoliczbowe ... 161
9.1. Warunki wstępne ... 161
9.2. Dzielenie wartości wieloelementowych ... 165
9.3. Krótkie dzielenie bez znaku za pomocą dzielenia ze znakiem ... 170
9.4. Długie dzielenie bez znaku ... 173

Rozdział 10. Dzielenie liczb całkowitych przez stałe ... 181
10.1. Dzielenie ze znakiem przez znaną potęgę liczby 2 ... 181
10.2. Reszta ze znakiem z dzielenia przez znaną potęgę liczby 2 ... 182
10.3. Dzielenie i reszta ze znakiem w przypadku innych wartości niż potęgi liczby 2 ... 184
10.4. Dzielenie ze znakiem przez dzielniki ≥ 2 ... 187
10.6. Zawieranie obsługi w kompilatorze ... 197
10.7. Inne zagadnienia ... 201
10.8. Dzielenie bez znaku ... 205
10.9. Dzielenie bez znaku przez dzielniki ≥ 1 ... 207
10.10. Zawieranie obsługi w kompilatorze ... 210
10.11. Inne zagadnienia (dzielenia bez znaku) ... 212
10.12. Zastosowalność w przypadku dzielenia z modułem i dzielenia z funkcją podłoga ... 215
10.13. Podobne metody ... 215
10.14. Przykładowe liczby magiczne ... 216
10.15. Dzielenie dokładne przez stałe ... 217
10.16. Sprawdzanie zerowej reszty po wykonaniu dzielenia przez stałą ... 225

Rozdział 11. Niektóre funkcje podstawowe ... 231
11.1. Całkowitoliczbowy pierwiastek kwadratowy ... 231
11.2 Całkowitoliczbowy pierwiastek sześcienny ... 239
11.3. Całkowitoliczbowe podnoszenie do potęgi ... 241
11.4. Logarytm całkowitoliczbowy ... 243

Rozdział 12. Niezwykłe podstawy systemów liczbowych 251
12.1. Podstawa –2 251
12.2. Podstawa –1 + i 258
12.3. Inne podstawy 261
12.4. Najbardziej wydajna podstawa 262

Rozdział 13. Kod Graya 263
13.1. Kod Graya 263
13.2. Zwiększanie wartości liczb całkowitych zakodowanych w kodzie Graya 266
13.3. Ujemno-binarny kod Graya 267
13.4. Rys historyczny i zastosowania 268

Rozdział 14. Krzywa Hilberta 269
14.1. Rekurencyjny algorytm generowania krzywej Hilberta 271
14.2. Określanie współrzędnych na podstawie odległości wzdłuż krzywej Hilberta 273
14.3. Określanie odległości na podstawie współrzędnych na krzywej Hilberta 280
14.4. Zwiększanie wartości współrzędnych na krzywej Hilberta 282
14.5. Nierekurencyjny algorytm generowania 285
14.6. Inne krzywe wypełniające przestrzeń 285
14.7. Zastosowania 286

Rozdział 15. Liczby zmiennopozycyjne 289
15.1. Standard IEEE 289
15.2. Porównywanie liczb zmiennopozycyjnych za pomocą operacji całkowitoliczbowych 292
15.3. Rozkład cyfr wiodących 293
15.4. Tabela różnych wartości 295

Rozdział 16. Wzory na liczby pierwsze 299
16.1. Wprowadzenie 299
16.2. Wzory Willansa 301
16.3. Wzory Wormella 305
16.4. Wzory na inne trudne funkcje 306

Dodatek A Tablice arytmetyczne dla maszyny 4-bitowej 313

Dodatek B Metoda Newtona 319

Bibliografia 321

Skorowidz 325

Przedmowa

Kiedy ponad 30 lat temu miałem sposobność zdobyć wakacyjne zatrudnienie przy MIT Project MAC, z wielką przyjemnością pracowałem z komputerem DEC PDP-10. Komputer ten był najwspanialszą platformą, na której można było programować w asemblerze. Mój entuzjazm w stosunku do tej maszyny wziął się z bogactwa udostępnianych przez nią instrukcji: umożliwiających operacje na bitach, maskowanie bitów czy też operacje na liczbach całkowitych. Pomimo to, że PDP-10 nie są już produkowane od wielu lat, ciągle istnieją grupy entuzjastów, którzy opiekują się starym sprzętem PDP-10 a także tacy, którzy wykorzystują programy na tę platformę — całe systemy operacyjne i oprogramowanie — wykorzystując do tego celu emulatory PDP-10 na komputery osobiste. Pisuje się nawet nowe programy — istnieje przynajmniej jeden serwis WWW, który działa na symulowanym systemie PDP-10. Nie ma w tym nic śmiesznego — w każdym razie nie bardziej niż pasjonowanie się zabytkowymi samochodami.

Tego samego lata, w roku 1972, zaczytywałem się całkiem nowym periodykiem MIT o nazwie HACKMEM, dziwaczną i eklektyczną mieszanką ciekawostek technicznych. Tematyka tej publikacji obejmowała zagadnienia od obwodów elektrycznych do teorii liczb, jednak rzeczą, która ciekawiła mnie najbardziej był niewielki katalog niesamowitych, niewielkich sztuczek programistycznych. Każda z tych perełek stanowiła użyteczną, lecz bardzo nieszablonową operację na liczbach całkowitych lub ciągach bitów, jak na przykład zliczenie bitów o wartości 1 w słowie. Każda z tych sztuczek mogła być zapisana w postaci ustalonej, dość długiej sekwencji instrukcji maszynowych lub za pomocą pętli. Następnie przedstawiany był sprytny sposób, w jaki ten sam efekt można było osiągnąć za pomocą zaledwie czterech, trzech lub nawet dwóch instrukcji, których połączone działanie było najczęściej mało oczywiste, zanim nie zostało dogłębnie wytłumaczone lub przemyślane. Dla mnie pożeranie tych specjałów było jak gryzienie orzeszków albo raczej słodyczy — po prostu nie mogłem przestać. I było w nich jakieś bogactwo, swoista głębia intelektualna, elegancja a nawet wręcz poetyka.

Pomyślałem wtedy, że z całą pewnością istnieje więcej takich sztuczek. I rzeczywiście, z biegiem lat zebrałem, a w niektórych przypadkach samodzielnie opracowałem kilka nowych. Następnie pomyślałem, że powinna powstać książka na ich temat.

Gdy dostałem do ręki rękopis Hanka Warrena, byłem poruszony do głębi. Znalazłem systematyczną kolekcję niewielkich sztuczek programistycznych, pogrupowanych tematycznie i dokładnie objaśnionych. Niektóre z nich mogą być opisane za pomocą instrukcji maszynowych, lecz nie jest to książka wyłącznie dla programujących w asemblerze. Głównym celem książki jest ukazanie podstawowych związków pomiędzy liczbami całkowitymi a ciągami bitów oraz omówienie efektywnych sposobów wykonywania operacji na danych tego typu.

Skąd wzięła się nazwa HACKMEM? Jest to skrót od określenia *hacker's memo* (notatka hakera). Jedno 36-bitowe słowo w PDP-10 mogło przechować sześć 6-bitowych znaków, dlatego wiele nazw, którymi posługiwali się hakerzy PDP-10, składało się maksymalnie z sześciu znaków. Przywykliśmy do tego, że jeden rzut oka na sześcioznakową nazwę pozwalał na sprawne odczytanie pełnej nazwy. Dlatego nazwa HACKMEM miała dużo sensu, w każdym razie dla hakerów.

Techniki opisane w niniejszej książce są równie przydatne dla osób programujących w asemblerze, jak również dla programistów C czy Javy.

Wiele książek dotyczących algorytmów zawiera wykłady skomplikowanych technik sortowania i wyszukiwania, zarządzania tablicami mieszającymi (ang. *hash tables*) i drzewami binarnymi (ang. *binary trees*), wykorzystania rekordów i wskaźników. Najczęściej jednak ignorują zagadnienia, które można zrealizować za pomocą bardzo małych struktur danych — bitów i tablic bitów. Sposoby, na które można wykorzystać operacje sumy i różnicy bitowej oraz operacje na bitach są po prostu zdumiewające. Świadomość faktu, że łańcuch przeniesień pozwala pojedynczym bitom na kontrolę stanu bitów po ich lewej stronie powoduje, że operacja dodawania staje się bardzo użytecznym mechanizmem manipulacji danymi. Wiedza tego typu nie należy do szeroko rozpowszechnionych.

Tak, książka na temat tych technik musiała powstać. Trzymasz ją teraz w swoich rękach i jest to wspaniałe dzieło. Jeśli tworzysz kompilatory optymalizujące kod pod kątem wydajności, jest to lektura obowiązkowa. W przeciwnym wypadku istnieje ryzyko, że każdego dnia mógłbyś przeoczyć tę listę sposobów optymalizacji. Jeśli znalazłeś się w sytuacji, gdy musisz przejrzeć po kolei pojedyncze bity w słowie lub wykonać jakieś specyficzne operacje na liczbach całkowitych albo też przyśpieszyć dwukrotnie jakąś operację wykonującą skomplikowane obliczenia całkowitoliczbowe lub przekształcenia bitowe, niniejsza książka jest odpowiednim źródłem informacji. Być może również zdecydujesz się przeczytać ją dla czystej przyjemności lektury.

Guy L. Steele, Jr.

Burlington, Massachusetts

Kwiecień 2002

Wstęp

Koszt pielęgnacji oprogramowania rośnie wraz z kwadratem kreatywności programisty.

Pierwsze prawo kreatywności programistycznej,
Robert D. Bliss, 1992.

Oto kolekcja niewielkich sztuczek programistycznych, które zbierałem przez wiele lat. Wiele z nich będzie działać poprawnie wyłącznie na komputerach reprezentujących liczby całkowite w formacie uzupełnienia dwójkowego. Pomimo to, że w większości przypadków zakładamy rozmiar rejestru jako 32 bity, większość z tych sztuczek można z łatwością zaadaptować do wykorzystania na maszynach o innym rozmiarze rejestrów.

Niniejsza książka nie omawia dużych algorytmów, takich jak zaawansowane sortowanie czy techniki optymalizacyjne kompilatorów. Zajmuje się natomiast niewielkimi sztuczkami, obejmującymi pojedyncze słowa i pojedyncze instrukcje procesora, jak na przykład zliczanie bitów o wartości 1 w słowie. Sztuczki tego typu wykorzystują najczęściej mieszankę instrukcji arytmetycznych i logicznych.

W całej książce zostało przyjęte założenie, że przerwania związane z przepełnieniem liczb całkowitych zostały zamaskowane, zatem nie mogą wystąpić. Języki C, Fortran a nawet Java spełniają ten wymóg, lecz osoby programujące w językach ADA czy też Pascal powinny mieć się na baczności.

Prezentacja zagadnień jest nieformalna. Dowody poprawności są przedstawiane wyłącznie w tych przypadkach, gdy algorytmy nie są oczywiste, bywa również, że nawet w takich sytuacjach nie przedstawiamy dowodów. Techniki wykorzystują arytmetykę komputerową, funkcję *podłoga*, mieszankę operacji arytmetycznych i logicznych itd. W takiej dziedzinie zagadnień dowody są często trudne do sformalizowania.

W celu zmniejszenia liczby błędów typograficznych i przeoczeń wiele z prezentowanych algorytmów zostało faktycznie uruchomionych. W takim przypadku prezentujemy je w prawdziwym języku programowania, mimo że tak jak każdy język programowania posiada on niepożądane cechy. Zastosowano język C, głównie dlatego, że jest

powszechnie znany, pozwala na proste zastosowanie mieszanki operacji całkowitoliczbowych i bitowych i istnieją kompilatory tego języka, które generują wysoce zoptymalizowany kod wynikowy.

W niektórych przypadkach został zastosowany kod maszynowy. Wykorzystuje format trójadresowy, głównie w celu zwiększenia czytelności. Zastosowany asembler nie jest „prawdziwym" językiem, reprezentuje fikcyjną maszynę obsługującą mechanizmy współczesnych procesorów w architekturze RISC.

Wszędzie, gdzie to możliwe stosowany jest kod bez rozgałęzień. Powodem jest to, że na większości komputerów instrukcje rozgałęzień spowalniają proces pobierania kolejnych instrukcji, jak również utrudniają równoległe wykonywanie. Innym problemem z rozgałęzieniami jest to, że powodują utrudnienia w zakresie działania mechanizmów optymalizacji zawartych w kompilatorach, takich jak szeregowanie instrukcji (ang. *instruction scheduling*), usuwanie podwyrażeń wspólnych (ang. *commoning*) czy też przydział rejestrów (ang. *register allocation*). Oznacza to przede wszystkim, że kompilator będzie bardziej efektywnie optymalizował kilka większych bloków kodu niż wiele małych fragmentów.

Proponowany kod wykorzystuje również niewielkie wartości bezpośrednie, porównania z zerem (zamiast porównań z inną liczbą) oraz równoległość na poziomie instrukcji. W niektórych przypadkach kod byłby bardziej czytelny, gdyby została zastosowana tablica przeglądowa (ang. *lookup table*) w pamięci lecz staramy się o tym nie wspominać. Powodem tego jest większy koszt czasowy załadowania danych z pamięci w stosunku do operacji arytmetycznych, ponadto techniki oparte na tablicach przeglądowych nie są szczególnie interesujące (choć często są bardzo praktyczne). Przypadki takie są jednak dość rzadkie.

Na końcu muszę nawiązać do terminu „haker" z tytułu książki. W klasycznym rozumieniu „haker" jest osobą zafascynowaną komputerami — kimś, kto znajduje radość w zmuszaniu komputera do robienia nowych rzeczy lub robienia rzeczy starych w nowy, sprytniejszy sposób. Haker jest zwykle bardzo dobry w tym, co robi, lecz często zdarza się, że nie jest zawodowym programistą lub projektantem. Praca hakera może być użyteczna ale może też być po prostu zabawą. Przykładem takiego przypadku może być program, który po uruchomieniu wypisuje kopię samego siebie[1] — „problem", którym zajmowało się wielu znanych hakerów. Właśnie w takim znaczeniu stosujemy termin „haker". Jeśli ktoś szuka wskazówek, w jaki sposób włamywać się do komputerów, nie znajdzie ich w tej książce.

---

[1] Najkrótszym programem tego typu w języku C znanym Autorowi jest następujący kod o długości 64 znaków, którego autorami są Vlad Taeerov oraz Rashit Fakhreyev:

```
main(a){printf(a,34,a="main(a){printf(a,34,a=%c%s%c,34);}",34);}
```

Rozdział 1.

Wprowadzenie

1.1. Notacja

W niniejszej książce stosujemy dwa rodzaje notacji: notację wyrażeń matematycznych oraz notację operacji wykonywanych przez komputer. W arytmetyce komputerowej argumenty operacji są ciągami bitów lub wektorami bitów o pewnej określonej długości. Wyrażenia arytmetyki komputerowej są podobne do wyrażeń matematycznych z tą różnicą, że wartości zmiennych odpowiadają wartościom rejestrów procesora. Wartości w arytmetyce komputerowej są po prostu ciągami bitów niepodlegającymi ustalonej z góry interpretacji. Operatory jednak dokonują interpretacji swoich argumentów. Na przykład operator porównania może interpretować swoje argumenty jako zapisane w notacji dwójkowej liczby całkowite ze znakiem lub bez znaku. W celu rozróżnienia typów argumentów w takich przypadkach, zastosowana przez nas notacja arytmetyki komputerowej wykorzystuje odpowiednią symbolikę.

Największą różnicą arytmetyki komputerowej w stosunku do arytmetyki matematycznej jest to, że arytmetyka komputerowa dokonuje redukcji wyniku dodawania, odejmowania oraz mnożenia do wartości modulo 2^n, gdzie n jest rozmiarem słowa w architekturze danego procesora. Kolejną różnicą jest większa liczba operacji obsługiwanych w ramach arytmetyki komputerowej. Oprócz czterech podstawowych operacji arytmetycznych, arytmetyka komputerowa obsługuje operacje *iloczynu* i *ilorazu logicznego*, *porównania*, *przesunięcia bitowego* itd.

O ile nie zostanie to określone inaczej, długość słowa ustalamy na 32 bity. Liczby całkowite ze znakiem są reprezentowane w formacie uzupełnienia dwójkowego.

Wyrażenia arytmetyki komputerowej są zapisywane w podobny sposób do wyrażeń arytmetyki matematycznej, z tą różnicą, że zmienne określające zawartość rejestrów procesora są zapisywane pogrubioną czcionką. Taka konwencja jest powszechnie wykorzystywana w algebrze wektorowej. Słowo w arytmetyce komputerowej jest wektorem pojedynczych bitów. Stałe odpowiadające wartościom rejestrów są również zapisywane pogrubioną czcionką. W tym przypadku nie ma analogii do algebry wektorowej,

gdzie jedynym sposobem reprezentacji stałej jest prezentacja składowych wektora. W przypadku, gdy stała określa część instrukcji, jak na przykład wynik instrukcji przesunięcia bitowego, zostanie zapisana czcionką standardową.

W przypadku, gdy argumenty operatora, na przykład dodawania (+), są zapisane pogrubioną czcionką, oznacza to, że mamy na myśli operację sumy w arytmetyce komputerowej (sumy wektorowej). W przypadku, gdy argumenty takiej operacji są zapisane czcionką standardową, mamy na myśli zwykłą operację skalarną. Zapis zmiennej x dokonany czcionką standardową określa wartość zmiennej x, zapisanej czcionką pogrubioną i zinterpretowanej zgodnie z kontekstem (to znaczy ze znakiem lub bez znaku). Na przykład w przypadku, gdy x = **0x80000000** oraz y = **0x80000000**, w interpretacji ze znakiem $x = y = -2^{31}$, $x + y = -2^{32}$ oraz $x + y = 0$. W tym przypadku zapis w notacji szesnastkowej **0x80000000** określa ciąg bitów składający się z jedynki, po której następuje trzydzieści jeden zer.

Bity są liczone od strony prawej do lewej. Prawy, najmniej znaczący bit ma indeks 0. Określenia bity (ang. *bits*), półbajty (ang. *nibbles*), bajty (ang. *bytes*), półsłowa (ang. *halfwords*), słowa (ang. *words*) oraz dwusłowa (*doublewords*) oznaczają ciągi bitów o długościach, odpowiednio, 1, 4, 8, 16, 32 oraz 64 bitów.

Krótkie i proste fragmenty kodu są zapisywane w algebrze komputerowej z wykorzystaniem operatora przypisania (strzałki w lewo) oraz instrukcji `if`. W naszym przypadku algebra komputerowa służy jako niezależny od sprzętu sposób zapisu kodu asemblera.

Dłuższe i bardziej skomplikowane programy będą prezentowane w języku programowania C++. Nie będą wykorzystane żadne techniki programowania obiektowego, programy te stanowią po prostu kod zgodny ze standardem ISO C, z komentarzami w stylu języka C++. W każdym przypadku, gdy różnica nie ma większego znaczenia, język ten będzie nazywany po prostu C.

Kompletny opis języka C wykracza poza zakres tematyczny tej książki, lecz tabela 1.1 zawiera krótkie podsumowanie najważniejszych elementów języka C [H&S], które będziemy wykorzystywać w tej książce. Tabela ta może być pomocna dla Czytelników posiadających doświadczenie z językiem programowania proceduralnego innym niż C. Tabela 1.1 zawiera również notację operatorów w zastosowanej w tej książce algebrze komputerowej. Operatory w tej tabeli zostały wyliczone w kolejności od najwyższego do najniższego priorytetu. Dodatkowy symbol **L** w kolumnie *Priorytet* oznacza łączenie lewostronne, na przykład:

```
a • b • c = (a • b) • c
```

Symbol **R** analogicznie określa łączenie prawostronne. Zastosowana notacja algebry komputerowej jest zgodna z językiem C pod względem priorytetów i zasad łączenia operatorów.

Oprócz operacji przedstawionych w tabeli 1.1 są wykorzystywane operacje algebry boolowskiej oraz standardowe operacje matematyczne.

Tabela 1.1. *Wyrażenia w C i odpowiadające im wyrażenia algebry komputerowej*

Priorytet	C	Algebra komputerowa	Opis
	`0x...`	**0x..., 0b...**	Stałe w notacji szesnastkowej i dwójkowej
16	`a[k]`		Wybór elementu o indeksie k
16		$x_0, x_1, \ldots$	Różne zmienne lub wybór bitów (określone w opisie)
16	`f(x,...)`	$f(x, \ldots)$	Wywołanie funkcji
16		$\mathrm{abs}(x)$	Wartość bezwzględna (ale: $\mathrm{abs}(-2^{31}) = -2^{31}$)
16		$\mathrm{nabs}(x)$	Wartość bezwzględna z ujemnym znakiem
15	`x++, x--`		Zwiększenie lub zmniejszenie wartości po wyliczeniu wyrażenia
14	`++x, --x`		Zwiększenie lub zmniejszenie wartości przed wyliczeniem wyrażenia
14	`(`*nazwa typu*`)x`		Przekształcenie typu
14R		x^k	x do potęgi k
14	`~x`	$\neg x, \overline{x}$	Odwrócenie bitowe (uzupełnienie jedynkowe)
14	`!x`		Logiczne zaprzeczenie (zwraca **1**, jeśli x jest równe zero, w przeciwnym wypadku zwraca **0**)
14	`-x`	$-x$	Arytmetyczna zmiana znaku
13L	`x*y`	$x * y$	Mnożenie, modulo rozmiar słowa
13L	`x/y`	$x \div y$	Dzielenie ze znakiem
13L	`x/y`	$x \overset{u}{\div} y$	Dzielenie bez znaku
13L	`x%y`	$\mathrm{rem}(x,y)$	Reszta z dzielenia (może być ujemna) liczb x i y ze znakiem
13L	`x%y`	$\mathrm{remu}(x,y)$	Reszta z dzielenia $x \overset{u}{\div} y$ liczb x i y bez znaku
		$\mathrm{mod}(x,y)$	Redukcja zmiennej x modulo y w przedziale $[0, \mathrm{abs}(y)-1]$. Argumenty ze znakiem
12L	`x + y, x - y`	$x + y, x - y$	Dodawanie, odejmowanie
11L	`x << y, x >> y`	$x << y, x \overset{u}{>>} y$	Przesunięcie w lewo i w prawo, wypełniane zerami (przesunięcie logiczne)
11L	`x >> y`	$x \overset{s}{>>} y$	Przesunięcie w prawo z wypełnieniem uzależnionym od znaku (przesunięcie arytmetyczne lub algebraiczne)
11L		$x \overset{rot}{<<} y, x \overset{rot}{>>} y$	Cykliczne przesunięcie w lewo i prawo
10L	`x < y, x <= y,`	$x < y, x \le y,$	Porównania ze znakiem
	`x > y, x >= y`	$x > y, x \ge y$	
10L	`x < y, x <= y,`	$x \overset{u}{<} y, x \overset{u}{\le} y,$	Porównania bez znaku
	`x > y, x >= y`	$x \overset{u}{>} y, x \overset{u}{\ge} y,$	
9L	`x == y, x != y`	$x = y, x \ne y$	Równość, nierówność

Tabela 1.1. *Wyrażenia w C i odpowiadające im wyrażenia algebry komputerowej (ciąg dalszy)*

Priorytet	C	Algebra komputerowa	Opis
8L	`x & y`	$x \& y$	Koniunkcja bitowa
7L	`x ^ y`	$x \oplus y$	Bitowa różnica symetryczna
7L		$x \equiv y$	Równoważność bitowa ($\neg(x \oplus y)$)
6L	`x | y`	$x \mid y$	Alternatywa bitowa
5L	`x && y`	$x \overset{\rightarrow}{\&} y$	Koniunkcja warunkowa (if $x = 0$ then **0** else if $y = 0$ then **0** else **1**)
4L	`x || y`	$x \overset{\rightarrow}{\mid} y$	Alternatywa warunkowa (if $x \neq 0$ then **0** else if $y \neq 0$ then **0** else **1**)
3L		$x \parallel y$	Łączenie
2R	`x = y`	$x \leftarrow y$	Przypisanie

Nasza algebra komputerowa wykorzystuje również inne funkcje, takie jak „abs", „rem" itd. Zostaną one omówione w miejscu wprowadzenia.

W języku C wyrażenie `x < y < z` spowoduje wyliczenie wartości porównania `x < y` (0 lub 1) a następnie porównania tej wartości z wartością zmiennej *z*. W algebrze komputerowej wyrażenie $x < y < z$ jest równoważne $(x < y)$ & $(y < z)$.

Język C obsługuje trzy instrukcje kontroli przebiegu pętli: `while`, `do` oraz `for`. Instrukcja `while` jest zapisywana następująco:

```
while (wyrażenie) instrukcja
```

Najpierw jest wyliczana wartość wyrażenia kontrolnego (w nawiasach). Jeśli wartość tego wyrażenia jest niezerowa, zostanie wykonana *instrukcja* a następnie kontrola przebiegu programu zostanie przeniesiona do ponownego wyliczenia wyrażenia kontrolnego. W momencie, gdy *wyrażenie* przyjmie wartość zerową, wykonywanie pętli zostanie zakończone.

Instrukcja `do` ma podobne działanie. Zapisuje się ją następująco:

```
do instrukcja while (wyrażenie)
```

W tym przypadku najpierw jest wykonywana *instrukcja*, następnie wyliczane jest *wyrażenie*. Jeśli ma ono wartość niezerową, *instrukcja* zostanie wykonana ponownie, w przeciwnym wypadku wykonywanie pętli zostanie zakończone.

Instrukcja `for` jest zapisywana następująco:

```
for (e1; e2; e3) instrukcja
```

W pierwszej kolejności jest wykonywane wyrażenie e_1, najczęściej jest to instrukcja przypisania. Następnie wykonujemy e_2, które najczęściej jest wyrażeniem porównania. Jeśli wartość e_2 jest równoważna wartości **fałsz**, wykonanie pętli zostanie zakończone, w przeciwnym wypadku pętla zostanie wykonana. Wyrażenie e_3 jest najczęściej

przypisaniem i jest wykonywane po zakończeniu działania pętli, po czym program przechodzi do wyliczenia wartości e_1. Wykorzystując instrukcję `for` możemy zapisać powszechnie stosowany algorytm: „wykonaj instrukcje od $i = 1$ do n":

```
for (i = 1; i <= n; i++)
```

Przy okazji, jest to jeden z przypadków powszechnego wykorzystania operatora zwiększenia wartości zmiennej po wyliczeniu wyrażenia.

1.2. Zestaw instrukcji i model wykonawczy

W celach porównawczych zakładamy, że prezentowane algorytmy są wykonywane na maszynach wykorzystujących zestawy instrukcji zbliżone do występujących we współczesnych komputerach RISC, takich jak Compaq Alpha, SGI MIPS czy też IBM RS/6000. Maszyna jest trójadresowa i posiada dużą liczbę (16 lub więcej) rejestrów ogólnego przeznaczenia. O ile nie zostanie to określone inaczej, zakładamy, że rejestry mają rozmiar 16 bitów. Rejestr ogólnego przeznaczenia o symbolu 0 zawiera wartość zero, pozostałe mogą być wykorzystywane do dowolnych celów.

Dla uproszczenia zakładamy brak występowania rejestrów specjalnych, takich jak rejestry warunkowe przechowujące bity stanu, na przykład informację o przekroczeniu zakresu (ang. *overflow*). Nie wykorzystujemy również operacji zmiennoprzecinkowych, ponieważ zagadnienia tego typu wykraczają poza tematykę niniejszej książki.

Rozróżniamy dwa rodzaje procesorów RISC: „podstawowy RISC", obsługujący zestaw instrukcji zebranych w tabeli 1.2, oraz „pełny RISC" — obsługujący wszystkie instrukcje zestawu podstawowego oraz wymienione w tabeli 1.3.

W powyższych opisach symbole `RA` i `RB`, określające operandy źródłowe, powinny być interpretowane jako wartości tych rejestrów.

Prawdziwy procesor posiada również instrukcję rozgałęzienia do wywołań podprocedur, rozgałęzienia do adresu określonego wartością rejestru (wykorzystywane w powrotach z podprocedur oraz instrukcjach przełączających, tzw. *switch*) oraz kilka instrukcji obsługujących rejestry specjalnego przeznaczenia. Oczywiście powinny również istnieć instrukcje uprzywilejowane oraz instrukcje obsługujące usługi zarządcze (*supervisor services*). Nie może również zabraknąć instrukcji operacji zmiennoprzecinkowych.

Niektóre z instrukcji obliczeniowych, które mogą być obsługiwane przez procesory RISC zebraliśmy w tabeli 1.3. Zostaną one omówione bliżej w kolejnych rozdziałach.

Często stosowanym ułatwieniem jest wyposażenie języka maszynowego w kilka „rozszerzonych mnemoników". Działają one w charakterze makr, które są przekształcane na prawdziwe instrukcje, najczęściej pojedyncze. Niektóre z takich możliwości prezentujemy w tabeli 1.4.

Tabela 1.2. *Podstawowy zestaw instrukcji RISC*

Mnemonik	Operandy	Opis
add, sub, mul, div, divu, rem, remu	RT, RA, RB	RT←RA op RB, gdzie op określa, odpowiednio, *dodawanie*, *odejmowanie*, *mnożenie*, *dzielenie ze znakiem*, *dzielenie bez znaku*, *reszta dzielenia ze znakiem*, *reszta dzielenia bez znaku*
addi, muli	RT, RA, I	RT←RA op I, gdzie op określa operację *dodawania* lub *mnożenia*, natomiast I jest 16-bitową wartością bezpośrednią ze znakiem
addis	RT, RA, I	RT←RA + (I || 0x0000)
and, or, xor	RT, RA, RB	RT←RA op RB, gdzie op określa bitową operację *koniunkcji*, *alternatywy* lub *różnicy symetrycznej*
andi, ori, xori	RT, RA, Iu	Jw., z tym, że drugi operand jest 16-bitową bezpośrednią wartością bez znaku
beq, bne, blt, ble, bgt, bge	RT, cel	Cel rozgałęzienia, odpowiednio, dla RT = 0, RT ≠ 0, RT < 0, RT ≤ 0, RT > 0 lub RT ≥ 0 (RT jest interpretowane ze znakiem)
bt, bf	RT, cel	Cel rozgałęzienia dla wartości **prawda/fałsz**. Działanie identyczne, odpowiednio, z bne, beq
cmpeq, cmpne, cmplt, cmple, cmpgt, cmpge, cmpltu, cmpleu, cmpgtu, cmpgeu	RT, RA, RB	W rejestrze RT zostaje zapisana wartość porównania wartości rejestrów RA i RB: 0 dla **fałszu**, 1 dla **prawdy**. Symbole określają odpowiednio: równość, nierówność, mniejsze niż, większe niż itd., jak w przypadku instrukcji rozgałęzienia. Przyrostek *u* określa porównanie bez znaku.
cmpieq, cmpine, cmpilt, cmpile, cmpigt, cmpige	RT, RA, I	Działanie podobne do operacji cmpeq i pokrewnych, z tą różnicą, że drugi z operandów jest 16-bitową wartością bezpośrednią ze znakiem
cmpiequ, cmpineu, cmpiltu, cmpileu, cmpigtu, cmpigeu	RT, RA, Iu	Działanie podobne do operacji cmpeq i pokrewnych, z tą różnicą, że drugi z operandów jest 16-bitową wartością bezpośrednią bez znaku
ldbu, ldh, ldhu, ldw	RT, d(RA)	Załaduj do rejestru RT bajt bez znaku, półsłowo ze znakiem, półsłowo bez znaku lub słowo z pamięci o adresie RA + d, gdzie d jest 16-bitową wartością bezpośrednią ze znakiem
mulhs, mulhu	RT, RA, RB	Zapisz do rejestru RT 32-bitowy iloczyn wartości rejestrów RA i RB, ze znakiem lub bez znaku
not	RT, RA	RT←wynik uzupełnienia jedynkowego wartości rejestru RA
shl, shr, shrs	RT, RA, RB	RT← wartość RA przesuniętą o RB miejsc w lewo lub prawo. Miejsca z przeciwnej strony przesunięcia są uzupełniane zerami, za wyjątkiem operacji shrs, w przypadku której uzupełnienie jest uzależnione od znaku. Wartość przesunięcia określana jest modulo 64
shli, shri, shrsi	RT, RA, Iu	RT← wartość RA przesuniętą o liczbę miejsc określoną 5-bitową wartością bezpośrednią bez znaku
stb, sth, stw	RS, d(RA)	Zapisz wartość rejestru RT w pamięci o adresie RA + d, gdzie d jest 16-bitową wartością bezpośrednią ze znakiem

Tabela 1.3. *Dodatkowe instrukcje architektury RISC*

Mnemonik	Operandy	Opis
abs, nabs	RT, RA	RT← wartość bezwzględna lub wartość bezwzględna z ujemnym znakiem rejestru RA
andc, eqv, nand, nor, orc	RT, RA, RB	*Bitowa koniunkcja z uzupełnieniem, równoważność, zaprzeczenie koniunkcji, zaprzeczenie alternatywy* lub *alternatywa z uzupełnieniem*
extr	RT, RA, I, L	Wydobycie bitów z zakresu I, I + L - 1 z rejestru RA i umieszczenie ich w RT z wyrównaniem do prawej strony i wypełnieniem zerami
extrs	RT, RA, I, L	Podobnie do extr, lecz z wypełnieniem zależnym od znaku
ins	RT, RA, I, L	Umieszczenie bitów z zakresu 0, L - 1 z RA do RT od miejsca I do I + L - 1
nlz	RT, RA	RT← liczba wiodących zer w RA (0 – 32)
pop	RT, RA	RT← liczba jedynek w RA (0 – 32)
ldb	RT, d(RA)	Załadowanie bajtu ze znakiem z adresu RA + d, gdzie d jest 16-bitową liczbą bezpośrednią ze znakiem
moveq, movne, movlt, movle, movgt, movge	RT, RA, RB	RT←RB jeśli RA = 0, lub RA ≠ 0 itd. W przeciwnym wypadku RT pozostaje bez zmian
shlr, shrr	RT, RA, RB	RT←RA przesunięte z obrotem w lewo lub w prawo o liczbę miejsc przekazaną w 5 najmniej znaczących bitach RB
shlri, shrri	RT, RA, Iu	RT←RA przesunięte z obrotem w lewo lub w prawo o liczbę miejsc przekazaną w 5-bitowej liczbie bezpośredniej
trpeq, trpne, trplt, trple, trpgt, trpge, trpltu, trpleu, trpgtu, trpgeu	RA, RB	Przerwanie (pułapka, ang. *trap*) jeśli RA = RB lub RA ≠ RB itd.
trpieq, trpine, trpilt, trpile, trpigt, trpige,	RT, I	Podobnie do instrukcji trpeq itp. z tą różnicą, że drugi operand jest 16-bitową liczbą bezpośrednią ze znakiem
trpiequ, trpineu, trpiltu, trpileu, trpigtu, trpigeu.	RT, Iu	Podobnie do instrukcji trpltu itp. z tą różnicą, że drugi operand jest 16-bitową liczbą bezpośrednią bez znaku

Tabela 1.4. *Rozszerzone mnemoniki*

Rozszerzony mnemonik	Rozwinięcie	Opis
b cel	beq R0,cel	Rozgałęzienie bezwarunkowe
li RT,I	patrz opis	Załadowanie wartości bezpośredniej. $-2^{31} \le I \le 2^{32}$
mov RT,RA	ori RT,RA,0	Przenieś wartość rejestru RA do rejestru RT
neg RT,RA	sub RT,R0,RA	Negacja (uzupełnienie dwójkowe)
subi RT,RA,I	addi RT,RA,-I	Odejmowanie wartości bezpośredniej ($I \neq -2^{15}$)

Instrukcja ładowania wartości bezpośredniej jest rozwijana do jednej lub dwóch instrukcji, w zależności od zastosowanej wartości bezpośredniej. Na przykład jeśli $0 \leq I < 2^{16}$, może zostać użyta instrukcja `ori` z rejestrem `R0`. Jeśli natomiast $-2^{15} \leq I < 0$, może zostać użyta instrukcja `addi` z rejestrem `R0`. Jeśli wartość `i` ma wyzerowane 16 najmniej znaczących bitów, można użyć instrukcji `addis`. W przeciwnym wypadku musimy zastosować dwie instrukcje, jak na przykład `addis` oraz `ori`. Innym rozwiązaniem może być załadowanie wartości z pamięci, lecz w celu ujednolicenia oszacowań czasu wykonania i rozmiaru kodu zakładamy, że w tym przypadku są wykorzystywane dwie instrukcje składowe.

Oczywiście podział instrukcji RISC na podstawowe oraz uzupełniające do pełnego zestawu jest kwestią dość subiektywnej oceny. Być może instrukcje *dzielenie bez znaku* oraz instrukcje *obliczania reszty z dzielenia* powinny trafić do grupy instrukcji uzupełniających do pełnego RISC. Instrukcja *przesunięcia w prawo ze znakiem* jest również dość kontrowersyjna, ponieważ jej wykorzystanie w testach wydajności SPEC jest sporadyczne. Problem jednak w tym, że programując w C dość łatwo jest sprowokować użycie tych instrukcji, wykonując dzielenie operandów bez znaku, które mogą mieć określony znak. Innym sposobem na sprowokowanie użycia jednej z tych instrukcji jest wykonanie przesunięcia w prawo operandu ze znakiem (`int`), który mógłby równie dobrze nie mieć znaku. Warto zaznaczyć w tym miejscu, że operacja *przesunięcia w prawo ze znakiem* (lub też *arytmetyczne przesunięcie w prawo*) nie jest po prostu dzieleniem liczby ze znakiem przez potęgę dwójki. W celu dokonania takiego dzielenia należy do wyniku dodać 1, jeśli dzielna jest ujemna i w wyniku przesunięcia zostają utracone jakiekolwiek niezerowe bity.

Rozróżnienie pomiędzy zestawem podstawowym a zestawem uzupełniającym do pełnego RISC obejmuje wiele podobnych, dość kontrowersyjnych kryteriów, których jednak nie będziemy rozpatrywać.

Instrukcje ograniczają się do dwóch rejestrów źródłowych i jednego rejestru docelowego, co upraszcza konstrukcję procesora (na przykład układ rejestrów wymaga nie więcej niż dwóch portów odczytu i jednego portu zapisu). Zasada ta upraszcza również kompilatory optymalizujące, ponieważ kompilator nie musi zajmować się obsługą przypadków, gdy wykonanie instrukcji wymaga zapisu wyniku do kilku rejestrów. Koszt takiego rozwiązania polega na tym, że program wymagający jednoczesnego obliczenia wyniku dzielenia oraz reszty z dzielenia (operacja wykorzystywana dość powszechnie) musi wykonać dwie instrukcje (*dzielenia* i *obliczania reszty z dzielenia*). Powszechnie stosowany algorytm obliczający wynik dzielenia jako produkt uboczny wylicza również resztę, dlatego niektóre architektury procesorów wykorzystują ten fakt wyliczając w jednej instrukcji dzielenia również resztę z tego działania. Podobne uwagi dotyczą również obliczania iloczynu dwóch słów, którego wynik ma rozmiar dwusłowa.

Instrukcje warunkowego przeniesienia wartości (ang. *conditional move*, np. `moveq`) na pierwszy rzut oka wydają się posiadać dwa operandy źródłowe, lecz pod pewnym względem posiadają trzy. Wynik jest uzależniony od wartości rejestrów `RT`, `RA` oraz `RB`, zatem procesor wykorzystujący mechanizmy wykonania równoległego musi traktować rejestr `RT` w trakcie wykonania tych instrukcji jako odczytywany i zapisywany. Oznacza

to, że jeśli wykonujemy instrukcję zapisującą w rejestrze RT a następnie instrukcję warunkowego przeniesienia wartości, która również zapisuje wynik w RT, instrukcje te muszą być wykonane jedna po drugiej, to znaczy wartość zapisana w wyniku pierwszej instrukcji nie może zostać pominięta. Z tego powodu projektant takiego procesora mógłby zdecydować się na pominięcie implementacji instrukcji warunkowego przeniesienia wartości w celu uniknięcia konieczności obsługi operacji z trzema operandami źródłowymi. Z drugiej strony operacje warunkowego przeniesienia wartości często pozwalają na uniknięcie rozgałęzienia.

Formaty instrukcji nie mają merytorycznego znaczenia w niniejszej książce, lecz warto zauważyć, że pełny zestaw instrukcji RISC omówiony powyżej, wraz z pominiętymi instrukcjami zmiennoprzecinkowymi oraz kilkoma instrukcjami nadzorczymi (ang. *supervisory*) dają się zaimplementować za pomocą 32-bitowych instrukcji w procesorze wykorzystującym 32 rejestry ogólnego przeznaczenia (z 5-bitowymi polami rejestrów — ang. *register fields*). Redukując do 14 bitów pola instrukcji *porównania*, *ładowania* oraz *zapisywania* wartości bezpośrednich, jak również instrukcji *pułapek* (ang. *trap*), na 32 bitach możemy również zaimplementować procesory wykorzystujące 64 rejestry ogólnego przeznaczenia (6-bitowe pola rejestrów).

Czas wykonania

Zakładamy, że wszystkie instrukcje są wykonywane w pojedynczym cyklu zegara, za wyjątkiem operacji *mnożenia*, *dzielenia* oraz *obliczania reszty z dzielenia*, dla których nie zakładamy żadnego określonego czasu wykonania. Rozgałęzienia zajmują pojedynczy cykl zegara niezależnie od tego, czy są wykonywane, czy pomijane.

Instrukcja *ładowania wartości bezpośredniej* jest szacowana jako jeden lub dwa cykle zegara w zależności od tego, czy w celu wyliczenia wartości rejestru jest potrzebna jedna, czy dwie operacje elementarne.

Pomimo to, że instrukcje *ładowania* i *zapisywania* wartości nie są często wykorzystywane w niniejszej książce, zakładamy, że zajmują one jeden cykl zegara i ignorujemy opóźnienia związane z dostępem (czas, który musi upłynąć pomiędzy zakończeniem operacji ładowania w jednostce arytmetycznej a udostępnieniem tej wartości do wykorzystania przez kolejną instrukcję).

Należy zdawać sobie sprawę z faktu, iż znajomość dokładnych czasów wykonania poszczególnych instrukcji arytmetycznych i logicznych to za mało, aby dokładnie oszacować czas wykonania programu. Wykonanie może być znacznie spowolnione wskutek opóźnień związanych z ładowaniem danych i pobieraniem instrukcji. Opóźnienia tego typu nie są obiektem rozważań niniejszej książki, choć należy uznać, że są bardzo ważnym czynnikiem wpływającym na wydajność pracy komputerów. Kolejnym czynnikiem, tym razem poprawiającym statystyki czasów wykonania programów, są mechanizmy zwane ogólnie „równoległym przetwarzaniem instrukcji". Mechanizmy tego typu można znaleźć w wielu współczesnych procesorach klasy RISC, szczególnie w maszynach klasy *high-end*.

Maszyny tego typu posiadają kilka jednostek wykonawczych jednego typu oraz zaimplementowane możliwości analizy instrukcji i przyporządkowania ich do równoległego wykonania innych instrukcji, z którymi nie mają bezpośredniego związku (to znaczy żadna z równolegle wykonywanych instrukcji nie wykorzystuje wyniku pozostałych i nie zapisuje tego samego rejestru lub bitu stanu, co którakolwiek z pozostałych instrukcji). Występowanie w omawianych algorytmach instrukcji o charakterze niezależnym od pozostałych jest często zaznaczane w opisach, ponieważ wspomniana możliwość procesorów staje się coraz bardziej powszechna. Możemy zatem powiedzieć, że jakaś formuła może zostać zakodowana z wykorzystaniem ośmiu instrukcji i wykonuje się w pięciu cyklach na maszynie udostępniającej nieograniczone możliwości równoległego wykonywania instrukcji. Oznacza to, że jeśli instrukcje są zaplanowane (ang. *scheduled*) w określonym porządku a maszyna posiada odpowiednią liczbę jednostek sumujących (ang. *adders*), przesuwających (ang. *shifters*), logicznych (ang. *logical units*) oraz odpowiednią liczbę rejestrów, może wykonać osiem instrukcji w pięciu cyklach maszynowych.

Nie zakładamy jednak takich możliwości jako powszechnych, głownie dlatego, że wiele maszyn znacznie różni się pod względem możliwości równoległego wykonywania instrukcji. Na przykład procesor IBM RS/6000 z roku 1992 posiada jednostkę sumującą z trzema wejściami, która jest w stanie wykonywać jednocześnie dwie instrukcje *dodawania*, nawet jeśli jedna z nich dostarcza dane drugiej (np. instrukcja *dodawania* dostarcza dane instrukcji *porównania* lub wylicza wartość rejestru adresowego operacji *ładowania* danych z pamięci). Jako przykład z przeciwnego bieguna może posłużyć prosty komputer, być może niedrogie urządzenie osadzone (ang. *embedded*), posiadające tylko jeden port odczytu w obszarze rejestrów. Urządzenie to zwykle potrzebuje dodatkowego cyklu w celu dokonania odczytu wartości drugiego rejestru wykorzystywanego w instrukcji z dwoma operandami źródłowymi. Możemy jednak założyć, że w procesorze tym istnieje mechanizm, pozwalający na uniknięcie operacji odczytu z obszaru rejestrów w przypadku, gdy wcześniejsza instrukcja wylicza jeden z operandów instrukcji wykonywanej bezpośrednio po niej. Na takiej maszynie bardzo pożądane jest takie szeregowanie instrukcji, aby wyliczanie wartości rejestrów następowało bezpośrednio przed ich wykorzystaniem w kolejnych instrukcjach — oczywiście pod warunkiem, że kod nie wykorzystuje mechanizmu równoległego wykonania instrukcji.

Rozdział 2.

Podstawy

2.1. Manipulowanie prawostronnymi bitami

Niektóre z formuł przedstawionych w niniejszym rozdziale będą stosowane w kolejnych rozdziałach.

W celu zastąpienia zerem najmniej znaczącej (prawostronnej — ang. *rightmost*) jedynki w słowie można zastosować następującą formułę (0101 1000 ⇒ 0101 0000):

$$x \,\&\, (x - 1)$$

Metoda ta może posłużyć do sprawdzenia, czy wartość liczby całkowitej bez znaku jest równa potędze liczby 2. Po wyliczeniu wartości formuły jej wynik powinien być równy zero.

Kolejna formuła może zostać zastosowana w celu sprawdzenia, czy dana liczba jest postaci $2^n - 1$ (włączając 0 oraz ciąg samych jedynek):

$$x \,\&\, (x + 1)$$

W celu wyodrębnienia skrajnej jedynki w słowie można zastosować następującą formułę (0101 1000 ⇒ 0000 1000):

$$x \,\&\, (-x)$$

W celu wyodrębnienia skrajnego zera w słowie można zastosować następującą formułę (0101 0111 ⇒ 0000 1000):

$$\neg x \,\&\, (x + 1)$$

Jedna z poniższych metod może posłużyć do zidentyfikowania skrajnych zer. W przypadku, gdy argument ma wartość 0, zostanie wygenerowany ciąg złożony z samych jedynek, gdy $x = 0$ (np. 01011000 ⇒ 00000111):

$$\neg x \,\&\, (x - 1)$$

lub:

$$\neg(x \mid -x)$$

lub:

$$(x \,\&\, -x) - 1$$

Pierwsza z powyższych formuł umożliwia zastosowanie przetwarzania równoległego.

W celu wygenerowania maski identyfikującej najmniej znaczącą jedynkę i skrajne zera dla $x = 0$ można zastosować następującą formułę (0101 1000 ⇒ 0000 1111):

$$x \oplus (x - 1)$$

W celu zastąpienia skrajnych zer jedynkami w słowie można zastosować następującą formułę (0101 1000 ⇒ 0101 1111):

$$x \mid (x - 1)$$

W celu wyłączenia w słowie ciągłego bloku najmniej znaczących jedynek można zastosować następującą formułę (0101 1000 ⇒ 0100 0000):

$$((x \mid (x - 1)) + 1) \,\&\, x$$

Formuła ta może zostać zastosowana w celu sprawdzenia, czy dodatnia liczba całkowita jest postaci $2^j - 2^k$ dla $j \geq k \geq 0$. Po wyliczeniu wartości formuły wynik powinien być równy zero.

Wszystkie powyższe formuły mają swoje odpowiedniki. Każdą z nich można przekształcić w formułę dopełniającą, zastępując w opisie jedynki zerami a następnie w formule zastąpić operację $x - 1$ operacją $x + 1$, $x + 1$ zastąpić $x - 1$, $-x$ zastąpić $\neg(x + 1)$, & zastąpić | oraz | zastąpić &. Pojedyncze x oraz $\neg x$ zostawiamy bez zmian. Wynik takich zamian stanowi nowy opis i nową formułę. Na przykład po zastosowaniu powyższych zastąpień w pierwszej formule uzyskamy następującą:

W celu zastąpienia jedynką najmniej znaczącego zera w słowie można zastosować następującą formułę (0101 0111 ⇒ 0101 1111):

$$x \mid (x + 1)$$

Istnieje prosty test pozwalający na określenie, czy dana funkcja może zostać zaimplementowana za pomocą sekwencji *dodawań*, *odejmowań*, operacji *koniunkcji*, *alternatywy* i *negacji* [War]. Możemy oczywiście rozszerzyć tę listę instrukcji o dostępne w podstawowym zestawie instrukcji RISC. Przykładem takiej instrukcji może być operacja *przesunięcia bitowego w lewo* o ustaloną liczbę miejsc (operacja ta jest równoważna sekwencji operacji *dodawania*) lub operacja *mnożenia*. Wykluczamy jednak te instrukcje, których nie można zasymulować za pomocą instrukcji z listy podstawowej. Test, o którym wspomnieliśmy jest opisany w następującym twierdzeniu:

twierdzenie

Funkcja przekształcająca słowa może być zaimplementowana za pomocą instrukcji *dodawania*, *odejmowania*, *koniunkcji*, *alternatywy* i *negacji* wtedy i tylko wtedy, gdy każdy z bitów wyniku jest uzależniony tylko od bitów znajdujących się na jego pozycji oraz z prawej strony we wszystkich operandach wejściowych.

Oznacza to, że próbujemy wyliczyć najmniej znaczący bit wyniku opierając się wyłącznie na wartościach najmniej znaczących bitów wszystkich operandów wejściowych. Następnie próbujemy wyliczyć kolejny bit położony po lewej stronie, opierając się na wartościach dwóch najmniej znaczących bitów każdego z operandów wejściowych itd. Jeśli opisane obliczenia powiodą się, funkcja może zostać opisana za pomocą wymienionych operacji podstawowych. Jeśli wyniku funkcji nie da się wyliczyć w ten sposób, nie da się jej zaimplementować wyłącznie za pomocą instrukcji podstawowych.

Z powyższego twierdzenia można wysnuć interesujący wniosek: instrukcje *dodawania*, *odejmowania*, *koniunkcji*, *alternatywy* oraz *negacji* mogą być wyliczane metodą od prawej do lewej, zatem wszelkie ich kombinacje również cechuje ta sama własność.

Aby opisać warunek twierdzenia, potrzebna jest konstrukcja dość trudna do opisania. Posłużmy się zatem odpowiednim przykładem. Załóżmy, że funkcja dwóch zmiennych x oraz y spełnia własność wyliczania od prawej strony do lewej. Załóżmy również, że bit wyniku o indeksie 2 można obliczyć za pomocą następującego wyrażenia:

$$r_2 = x_2 \mid (x_0 \,\&\, y_1) \tag{1}$$

Bity liczymy od prawej do lewej, zaczynając od zera, kończąc na 31. Wartość bitu o indeksie 2 w wyniku można wyliczyć na podstawie bitów operandów wejściowych od pozycji bitu 2 w prawo, zatem funkcja obliczająca bit 2 spełnia wymóg twierdzenia.

Ułóżmy kolejno słowo x, x przesunięte w lewo o dwa, y przesunięte w lewo o jeden jak w poniższym wzorcu. Do wzorca dodajmy również maskę izolującą bit 2:

$$\begin{array}{ccccccc}
x_{31} & x_{30} & \dots & x_3 & x_2 & x_1 & x_0 \\
x_{29} & x_{28} & \dots & x_1 & x_0 & 0 & 0 \\
y_{30} & y_{29} & \dots & y_2 & y_1 & y_0 & 0 \\
0 & 0 & \dots & 0 & 1 & 0 & 0 \\
0 & 0 & \dots & 0 & r_2 & 0 & 0
\end{array}$$

Następnie wyliczamy wynik *koniunkcji* bitowej wiersza 2. i 3., którego wynik poddajemy operacji *alternatywy* bitowej z wierszem 1. (zgodnie z równaniem (1)), który to wynik z kolei poddajemy operacji *koniunkcji* z maską (wiersz 4.). W wyniku uzyskujemy wiersz składający się z zer oraz poszukiwanej wartości bitu na pozycji 2. Podobne obliczenia wykonujemy dla pozostałych bitów wyniku i wszystkie 32 wyniki składamy razem za pomocą instrukcji *alternatywy* bitowej. Uzyskany wynik będzie wynikiem naszej funkcji.

Powyższy przykład nie stanowi optymalnego programu. Jest jednak przykładem zastosowania wyłącznie instrukcji podstawowych.

Analizując powyższe twierdzenie widzimy, że nie istnieje sekwencja instrukcji podstawowych, która pozwoliłaby zastąpić zerem najbardziej znaczącą jedynkę w słowie, ponieważ w celu sprawdzenia, czy dana jedynka jest najbardziej znacząca musimy sprawdzić bity znajdujące się na lewo od tego miejsca (co jest w sprzeczności z twierdzeniem). Podobnie nie istnieje sekwencja instrukcji podstawowych wykonująca przesunięcie w prawo lub też przesunięcie cykliczne (w dowolnym kierunku), ani przesunięcie w lewo o zmienną liczbę pozycji, ani obliczająca liczbę najmniej znaczących zer w słowie (w przypadku obliczania liczby takich zer najmniej znaczący bit wyniku będzie miał wartość 1, jeżeli istnieje nieparzysta ich liczba i w celu stwierdzenia tego faktu należy sprawdzić bity znajdujące się na lewo od najmniej znaczącego bitu).

Jedno z ciekawych zastosowań z rodziny zabaw z bitami polega na odnalezieniu następnej liczby zawierającej tę samą co wzorzec liczbę jedynek. W pierwszej chwili trudno domyślić się celu stosowania tego typu mechanizmów. Istnieją jednak algorytmy wykorzystujące ciągi bitów w celu reprezentacji podzbiorów. Możliwe elementy zbioru są odwzorowane w postaci bitów, natomiast podzbiory są reprezentowane w postaci ciągów bitów, w których odpowiedni bit ma wartość 1, jeśli odpowiadający mu element zbioru występuje w podzbiorze. Suma podzbiorów jest obliczana za pomocą *alternatywy* bitowej słów reprezentujących podzbiory, przecięcia zbiorów są obliczane za pomocą *koniunkcji* bitowej itd.

Możliwe jest przeglądanie po kolei wszystkich podzbiorów o zadanym rozmiarze. Można to zrobić w bardzo prosty sposób, wykorzystując funkcję odwzorowującą podzbiór na kolejną liczbę całkowitą (gdzie podzbiór jest interpretowany jako liczba całkowita) o tej samej liczbie jedynek.

Rozsądny algorytm tego typu został wymyślony przez R. W. Gospera [HAK, poz. 175][1]. Na podstawie słowa x, reprezentującego podzbiór, należy znaleźć prawostronny ciąg samych jedynek i następujących po nich zer a następnie zwiększyć tę wartość do następnej wartości o identycznej liczbie jedynek. Na przykład ciąg xxx0 1111 0000, gdzie xxx reprezentuje dowolne bity, zostanie przekształcony na xxx1 0000 0111. W pierwszej kolejności algorytm za pomocą znanej nam już formuły $s = x \,\&\, -x$ wyszukuje w x najmniej znaczącą jedynkę, co daje w wyniku 0000 0001 0000. Wynik ten dodajemy do naszej liczby wejściowej x i otrzymujemy r = xxx1 0000 0000. Jedynka w tym wyniku jest jedną z jedynek wyniku ostatecznego. Pozostałe jedynki tworzymy dodając do r wyrównany do prawej strony ciąg jedynek o długości $n - 1$, gdzie n jest liczbą jedynek w prawostronnym bloku jedynek w x. Można tego dokonać za pomocą operacji *różnicy symetrycznej* r oraz x, co w naszym przykładzie daje wynik 0001 1111 0000.

Wynik posiada za dużo jedynek i należy wyrównać go do prawej strony. Możemy tego dokonać za pomocą dzielenia przez s, co spowoduje przesunięcie w prawo (s jest potęgą dwójki). Następnie dokonujemy przesunięcia wyniku w prawo o dwa, co spowoduje usunięcie dwóch nadmiarowych jedynek. Ostateczny wynik uzyskamy obliczając wynik alternatywy logicznej wyniku ostatniej operacji oraz liczby r.

[1] Nieco inna wersja tego algorytmu jest dostępna w pozycji [H&S], punkt 7.6.7

W zapisie naszej algebry matematycznej powyższe operacje wyglądają następująco:

$$s \leftarrow x \,\&\, -x \tag{2}$$

$$r \leftarrow s + x$$

$$y \leftarrow r \mid (((x \oplus r) \overset{u}{>>} 2) \overset{u}{\div} s)$$

Pełna procedura w języku C została przedstawiona na listingu 2.1. Wykorzystuje ona siedem podstawowych instrukcji RISC, w tym instrukcję *dzielenia*. Nie należy wykorzystywać jej z $x = 0$, ponieważ spowoduje błąd dzielenia przez zero.

Listing 2.1. *Następna liczba większa od x o tej samej liczbie jedynek*

```
unsigned snoob(unsigned x) {
   unsigned smallest, ripple, ones;
                                  // x = xxx0 1111 0000
   smallest = x & -x;             //     0000 0001 0000
   ripple = x + smallest;         //     xxx1 0000 0000
   ones = x ^ ripple;             //     0001 1111 0000
   ones = (ones >> 2)/smallest;   //     0000 0000 0111
   return ripple | ones;          //     xxx1 0000 0111
}
```

Jeśli operacja *dzielenia* jest wolno wykonywana na komputerze o danej architekturze, natomiast mamy dostęp do szybkiej operacji obliczania liczby końcowych zer (ang. *number of trailing zeros*) ntz(x), liczby wiodących zer (ang. *number of leading zeros*) nlz(x) lub też wielkości populacji (ang. *population count*) pop(x), będącej liczbą jedynek w słowie, równanie (2) możemy zastąpić jedną z następujących operacji:

$$y \leftarrow r \mid ((x \oplus r) \overset{u}{>>} (2 + \mathrm{ntz}(x)))$$

$$y \leftarrow r \mid ((x \oplus r) \overset{u}{>>} (33 - \mathrm{nlz}(s)))$$

$$y \leftarrow r \mid ((1 << (\mathrm{pop}(x \oplus r) - 2)) - 1)$$

2.2. Łączenie dodawania z operacjami logicznymi

Zakładamy, że Czytelnik jest zaznajomiony z podstawowymi pojęciami algebry arytmetycznej oraz algebry boolowskiej. Poniżej przedstawiamy wybór operacji obejmujących łączenie operacji *dodawania* i *odejmowania* z operacjami logicznymi:

a. $\quad -x = \neg x + 1$

b. $\quad = \neg(x - 1)$

c. $\quad \neg x = -x - 1$

d. $-\neg x = x + 1$

e. $\neg -x = x - 1$

f. $x + y = x - \neg y - 1$

g. $= (x \oplus y) + 2(x \,\&\, y)$

h. $= (x \mid y) + (x \,\&\, y)$

i. $= 2(x \mid y) - (x \oplus y)$

j. $x - y = x + \neg y + 1$

k. $= (x \oplus y) - 2(\neg x \,\&\, y)$

l. $= (x \,\&\, \neg y) - (\neg x \,\&\, y)$

m. $= 2(x \,\&\, \neg y) - (x \oplus y)$

n. $x \oplus y = (x \mid y) - (x \,\&\, y)$

o. $x \,\&\, \neg y = (x \mid y) - y$

p. $= x - (x \,\&\, y)$

q. $\neg(x - y) = y - x - 1$

r. $= \neg x + y$

s. $x \equiv y = (x \,\&\, y) - (x \mid y) - 1$

t. $= (x \,\&\, y) + \neg(x \mid y)$

u. $x \mid y = (x \,\&\, \neg y) + y$

v. $x \,\&\, y = (\neg x \mid y) - \neg x$

Równanie (d) może być wykonywane sekwencyjnie na samym sobie, co w wyniku daje równanie $- \neg - \neg x = x + 2$ itd. Podobnie z wzoru (e) możemy uzyskać $\neg - \neg - x = x - 2$. Możemy zatem dodawać i odejmować dowolne liczby za pomocą tylko tych dwóch kombinacji.

Równanie (f) jest odpowiednikiem równania (j), (j) natomiast jest znaną relacją pozwalającą na zbudowanie układu odejmującego z sumatora.

Równania (g) oraz (h) pochodzą z oryginalnego artykułu w HAKMEM [HAK, pozycja 23]. Równanie (g) formułuje sumę za pomocą różnicy symetrycznej $(x \oplus y)$, zwanej również sumą bez przeniesienia, z wynikiem operacji wyliczającej przeniesienia. Równanie (h) po prostu modyfikuje operandy dodawania w ten sposób, że operacja 0 + 1 nie występuje na żadnej pozycji bitowej — zamiast tego pojawi się operacja 1 + 0.

Można wykazać, że w operacji *dodawania* liczb binarnych, w których każdy bit jest równy 0 lub 1, na każdej z pozycji może wystąpić sytuacja przeniesienia z prawdopodobieństwem równym 0,5. W przypadku zastosowania mechanizmu sumującego przedstawionego formułą (g), prawdopodobieństwo przeniesienia jest równe 0,25. Obserwacja

ta być może nie ma większego znaczenia na etapie konstrukcji układu sumującego, ponieważ w tym przypadku krytyczne znaczenie ma liczba bramek logicznych, przez które musi przejść sygnał przeniesienia. W formule (g) liczba takich bramek logicznych została zmniejszona tylko o jedną.

Równania (k) oraz (l) są odpowiednikami równań (g) oraz (h), dotyczącymi operacji *odejmowania*. A dokładniej formuła (k) definiuje odejmowanie, rozbijając je na dwa elementy: różnica bez „pożyczek" ($x \oplus y$) a następnie odjęcie samych „pożyczek". Podobnie równanie (l) stanowi prostą modyfikację operandów odejmowania w ten sposób, że kombinacja bitów 1 – 1 nie może wystąpić na żadnej z pozycji, jest ona zastępowana kombinacją 0 – 0.

Równanie (n) demonstruje sposób implementacji różnicy symetrycznej za pomocą trzech instrukcji podstawowego zestawu instrukcji RISC. Wykorzystując jedynie instrukcje *koniunkcji*, *alternatywy* i *negacji* możemy skonstruować różnicę symetryczną w czterech instrukcjach (($x \mid y$) & $\neg(x \,\&\, y)$). Podobnie formuły (u) oraz (v) demonstrują sposób implementacji instrukcji *koniunkcji* oraz *alternatywy* za pomocą trzech innych instrukcji elementarnych, podczas gdy z własności DeMorgana wynika konieczność zastosowania czterech operacji.

2.3. Nierówności w wyrażeniach logicznych i arytmetycznych

Rozstrzyganie nierówności w binarnych wyrażeniach logicznych, których wartości są interpretowane jako liczby całkowite bez znaku jest bardzo łatwe w realizacji. Oto dwa przykłady: $f(x,y) \overset{u}{\le} g(x,y)$

$$(x \oplus y) \overset{u}{\le} (x \mid y) \quad \text{oraz}$$

$$(x \,\&\, y) \overset{u}{\le} (x \equiv y).$$

Powyższe nierówności możemy uzyskać z listy binarnych operacji logicznych, przedstawionych w tabeli 2.1. Załóżmy, że $f(x, y)$ oraz $g(x, y)$ reprezentują dwa wiersze tabeli 2.1. Jeśli dla każdej z kolumn, w której $f(x, y)$ jest równe 1, $g(x, y)$ jest również równe 1, to dla każdego (x, y) jest spełniona nierówność $f(x,y) \overset{u}{\le} g(x,y)$. Oczywiście rozumowanie to można rozszerzyć na równoległe operacje logiczne. Tego rodzaju relacje (które w większości przypadków są banalne) można z łatwością odczytywać jako $(x \,\&\, y) \overset{u}{\le} x \overset{u}{\le} (x \mid \neg y)$ i tak dalej. Ponadto jeśli dwa wiersze posiadają kolumnę, w której jedną wartością jest 0 a drugą 1 oraz drugą kolumnę, w której wartościami są, odpowiednio, 1 i 0, wtedy nie istnieje żadna relacja nierówności pomiędzy odpowiadającymi sobie wyrażeniami logicznymi. Zatem na pytanie, czy jest spełniona nierówność $f(x,y) \overset{u}{\le} g(x,y)$ można jednoznacznie i z łatwością odpowiedzieć w przypadku wszystkich binarnych funkcji logicznych f oraz g.

Podczas wykonywania operacji na tych nierównościach należy zachować ostrożność. Na przykład dla zwykłego dodawania wynika prosta zależność, jeśli $x + y \le a$ oraz $z \le x$, to $z + y \le a$. Zależność taka nie jest jednak spełniona, jeśli operację + zastąpimy operacją logicznej *alternatywy*.

Tabela 2.1. *16 binarnych operacji logicznych*

x	0	0	1	1
y	0	1	0	1
0	0	0	0	0
$x \& y$	0	0	0	1
$x \& \neg y$	0	0	1	0
x	0	0	1	1
$\neg x \& y$	0	1	0	0
y	0	1	0	1
$x \oplus y$	0	1	1	0
$x \mid y$	0	1	1	1
$\neg(x \mid y)$	1	0	0	0
$x \equiv y$	1	0	0	1
$\neg y$	1	0	1	0
$x \mid \neg y$	1	0	1	1
$\neg x$	1	1	0	0
$\neg x \mid y$	1	1	0	1
$\neg(x \& y)$	1	1	1	0
1	1	1	1	1

Bardziej interesujące są nierówności, w których są stosowane naprzemian operacje arytmetyczne i logiczne. Poniżej prezentujemy niewielkie zestawienie takich nierówności:

a. $(x \mid y) \overset{u}{\geq} \max(x, y)$

b. $(x \& y) \overset{u}{\leq} \min(x, y)$

c. $(x \mid y) \overset{u}{\leq} x + y$ jeśli dodawanie nie powoduje przepełnienia

d. $(x \mid y) \overset{u}{>} x + y$ jeśli dodawanie powoduje przepełnienie

e. $|x - y| \overset{u}{\leq} (x \oplus y)$

Dowody powyższych nierówności są raczej proste do przeprowadzenia, za wyjątkiem nierówności $|x - y| \overset{u}{\leq} (x \oplus y)$. Zapis $|x - y|$ oznacza wartość bezwzględną operacji $x - y$, co w dziedzinie liczb bez znaku można zapisać jako $\max(x, y) - \min(x, y)$. Zależność tę można dowieść przez indukcję względem długości liczb x i y (dowód jest łatwiejszy do przeprowadzenia, gdy liczby te zostaną rozszerzone po lewej, a nie po prawej stronie).

2.4. Wartość bezwzględna

Jeśli wykorzystywany procesor nie udostępnia instrukcji obliczania wartości bezwzględnej, można tego dokonać za pomocą trzech lub czterech instrukcji, bez konieczności

stosowania rozgałęzień. W pierwszej kolejności obliczamy $y \leftarrow x \overset{s}{>>} 31$ a następnie wykorzystujemy jedną z następujących procedur:

abs

$$(x \oplus y) - y$$

$$(x + y) \oplus y$$

$$x - (2x \,\&\, y)$$

lub:

nabs

$$y - (x \oplus y)$$

$$(y - x) \oplus y$$

$$(2x \,\&\, y) - x$$

Przez zapis $2x$ rozumiemy oczywiście $x + x$, lub $x << \mathbf{1}$.

Jeśli w procesorze jest dostępna szybka operacja *mnożenia* przez zmienną, której wartość wynosi ±1, do obliczenia wartości bezwzględnej wystarczy następująca operacja:

$$((x \overset{s}{>>} 30) \mid \mathbf{1}) * x$$

2.5. Rozszerzenie o znak

Przez *rozszerzenie o znak* (ang. *sign extension*) rozumiemy taką sytuację, w której jeden z bitów w liczbie traktujemy jako bit znaku i wartość tego bitu powtarzamy w lewo, niezależnie od wartości pozostałych bitów. Standardowym sposobem na realizację tego celu jest wykonanie operacji logicznego przesunięcia w lewo (ang. *shift left logical*), po którym wykonujemy przesunięcie w prawo ze znakiem (ang. *shift right signed*). Jeśli jednak instrukcje te nie są dostępne w wykorzystywanej maszynie lub działają wolno, możemy tego dokonać za pomocą jednej z poniższych operacji. W naszym przykładzie propagujemy w lewo znak na 7. pozycji w słowie:

$$((x + \mathbf{0x00000080}) \,\&\, \mathbf{0x000000FF}) - \mathbf{0x00000080}$$

$$((x \,\&\, \mathbf{0x000000FF}) \oplus \mathbf{0x00000080}) - \mathbf{0x00000080}$$

Znak „+” w powyższych operacjach możemy zastąpić znakiem „–” lub „⊕”. Druga z formuł może być szczególnie użyteczna w przypadku, gdy wiemy, że wszystkie bardziej znaczące bity są zerami, ponieważ w tej sytuacji możemy pominąć operacją *koniunkcji* (&).

2.6. Przesunięcie w prawo ze znakiem za pomocą instrukcji przesunięcia bez znaku

W przypadku, gdy procesor nie obsługuje instrukcji *przesunięcia w prawo ze znakiem* (ang. *shift right signed*), wartość tej operacji można wyliczyć za pomocą przedstawionych niżej formuł. Pierwsza z nich pochodzi z [GM], druga zaś jest modyfikacją tego samego pomysłu. Założywszy, że procesor udostępnia przesunięcia modulo 64, pierwsze cztery formuły dają prawidłowe wyniki dla $0 \le n \le 31$, natomiast ostatnia daje prawidłowe wyniki dla $0 \le n \le 63$. Ostatnia z przedstawionych formuł daje prawidłowe wyniki dla każdej liczby n, to znaczy traktuje rozmiar przesunięcia tak samo, jak ma to miejsce w operacji *przesunięcia* logicznego.

W przypadku, gdy n jest zmienną, każda z poniższych formuł wymaga zastosowania pięciu lub sześciu instrukcji z podstawowego zestawu RISC.

$$((x + \mathbf{0x80000000}) \overset{u}{>>} n) - (\mathbf{0x80000000} \overset{u}{>>} n)$$

$$t \leftarrow \mathbf{0x80000000} \overset{u}{>>} n; \quad ((x \overset{u}{>>} n) \oplus t) - t$$

$$t \leftarrow (x \,\&\, \mathbf{0x80000000} \overset{u}{>>} n; \quad (x \overset{u}{>>} n) - (t + t)$$

$$(x \overset{u}{>>} n) \mid (-(x \overset{u}{>>} 31) << \mathbf{31} - n)$$

$$t \leftarrow -(x \overset{u}{>>} 31); \quad ((x \oplus t) \overset{u}{>>} n) \oplus t$$

W pierwszych dwóch wyrażeniach zapis $\mathbf{0x80000000} \overset{u}{>>} n$ można zastąpić zapisem $1 << \mathbf{31} - n$.

Jeśli n jest stałą, pierwsze dwie formuły na większości maszyn są wykonywane z wykorzystaniem tylko trzech instrukcji. Jeśli $n = 31$, funkcja może zostać wykonana w dwóch instrukcjach, po sprowadzeniu jej do postaci $-(x \overset{u}{>>} 31)$.

2.7. Funkcja signum

Funkcja rozpoznająca znak liczby (ang. *sign* lub *signum*) może zostać zdefiniowana za pomocą następującej formuły:

$$\operatorname{sign}(x) = \begin{cases} -1, & x < 0, \\ 0, & x = 0, \\ 1, & x > 0. \end{cases}$$

Na większości maszyn powyższa formuła może być wyliczona za pomocą czterech instrukcji [Hop]:

$$(x \overset{s}{>>} 31) \mid (-x \overset{u}{>>} 31)$$

Jeśli na maszynie nie jest dostępna instrukcja *przesunięcia w prawo ze znakiem*, można posłużyć się formułą z końca podrozdziału 2.6, co sprowadza nas do poniższej, symetrycznej formuły:

$$-(x \overset{u}{>>} 31) \mid (-x \overset{u}{>>} 31)$$

Instrukcje predykatów porównań umożliwiają rozwiązanie wykorzystujące trzy instrukcje. Można to zrobić na jeden z dwóch sposobów:

$$(x > 0) - (x < 0)$$

lub (3)

$$(x \geq 0) - (x \leq 0)$$

Można również zauważyć, że formuła $(-x \overset{u}{>>} 31) - (x \overset{u}{>>} 31)$ spełnia nasze oczekiwania, za wyjątkiem przypadku $x = -2^{31}$.

2.8. Funkcja porównania trójwartościowego

Funkcja porównania trójwartościowego (ang. *three-valued compare*) stanowi uogólnienie funkcji *signum*. Można ją zdefiniować w następujący sposób:

$$\mathrm{cmp}(x, y) = \begin{cases} -1, & x < y, \\ 0, & x = y, \\ 1, & x > y. \end{cases}$$

Istnieją implementacje powyższej formuły ze znakiem i bez znaku. O ile nie zostanie to określone inaczej, w niniejszym podrozdziale omawiamy obydwa typy.

Instrukcje predykatów porównań umożliwiają implementację tej funkcji za pomocą trzech instrukcji, co stanowi oczywistą generalizację równań (3):

$$(x > y) - (x < y)$$

lub:

$$(x \geq y) - (x \leq y).$$

Poniżej prezentujemy rozwiązanie powyższego problemu na komputerach PowerPC [CWG]. Na tych maszynach przeniesienie oznacza „brak pożyczki".

```
subf  R5,Ry,Rx   # R5 <-- Rx - Ry
subfc R6,Rx,Ry   # R6 <-- Ry - Rx, z przeniesieniem
subfe R7,Ry,Rx   # R7 <-- Rx - Ry + przeniesienie, z przeniesieniem
subfe R8,R7,R5   # R8 <-- R5 - R7 + przeniesienie, z przeniesieniem
```

W przypadku ograniczenia do podstawowego zestawu instrukcji RISC trudno znaleźć dobry sposób obliczenia tej funkcji. Predykaty porównań $x < y$, $x \le y$ itd. wymagają około pięciu instrukcji (zob. podrozdział 2.11), co sprowadza nas do rozwiązania wykorzystującego około 12 instrukcji (wykorzystując możliwość sprowadzenia do jednej instrukcji dwóch porównań $x < y$ oraz $x > y$). W przypadku ograniczenia do podstawowego zestawu instrukcji RISC najprawdopodobniej najlepiej byłoby wykorzystać porównania z instrukcjami rozgałęzień (w najgorszym wypadku sześć instrukcji, jeśli wykorzystamy wspomniane wyżej oszczędności na porównaniach tych samych wartości).

2.9. Przeniesienie znaku

Funkcja *przeniesienia znaku* (ang. *transfer of sign*), w języku Fortran nosząca nazwę ISIGN, jest zdefiniowana następująco:

$$\mathrm{ISIGN}(x, y) = \begin{cases} \mathrm{abs}(x), & y \ge 0, \\ -\mathrm{abs}(x), & y < 0. \end{cases}$$

Funkcję te możemy wyliczyć (modulo 2^{32}) na większości maszyn za pomocą czterech instrukcji:

$$t \leftarrow y \overset{s}{>>} 31;$$

$$\mathrm{ISIGN}(x, y) = (\mathrm{abs}(x) \oplus t) - t = (\mathrm{abs}(x) + t) \oplus t$$

albo

$$t \leftarrow (x \oplus y) \overset{s}{>>} 31;$$

$$\mathrm{ISIGN}(x, y) = (x \oplus t) - t = (x + t) \oplus t$$

2.10. Dekodowanie pola „zero oznacza 2^n"

Czasami zastosowanie wartości 0 lub liczby ujemnej nie ma sensu dla określenia ilości. W takim przypadku koduje się ją w postaci pola bitowego o długości n i wartość 0 interpretuje się jako liczbę 2^n, natomiast wartości niezerowe interpretowane są w standardowy

sposób. Przykładem takiego zastosowania może być pole długości w instrukcji `lswi` (ang. *load string word immediate*) ładowania ciągu słów, mające rozmiar pięciu bitów. Nie ma sensu ładować ciągu słów o długości zero, natomiast z całą pewnością przydałaby się możliwość załadowania ciągu o długości 32 słów. Pole określające długość ciągu może zakodować liczbę od 0 do 31, co można sprowadzić do wartości od 1 do 32, lecz konwencja „zero oznacza 32" powoduje, że implementacja tego typu instrukcji jest znacznie prostsza. Jest to szczególnie korzystny sposób w sytuacji, gdy procesor musi obsługiwać również pokrewną instrukcję o zmiennej długości, która wykorzystuje zwykłe kodowanie binarne (np. instrukcja `lswx` w procesorach PowerPC).

Kodowanie liczby całkowitej w przedziale od 1 do 2^n w znaczeniu „zero oznacza 2^n" jest trywialne — po prostu maskujemy liczbę całkowitą wartością $2^n - 1$. Odkodowanie takiego zapisu bez zastosowania techniki testów i rozgałęzień nie jest już tak proste. Poniżej przedstawiamy kilka sposobów poradzenia sobie z tym problemem dla pola o długości 3 bitów. Każdy z nich wymaga zastosowania trzech instrukcji, pomijając czas na załadowanie wartości stałych:

$$((x - 1) \,\&\, 7) + 1$$

$$((x + 7) \mid -8) + 9$$

$$8 - (-x \,\&\, 7)$$

$$((x + 7) \,\&\, 7) + 1$$

$$((x + 7) \mid 8) - 7$$

$$-(-x \mid -8)$$

$$((x - 1) \mid -8) + 9$$

$$((x - 1) \,\&\, 8) + x$$

2.11. Predykaty porównań

Predykat porównania (ang. *comparison predicate*) jest funkcją porównującą dwie wartości i zwracającą jednobitowy wynik o wartości 1 w przypadku, gdy porównanie jest spełnione a 0 w przeciwnym razie. Poniżej przedstawiamy wyrażenia zwracające wyniki porównań w bicie znaku. W celu uzyskania wyniku 1 lub 0 wykorzystywanego przez część języków programowania (np. C) należy wynik przesunąć o 31 miejsc w prawo. W celu uzyskania wyniku –1 lub 0, wykorzystywanego przez niektóre języki (np. Basic) należy wynik przesunąć ze znakiem o 31 miejsc w prawo.

Formuły tego typu są oczywiście bezwartościowe na maszynach klasy MIPS, Compaq Alpha oraz w naszym modelowym procesorze klasy RISC, które posiadają instrukcje porównujące i wyliczające bezpośrednio wiele z predykatów, umieszczając wartość 0 lub 1 bezpośrednio w rejestrze ogólnego przeznaczenia.

W przypadku predykatów porównań bardzo użyteczna jest instrukcja obliczająca wartość bezwzględną z ujemnym znakiem. Funkcje tę będziemy zapisywać jako „nabs". W przeciwieństwie do wartości bezwzględnej wartość tej funkcji nigdy nie ulega przepełnieniu. Na maszynach, które nie posiadają instrukcji „nabs" lecz posiadają instrukcję „abs", w miejsce nabs(x) można zastosować –abs(x). W przypadku, gdy x jest największą liczbą ujemną, instrukcja ta spowoduje dwukrotne przepełnienie lecz wynik będzie prawidłowy (zakładamy, że wartość bezwzględna oraz negacja maksymalnej liczby ujemnej są równe). Ponieważ niektóre maszyny nie udostępniają instrukcji „abs" ani „nabs", przedstawiamy również alternatywne rozwiązanie, niewykorzystujące tych instrukcji.

Funkcja „nlz" zwraca liczbę wiodących zer w argumencie. Funkcja „doz" (ang. *difference or zero* — różnica lub zero) zostanie opisana w podrozdziale 2.18, na stronie 54.

$$
\begin{array}{lll}
x = y: & \mathrm{abs}(x - y) - \mathbf{1} & \\
 & \mathrm{abs}(x - y + \mathbf{0x8000\ 0000}) & \\
 & \mathrm{nlz}(x - y) << 26 & \\
 & -(\mathrm{nlz}(x - y) \overset{u}{>>} 5) & \\
 & \neg(x - y \mid y - x) & \\
x \neq y: & \mathrm{nabs}(x - y) & \\
 & \mathrm{nlz}(x - y) - \mathbf{32} & \\
 & x - y \mid y - x & \\
x < y: & (x - y) \oplus [(x \oplus y) \,\&\, ((x - y) \oplus x)] & \\
 & (x \,\&\, \neg y) \mid ((x \equiv y) \,\&\, (x - y)) & \\
 & \mathrm{nabs}(\mathrm{doz}(y, x)) & [\mathrm{GSO}] \\
x \leq y: & (x \mid \neg y) \,\&\, ((x \oplus y) \mid \neg(y - x) & \\
 & ((x \equiv y) \overset{s}{>>} 1) + (x \,\&\, \neg y) & [\mathrm{GSO}] \\
\overset{u}{x < y}: & (\neg x \,\&\, y) \mid ((x \equiv y) \,\&\, (x - y)) & \\
 & (\neg x \,\&\, y) \mid ((\neg x \mid y) \,\&\, (x - y)) & \\
\overset{u}{x \leq y}: & (\neg x \mid y) \,\&\, ((x \oplus y) \mid \neg(y - x)) &
\end{array}
$$

W zapisach $x > y$ oraz $x \geq y$ itp. możemy zamienić x oraz y i w ten sposób uzyskamy procedury wyliczające wyniki porównań $x < y$, $x \leq y$ itd. Zamiast *dodawać* wartość **0x8000 0000** możemy zastosować dowolną operację odwracającą najbardziej znaczący bit (wartości x, y lub $x - y$).

Kolejny zestaw formuł możemy wyprowadzić dzięki obserwacji, że wartość predykatu $x < y$ można uzyskać analizując znak działania $x/2 - y/2$ i wynik powyższych obliczeń

nie ulega przepełnieniu. Wynik można poprawić odejmując od niego 1 w przypadkach, gdy w wyniku przesunięć utracone zostają ważne informacje:

$$x < y: \quad (x \overset{s}{>>} 1) - (y \overset{s}{>>} 1) - (\neg x \,\&\, y \,\&\, \mathbf{1})$$
$$x \overset{u}{<} y: \quad (x \overset{u}{>>} 1) - (y \overset{u}{>>} 1) - (\neg x \,\&\, y \,\&\, \mathbf{1})$$

Powyższe operacje wymagają siedmiu instrukcji (sześciu, jeśli dostępna jest instrukcja *and not*), co niestety daje gorszy rezultat niż operacje omówione wcześniej (wymagające pięć do siedmiu instrukcji, w zależności od dostępnego zestawu instrukcji logicznych).

Powyższe formuły wykorzystujące instrukcję „nlz" zostały zaprezentowane przez [Shep] a formuła dotycząca predykatu $x = y$ pochodząca z tego źródła jest szczególnie użyteczna, ponieważ niewielka modyfikacja tej formuły pozwala na uzyskanie wyniku typu 0 lub 1 za pomocą zaledwie trzech instrukcji:

$$\mathrm{nlz}(x - y) \overset{u}{>>} 5 \,.$$

Porównania ze znakiem z wartością 0 są na tyle powszechne, że warto poświęcić im więcej uwagi. Poniżej przedstawiamy kilka formuł, w większości uzyskanych bezpośrednio z formuł wcześniejszych. Ponownie wynik jest zapisywany w bicie znaku.

$x = \mathbf{0}$: $\mathrm{abs}(x) - \mathbf{1}$

$\mathrm{abs}(x + \mathbf{0x8000\ 0000})$

$\mathrm{nlz}(x) << 26$

$-(\mathrm{nlz}(x) \overset{u}{>>} 5)$

$\neg(x \mid -x)$

$\neg x \,\&\, (x - \mathbf{1})$

$x \neq \mathbf{0}$: $\mathrm{nabs}(x)$

$\mathrm{nlz}(x) - \mathbf{32}$

$x \mid -x$

$(x \overset{u}{>>} 1) - x$ [CWG]

$x < \mathbf{0}$: x

$x \leq \mathbf{0}$: $x \mid (x - \mathbf{1})$

$x \mid \neg\neg x$

$x > \mathbf{0}$: $x \oplus \mathrm{nabs}(x)$

$(x \overset{s}{>>} 1) - x$

$\neg x \,\&\, \neg x$

$x \geq \mathbf{0}$: $\neg x$

Porównania ze znakiem można uzyskać z porównań bez znaku dodając do porównywanych wartości 2^{31} i wyniki tych operacji porównać jako liczby bez znaku. Odwrotna operacja również daje prawidłowe wyniki.

$$x < y = x + 2^{31} \overset{u}{<} y + 2^{31},$$

$$x \overset{u}{<} y = x - 2^{31} < y - 2^{31}.$$

Podobne związki zachodzą również dla relacji $\leq$, $\overset{u}{\leq}$ itd. Dodawanie i odejmowanie wartości 2^{31} są operacjami równoważnymi, gdyż odpowiadają odwróceniu bitu znaku.

Inny sposób uzyskiwania porównań ze znakiem za pomocą porównań bez znaku jest oparty na obserwacji, że w przypadku, gdy x oraz y są tego samego znaku, wtedy $(x < y) = (x \overset{u}{<} y)$, jeśli natomiast mają różne znaki, spełniona jest własność $(x < y) = (x \overset{u}{>} y)$ [Lamp]. Ponownie transformacja odwrotna również działa, zatem:

$$x < y = (x \overset{u}{<} y) \oplus x_{31} \oplus y_{31}$$

oraz

$$x \overset{u}{<} y = (x < y) \oplus x_{31} \oplus y_{31}.$$

W powyższych formułach x_{31} oraz y_{31} określają odpowiednio bity znaku w zmiennych x oraz y. Podobne relacje mają miejsce również w przypadku porównań $\leq$ oraz $\overset{u}{\leq}$ itd.

Wykorzystując dowolną z wspomnianych obserwacji możemy skonstruować dowolny predykat porównawczy (za wyjątkiem = oraz $\neq$) na podstawie dowolnego innego predykatu za pomocą co najwyżej trzech dodatkowych instrukcji na większości procesorów. Jako podstawę przyjmijmy predykat $x \overset{u}{\leq} y$ jako jeden z najprostszych w implementacji (wynikiem jego jest bit przeniesienia z operacji $y - x$). Pozostałe predykaty możemy zaimplementować w następujący sposób:

$$x < y = \neg(y + 2^{31} \overset{u}{\leq} x + 2^{31})$$

$$x \leq y = x + 2^{31} \overset{u}{\leq} y + 2^{31}$$

$$x > y = \neg(x + 2^{31} \overset{u}{\leq} y + 2^{31})$$

$$x \geq y = y + 2^{31} \overset{u}{\leq} x + 2^{31}$$

$$x \overset{u}{<} y = \neg(y \overset{u}{\leq} x)$$

$$x \overset{u}{>} y = \neg(x \overset{u}{\leq} y)$$

$$x \overset{u}{\geq} y = y \overset{u}{\leq} x$$

Konstrukcja predykatów porównawczych za pomocą bitu przeniesienia

Jeśli w procesorze istnieje mechanizm pozwalający na szybkie umieszczenie bitu przeniesienia w rejestrze ogólnego przeznaczenia, można go wykorzystać do skonstruowania niektórych z predykatów porównań. Poniżej wymieniamy niektóre z tego typu relacji. Notacja carry(*wyrażenie*) oznacza bit przeniesienia wygenerowany w wyniku wyliczenia *wyrażenia*. Zakładamy, że bit przeniesienia w operacji *odejmowania* $x - y$ jest równoznaczny z wartością wyjściową z sumatora dla operacji $x + \bar{y} + 1$, która stanowi dopełnienie pożyczki:

$x = y$: carry($0 - (x - y)$) lub carry($(x + \bar{y}) + 1$) lub carry($(x - y - 1) + 1$)

$x \neq y$: carry($(x - y) - 1$), tzn. carry($(x - y) + (-1)$)

$x < y$: $\neg$carry($(x + 2^{31}) - (y + 2^{31})$)

$x \leq y$: carry($(y + 2^{31}) - (x + 2^{31})$)

$x \overset{u}{<} y$: $\neg$carry($x - y$)

$x \overset{u}{\leq} y$: carry($y - x$)

$x = 0$: carry($0 - x$) lub carry($\bar{x} + 1$)

$x \neq 0$: carry($x - 1$), tzn. carry($x + (-1)$)

$x < 0$: carry($x + x$)

$x \leq 0$: carry($2^{31} - (x + 2^{31})$)

Dla predykatu $x > y$ należy zastosować uzupełnienie wyrażenia $x \leq y$, podobnie jak dla pozostałych relacji wykorzystujących predykat większości.

Problem obliczania wyrażeń predykatów na platformie IBM RS/6000 oraz pokrewnej PowerPC poddano działaniu aplikacji GNU Superoptimiser [GK]. Procesor RS/6000 posiada instrukcje abs(x), nabs(x), doz(x, y) oraz kilka form operacji *dodawania* i *odejmowania*. Okazało się, że RS/6000 jest w stanie wyliczyć wszystkie predykaty całkowitoliczbowe za pomocą trzech lub mniejszej liczby instrukcji elementarnych (tzn. takich, których wykonanie trwa pojedynczy cykl zegara). Był to wynik, który zaskoczył nawet architektów tej maszyny. Pojęcie *wszystkie predykaty* oznacza sześć dwuoperandowych porównań ze znakiem oraz cztery dwuoperandowe porównania bez znaku. We wszystkich tych operacjach drugim z operandów było zero i wszystkie zwracały wynik 1 lub 0, względnie –1 lub 0. Procesor PowerPC, w którym nie zostały zaimplementowane instrukcje abs(x), nabs(x) oraz doz(x,y), jest w stanie obliczyć wszystkie omówione predykaty w czterech lub mniejszej liczbie instrukcji elementarnych.

Sposoby rozstrzygania predykatów porównawczych

Większość komputerów udostępnia sposoby wyliczania predykatów porównań całkowitoliczbowych do jednobitowego wyniku. Wynik może być zapisany w specjalnym „rejestrze warunkowym" (ang. *condition register*) lub, w przypadku niektórych platform (jak nasz modelowy procesor RISC), w rejestrze ogólnego przeznaczenia. Mechanizm ten jest zaimplementowany za pomocą operacji *odejmowania* operandów a następnie niewielkiej liczby operacji logicznych na wyniku, które zwracają jednobitowy wynik całego porównania.

Poniżej opisujemy logikę takich operacji. Zostało przyjęte założenie, że komputer oblicza $x - y$ jako $x + \overline{y} + 1$ i w wyniku są dostępne następujące informacje:

- C_0, przeniesienie z najbardziej znaczącej pozycji;
- C_i, przeniesienie na najbardziej znaczącą pozycję;
- N, bit znaku w wyniku,
- Z, które ma wartość 1 jeśli wynik z pominięciem C_0 składa się z samych zer, w przeciwnym wypadku ma wartość 0.

Następnie w notacji algebry boolowskiej możemy zapisać następujące predykaty:

V: $C_i \oplus C_0$ (przepełnienie ze znakiem)

$x = y$: Z

$x \neq y$: $\overline{Z}$

$x < y$: $N \oplus V$

$x \leq y$: $(N \oplus V) + Z$

$x > y$: $(N \equiv V)\overline{Z}$

$x \geq y$: $N \equiv V$

$x \overset{u}{<} y$: $\overline{C_0}$

$x \overset{u}{\leq} y$: $\overline{C_0} + Z$

$x \overset{u}{>} y$: $C_0\overline{Z}$

$x \overset{u}{\geq} y$: C_0

2.12. Wykrywanie przepełnienia

Przepełnienie (ang. *overflow*) oznacza sytuację, gdy wynik operacji arytmetycznej jest zbyt duży lub za mały, aby mógł być prawidłowo reprezentowany w rejestrze wynikowym. Niniejszy podrozdział omawia metody programistyczne pozwalające na wykrycie

sytuacji przepełnienia bez potrzeby wykorzystania „bitów stanu" (ang. *status bits*) maszyny, które to są często dostępne właśnie w tym celu. Opisywane techniki mogą jednak być potrzebne, szczególnie w sytuacji, gdy wykorzystywana platforma nie udostępnia bitów stanu (na przykład MIPS), jak również dlatego, że nawet w przypadku, gdy wykorzystywana platforma udostępnia bity stanu, najczęściej dostęp do nich w języku wyższego poziomu jest trudny lub wręcz niemożliwy.

Dodawanie i odejmowanie ze znakiem

W przypadku wystąpienia sytuacji przepełnienia w wyniku operacji *dodawania* lub *odejmowania*, współczesne procesory nieodwołalnie odrzucają najbardziej znaczący bit wyniku zachowując mniej znaczące bity. Przepełnienie operacji *dodawania* występuje wyłącznie w przypadku, gdy sumowane są operandy o tym samym znaku natomiast wynik ma znak przeciwny w stosunku do operandów. Ta sama zasada ma zastosowanie również w przypadku, gdy instrukcja dodawania uwzględnia przeniesienie wejściowe, to znaczy obliczenia są typu $x + y + 1$. Operacje tego typu wykorzystuje się w przypadku sumowania liczb całkowitych złożonych z wielu słów, kiedy to kolejne sumowanie obejmuje odpowiadające sobie pozycjami słowa oraz bit przeniesienia wejściowego, który może mieć wartość 0 lub 1.

Jako dowód reguły dodawania załóżmy, że sumowane liczby x oraz y są liczbami całkowitymi o długości słowa ze znakiem, zaś c jest bitem przeniesienia wejściowego (ang. *carry-in*) o wartości 0 lub 1. Dla uproszczenia załóżmy 4 bitowy rozmiar rejestru procesora. W przypadku różnych znaków operandów mamy na przykład następujące założenia:

$$-8 \le x \le -1$$

oraz

$$0 \le y \le 7.$$

Podobne ograniczenia można przyjąć, gdy liczba x jest nieujemna, natomiast liczba y jest ujemna. W obu przypadkach po dodaniu tych wartości oraz, opcjonalnie wartości 1 bitu c otrzymamy następującą zależność:

$$-8 \le x + y + c \le 7.$$

Wynik takiej operacji, niezależnie od sytuacji, może zostać zinterpretowany jako wartość 4-bitowa ze znakiem, zatem w przypadku operandów o różnych znakach nie występuje przepełnienie.

Załóżmy teraz, że x oraz y posiadają różne znaki. Możliwe są dwa przypadki:

(a)	(b)
$-8 \le x \le -1$	$0 \le x \le 7$
$-8 \le y \le -1$	$0 \le y \le 7$

Stąd:

(a)	(b)
$-16 \le x + y + c \le -1$	$0 \le x + y + c \le 15$

Przepełnienie występuje w sytuacji, gdy suma nie może zostać przedstawiona za pomocą 4-bitowej liczby ze znakiem, to znaczy w przypadku, gdy:

(a)	(b)
$-16 \le x + y + c \le -9$	$8 \le x + y + c \le 15$

W przypadku (a) oznacza to, że najbardziej znaczący bit 4-bitowej sumy ma wartość 0, co stanowi negację znaku liczb x oraz y. W przypadku (b) oznacza to, że najbardziej znaczący bit 4-bitowej sumy ma wartość 1, co stanowi negację znaku liczb x oraz y.

Dla różnicy liczb składających się z wielu słów interesujące nas obliczenia mają postać $x - y - c$, gdzie c ponownie ma wartość 0 lub 1, przy czym wartość 1 oznacza pożyczkę. Z analizy podobnej do powyższej wynika, iż przepełnienie w operacji $x - y - c$ ma miejsce w przypadku, gdy x oraz y są przeciwnych znaków oraz znak wyniku jest negacją znaku liczby x (czyli ma znak zgodny ze znakiem liczby y).

Powyższe rozważania prowadzą do następujących wyrażeń predykatów przepełnienia, których wynik jest umieszczany w bicie znaku. Wynik tych wyrażeń przesunięty w prawo o 31 pozycji ze znakiem lub bez znaku da wynik 1 lub 0, względnie –1 lub 0.

$x + y + c$	$x - y - c$
$(x \equiv y)\ \&\ ((x + y + c) \oplus x)$	$(x \oplus y)\ \&\ ((x - y - c) \oplus x)$
$((x + y + c) \oplus x)\ \&\ ((x + y + c) \oplus y)$	$((x - y - c) \oplus x)\ \&\ ((x - y - c) \equiv y)$

Wybór drugiej możliwości w pierwszej kolumnie i pierwszej w drugiej kolumnie (unikając konieczności wykonywania operacji *równoważności*) powoduje, że nasz wzorcowy RISC jest w stanie wyliczyć wynik w trzech instrukcjach, oprócz instrukcji potrzebnych do wyliczenia $x + y + c$ lub $x - y - c$. Możemy do tego dodać czwartą instrukcję, będącą instrukcją rozgałęzienia w przypadku wartości ujemnej (ang. *branch if negative*), dzięki której możemy obsłużyć sytuację przepełnienia.

Jeśli aktywna jest obsługa przerwań w przypadku przepełnienia, przydatna może okazać się możliwość kontroli przepełnienia bez wywoływania go. Jeden z tego typu sposobów niewykorzystujących rozgałęzień można przedstawić następująco:

$x + y + c$	$x - y - c$
$z \leftarrow (x \equiv y)\ \&\ \mathbf{0x8000\ 0000}$	$z \leftarrow (x \oplus y)\ \&\ \mathbf{0x8000\ 0000}$
$(x \equiv y)\ \&\ ((x \oplus z) + y + c) \equiv y$	$(x \oplus y)\ \&\ ((x \oplus z) - y - c) \oplus y$

W przypadku, gdy x i y mają ten sam znak, rejestrowi z w lewej kolumnie zostanie przypisana wartość **0x8000 0000**, natomiast gdy znaki liczb x i y różnią się, z będzie miało wartość **0**. Następnie dodawanie w drugim wyrażeniu odbywa się z użyciem liczb o przeciwnych znakach, zatem nie ma obawy o wystąpienie przepełnienia. W przypadku, gdy x oraz y nie są liczbami ujemnymi, bit znaku w drugim z wyrażeń będzie miał wartość 1 wyłącznie wtedy, gdy $(x - \mathbf{2}^{31}) + y + c \geq \mathbf{0}$, to znaczy wtedy i tylko wtedy, gdy $x + y + c \geq 2^{31}$, co stanowi warunek przepełnienia w przypadku obliczeń typu $x + y + c$. Jeśli x oraz y są wartościami ujemnymi, bit znaku w drugim z wyrażeń będzie miał wartość 1 wtedy i tylko wtedy, gdy $(x + \mathbf{2}^{31}) + y + c < \mathbf{0}$, to znaczy wtedy, gdy $x + y + c < -2^{31}$, co również jest jednym z warunków przepełnienia. Wyrażenie $x \equiv y$ zapewnia prawidłowy wynik (0 na pozycji znaku) wtedy, gdy x oraz y są przeciwnych znaków. Podobne uwagi odnoszą się do operacji *odejmowania* (prawa kolumna). Przedstawiony kod wymaga dziewięciu instrukcji z podstawowego zestawu instrukcji RISC.

Może wydawać się, że dostępność bitu przeniesienia z operacji dodawania może być pomocna w wyliczeniu całkowitoliczbowego predykatu przepełnienia. W naszym przypadku tak nie jest. Istnieje jednak metoda wykorzystująca ten element.

Jeśli x jest liczbą całkowitą ze znakiem, $x + 2^{31}$ można prawidłowo zinterpretować jako liczbę całkowitą bez znaku przez odwrócenie najbardziej znaczącego bitu wyniku. Przepełnienie ze znakiem w kierunku dodatnim (ang. *signed overflow in the positive direction*) występuje wtedy, gdy $x + y \geq 2^{31}$, to znaczy wtedy, gdy $(x + 2^{31}) + (y + 2^{31}) \geq 3 \cdot 2^{31}$. Ostatni warunek można scharakteryzować wystąpieniem sytuacji przeniesienia w operacji *dodawania bez znaku* (co oznacza, że suma jest większa lub równa 2^{32}) podczas, gdy najbardziej znaczący bit sumy ma wartość 1. Podobnie przepełnienie w kierunku ujemnym (ang. *signed overflow in the negative direction*) występuje wyłącznie w sytuacji, gdy przeniesienie ma wartość 0 a najbardziej znaczący bit sumy również ma wartość 0.

Powyższe spostrzeżenia mogą posłużyć do opracowania następującego algorytmu wykrywania przepełnienia podczas operacji *dodawania ze znakiem*:

> Wyliczamy $(x \oplus \mathbf{2}^{31}) + (y \oplus \mathbf{2}^{31})$, co daje sumę s oraz przeniesienie c. Przepełnienie sumy $x + y$ wystąpi wtedy i tylko wtedy, gdy c ma wartość równą najbardziej znaczącemu bitowi sumy s.

Suma stanowi prawidłową wartość w przypadku dodawania ze znakiem, ponieważ odwrócenie najbardziej znaczących bitów obydwu operandów nie ma wpływu na ich sumę.

Algorytm dla odejmowania jest podobny z tą różnicą, że w pierwszym kroku operację *dodawania* zastępujemy operacją *odejmowania*. Zakładamy, że przeniesienie w tym działaniu jest efektem zastąpienia działania $x - y$ działaniem $x + \bar{y} + 1$. Wyliczona różnica stanowi prawidłowy wynik w przypadku operacji odejmowania ze znakiem.

Opisane formuły są zapewne interesujące, lecz na większości maszyn nie będą tak wydajne jak formuły niewykorzystujące bitu przeniesienia, to znaczy *przepełnienie* = $(x \equiv y)$ & $(s \oplus x)$ dla dodawania oraz *przepełnienie* = $(x \oplus y)$ & $(d \oplus x)$ dla odejmowania, gdzie s oraz d stanowią, odpowiednio, sumę oraz różnicę liczb x i y.

Sposoby ustawiania bitu przepełnienia w operacjach dodawania i odejmowania ze znakiem

Maszyny często ustawiają bit przepełnienia w operacji *dodawania* ze znakiem na zasadzie „przeniesienie na pozycji znaku nie jest równe przeniesieniu poza pozycję znaku". Co ciekawe, zasada ta umożliwia otrzymywanie poprawnych wyników dla przepełnienia w operacjach *dodawania* i *odejmowania*, z założeniem, że różnica $x - y$ jest obliczana następująco: $x + \bar{y} + 1$. Zasada ta umożliwia otrzymywanie poprawnych wyników również w sytuacji wystąpienia przeniesienia wejściowego (ang. *carry-in*) oraz pożyczki wejściowej (ang. *borrow-in*). Jak się jednak wydaje, metoda ta nie wpływa na powstawanie szczególnie dobrych technik obliczania przepełnienia w sposób programistyczny, choć obliczenie przeniesienia na pozycji znaku jest łatwym zadaniem.

Reguły obliczania pożyczki (ang. *borrow*) oraz przeniesienia (ang. *carry*) na pozycji znaku dla operacji *dodawania* i *odejmowania* można przedstawić następująco (c przyjmuje wartość **0** lub **1**):

przeniesienie

$$(x + y + c) \oplus x \oplus y$$

pożyczka

$$(x - y - c) \oplus x \oplus y$$

W rzeczywistości formuły te wyliczają wartość przeniesienia oraz pożyczki na każdej pozycji w wyniku.

Dodawanie i odejmowanie bez znaku

W celu wyliczenia predykatu przepełnienia w operacjach *dodawania* i *odejmowania* bez znaku można zastosować następujący, niewykorzystujący rozgałęzień kod. Wynik jest zapisywany na pozycji znaku. Wyrażenia wykorzystujące przesunięcie w prawo są użyteczne głównie w przypadku, gdy wiadomo, iż $c = \mathbf{0}$. Wyrażenia w nawiasach kwadratowych wyliczają przeniesienie lub pożyczkę występujące na najmniej znaczącej pozycji.

$x + y + c$, bez znaku $(x \,\&\, y) \mid ((x \mid y) \,\&\, \neg(x + y + c))$

$$(x \overset{u}{>>} \mathbf{1}) + (y \overset{u}{>>} \mathbf{1}) + [((x \,\&\, y) \mid ((x \mid y) \,\&\, c)) \,\&\, \mathbf{1}]$$

$x - y - c$, bez znaku $(\neg x \,\&\, y) \mid ((x \equiv y) \,\&\, (x - y - c))$

$$(\neg x \,\&\, y) \mid ((\neg x \mid y) \,\&\, (x - y - c))$$

$$(x \overset{u}{>>} \mathbf{1}) - (y \overset{u}{>>} \mathbf{1}) - [((\neg x \,\&\, y) \mid ((\neg x \mid y) \,\&\, c)) \,\&\, \mathbf{1}]$$

Dla *dodawania* i *odejmowania* bez znaku dostępne są znacznie prostsze formuły porównań [MIPS]. W przypadku dodawania bez znaku przepełnienie (przeniesienie) występuje w przypadku, gdy suma ma mniejszą wartość (pod względem porównania bez znaku), niż którykolwiek z operandów. Poniżej prezentujemy tę formułę oraz pokrewne. Niestety w formułach tych nie ma możliwości wykorzystania zmiennej c, reprezentującej przeniesienie lub pożyczkę. Zamiast tego program musi sprawdzać wartość zmiennej c i wykorzystywać różne rodzaje porównań w zależności od wartości c (**0** lub **1**).

$x + y$, bez znaku $\quad \neg x \overset{u}{<} y$

$\qquad\qquad\qquad\qquad x + y \overset{u}{<} x$

$x + y + 1$, bez znaku $\quad \neg x \overset{u}{\leq} y$

$\qquad\qquad\qquad\qquad x + y + 1 \overset{u}{\leq} x$

$x - y$, bez znaku $\quad x \overset{u}{<} y$

$\qquad\qquad\qquad\qquad x - y \overset{u}{>} x$

$x - y - 1$, bez znaku $\quad x \overset{u}{\leq} y$

$\qquad\qquad\qquad\qquad x - y - 1 \overset{u}{\geq} x$

Pierwsza formuła w każdym z powyższych przypadków jest wyliczana przed operacją *dodawania* lub *odejmowania*, które mogą spowodować przepełnienie i umożliwia przeprowadzenie sprawdzenia występowania sytuacji przepełnienia bez wywoływania jej. Druga z formuł jest wyliczana po operacji *dodawania* lub *odejmowania* i może spowodować przepełnienie.

Jak dotąd nie wymyślono podobnego prostego sposobu (wykorzystującego porównania) na określenie predykatu przepełnienia ze znakiem.

Mnożenie

W przypadku operacji *mnożenia* przepełnienie oznacza, że wynik nie może być zapisany na 32 bitach (lecz zawsze może być zapisany na 64 bitach, niezależnie od tego, czy liczba jest ze znakiem czy bez). Sprawdzanie sytuacji przepełnienia jest proste w przypadku, gdy mamy dostęp do bardziej znaczących 32 bitów wyniku. Załóżmy, że dwie połowy 64-bitowego iloczynu, odpowiednio bardziej znaczącą i mniej znaczącą, zapiszemy jako hi($x * y$) oraz lo($x * y$). Predykaty przepełnienia możemy wyliczyć w następujący sposób [MIPS]:

$x * y$, bez znaku: $\quad \text{hi}(x * y) \neq 0$

$x * y$, ze znakiem; $\quad \text{hi}(x * y) \neq (\text{lo}(x * y) \overset{s}{>>} 31)$

Jednym ze sposobów sprawdzenia przepełnienia mnożenia jest wykonanie mnożenia a następnie dzielenia sprawdzającego poprawność wyniku. Należy jednak uważać, aby nie spowodować operacji dzielenia przez 0, ponadto z mnożeniem liczb ze znakiem są

związane dalsze komplikacje. Przepełnienie w przypadku operacji *mnożenia* występuje wtedy, gdy następujące wyrażenia zwracają wartość **prawda**:

Bez znaku $z \leftarrow x * y$

$$y \neq 0 \overrightarrow{\&} z \overset{u}{\div} y \neq x$$

Ze znakiem: $z \leftarrow x * y$

$$(y < 0 \,\&\, x = -2^{31}) \mid (y \neq 0 \overrightarrow{\&} z \div y \neq x)$$

W przypadku, gdy $x = -2^{31}$ oraz $y = -1$ pojawiają się kłopoty. W tym przypadku mnożenie powoduje przepełnienie, lecz procesor może zwrócić wynik -2^{31}. To powoduje przepełnienie dzielenia i w rezultacie na niektórych maszynach można spodziewać się przypadkowego wyniku. Dlatego ten przypadek wymaga osobnego sprawdzenia za pomocą formuły $y < 0 \,\&\, x = -2^{31}$. Wyrażenia przestawione powyżej w celu uniknięcia dzielenia przez 0 wykorzystują operację „koniunkcji warunkowej" (ang. *conditional and*). W języku C w tym miejscu należy wykorzystać operator &&.

Możliwe jest również zastosowanie dzielenia sprawdzającego wystąpienie sytuacji przepełnienia bez wykonywania mnożenia (to znaczy unikając ryzyka wystąpienia przepełnienia). W przypadku liczb całkowitych bez znaku iloczyn powoduje przepełnienie wtedy i tylko wtedy, gdy $xy > 2^{32} - 1$, lub $x > ((2^{32} - 1) / y)$ lub, z uwagi na fakt, że x jest liczbą całkowitą, $x > \lfloor (2^{32} - 1) / y \rfloor$. W arytmetyce komputerowej możemy zapisać to w następujący sposób:

$$y \neq 0 \overrightarrow{\&} x \overset{u}{>} (\mathbf{0xFFFFFFFF} \overset{u}{\div} y).$$

W przypadku liczb całkowitych ze znakiem określenie przepełnienia iloczynu $x * y$ nie jest tak łatwe. W przypadku, gdy x oraz y są liczbami o jednakowym znaku, przepełnienie występuje wtedy i tylko wtedy, gdy $xy > 2^{31} - 1$. W przypadku, gdy liczby te mają różne znaki, przepełnienie występuje wtedy i tylko wtedy, gdy $xy < -2^{31}$. Poszczególne przypadki można sprawdzać w sposób przedstawiony w tabeli 2.2. Metoda ta wykorzystuje dzielenie ze znakiem:

Tabela 2.2. *Testowanie sytuacji przepełnienia w przypadku mnożenia ze znakiem*

	$y > 0$	$y \leq 0$
$x > 0$	$x > \mathbf{0x7FFFFFFF} \div y$	$y < \mathbf{0x80000000} \div x$
$x \leq 0$	$x < \mathbf{0x80000000} \div y$	$x \neq 0 \overrightarrow{\&} y < \mathbf{0x7FFFFFFF} \div x$

Testy przestawione w tabeli 2.2 są dość niezręczne z powodu występowania czterech różnych przypadków. Nie jest łatwo zunifikować te wyrażenia, ponieważ należy unikać przepełnienia, jak również niemożliwe jest zapisanie liczby $+2^{31}$.

Test ten można uprościć w przypadku, gdy mamy dostęp do instrukcji dzielenia bez znaku. Możemy zastosować wartości bezwzględne liczb x oraz y, które można bez problemu zapisać w reprezentacji całkowitej bez znaku. Poniżej przedstawiamy procedurę

sprawdzenia wszystkich warunków testu. Zmienna $c = 2^{31} - 1$ w przypadku, gdy x i y są liczbami o jednakowym znaku, w przeciwnym razie $c = 2^{31}$.

$$c \leftarrow ((x \equiv y) \overset{s}{>>} 31) + 2^{31}$$
$$x \leftarrow \text{abs}(x)$$
$$y \leftarrow \text{abs}(y)$$
$$y \neq 0 \overset{\rightarrow}{\&} x \overset{u}{>} (c \overset{u}{\div} y)$$

W celu oszacowania możliwości wystąpienia przepełnienia operacji $x * y$ możemy wykorzystać instrukcję zwracającą liczbę zer wiodących. Oszacowanie tego typu można dość dobrze dopracować w ten sposób, że będzie zwracać stosunkowo dokładne wyniki. W pierwszej kolejności rozpatrzmy mnożenie liczb bez znaku. Łatwo można wykazać, że jeśli liczby x oraz y interpretowane jako wartości 32-bitowe posiadają, odpowiednio, m oraz n zer wiodących, to 64-bitowy wynik będzie posiadał $m + n$ lub $m + n + 1$ zer wiodących (lub 64, jeśli $x = 0$ lub $y = 0$). Stąd mamy następujące oszacowanie:

$\text{nlz}(x) + \text{nlz}(y) \geq 32$: mnożenie na pewno nie powoduje przepełnienia

$\text{nlz}(x) + \text{nlz}(y) \leq 30$: mnożenie na pewno powoduje przepełnienie.

W przypadku, gdy $\text{nlz}(x) + \text{nlz}(y) = 31$ przepełnienie może ale nie musi wystąpić. W tym przypadku dalsze oszacowanie możliwości wystąpienia przepełnienia można przeprowadzić za pomocą wyliczenia $t = x\lfloor y/2 \rfloor$. Obliczenie to nie spowoduje przepełnienia. Wartość $xy = 2t$ lub w przypadku, gdy y jest liczbą nieparzystą — $2t + x$. Iloczyn xy powoduje przepełnienie wtedy, gdy $t \geq 2^{31}$. Powyższe spostrzeżenie prowadzą do określenia algorytmu wyliczającego xy, który wywołuje instrukcję rozgałęzienia w przypadku, gdy następuje przepełnienie. Algorytm ten prezentujemy na listingu 2.2.

Listing 2.2. *Oszacowanie przepełnienia operacji mnożenia bez znaku*

```
unsigned x, y, z, m, n, t;

m = nlz(x);
n = nlz(y);
if (m + n <= 30) goto overflow;
t = x*(y >> 1);
if ((int)t < 0) goto overflow;
z = t*2;
if (y & 1) {
   z = z + x;
   if (z < x) goto overflow;
}
// z zawiera poprawny iloczyn x oraz y.
```

W przypadku mnożenia liczb całkowitych ze znakiem można również częściowo oszacować możliwość wystąpienia przepełnienia, wykorzystując liczbę zer wiodących w czynnikach nieujemnych oraz liczbę jedynek wiodących w czynnikach ujemnych. Załóżmy, że:

$$m = \mathrm{nlz}(x) + \mathrm{nlz}(\overline{x})$$

$$n = \mathrm{nlz}(y) + \mathrm{nlz}(\overline{y})$$

Otrzymujemy następującą zależność:

$m + n \geq \mathbf{34}$: mnożenie na pewno nie powoduje przepełnienia

$m + n \leq \mathbf{31}$: mnożenie na pewno powoduje przepełnienie.

Tym razem mamy dwa niejednoznaczne przypadki: 32 oraz 33. W przypadku, gdy $m + n = 33$ iloczyn xy powoduje przepełnienie wyłącznie wtedy, gdy obydwa czynniki są liczbami ujemnymi a iloczyn wynosi dokładnie 2^{31} (wynik w procesorze zostanie zinterpretowany jako -2^{31}), zatem przypadek ten może zostać rozpoznany za pomocą sprawdzenia, czy wynik posiada poprawny znak (oznacza to, że przepełnienie występuje wtedy, gdy $m \oplus n \oplus (m * n) < 0$). W przypadku, gdy $m + n = 32$ nie jest łatwo określić przypadki przepełnienia.

Nie będziemy się zagłębiać w dalsze szczegóły. Należy jedynie wspomnieć, że oszacowanie przepełnienia w przypadku mnożenia liczb całkowitych ze znakiem można przeprowadzić za pomocą wyrażenia $\mathrm{nlz}(\mathrm{abs}(x)) + \mathrm{nlz}(\mathrm{abs}(y))$. W tym przypadku ponownie występują jednak dwa niejednoznaczne przypadki, 31 oraz 32.

Dzielenie

W przypadku dzielenia ze znakiem $x \div y$ przepełnienie występuje, gdy spełniona jest następująca własność:

$$y = \mathbf{0} \mid (x = \mathbf{0x80000000} \;\&\; y = -\mathbf{1})$$

Większość procesorów sygnalizuje (lub wywołuje specjalne przerwanie pułapki — *trap*) w przypadku wywołania działania $\mathbf{0} \div \mathbf{0}$.

Bezpośrednia implementacja powyższego wyrażenia wykorzystuje siedem instrukcji, wliczając w to rozgałęzienie do kodu obsługującego sytuację przepełnienia. Trzy z tych siedmiu instrukcji to instrukcje rozgałęzień. Nie są znane żadne szczególnie skuteczne usprawnienia tego kodu, lecz poniżej przedstawiamy kilka możliwości:

$$[\mathrm{abs}(y \oplus \mathbf{0x80000000}) \mid (\mathrm{abs}(x) \;\&\; \mathrm{abs}(y \equiv \mathbf{0x80000000}))] < \mathbf{0}$$

Po obliczeniu wyrażenia w nawiasach kwadratowych należy wykonać rozgałęzienie, jeśli wynik jest mniejszy od 0. Wyrażenie to wymaga zastosowania dziewięciu instrukcji, wliczając w to załadowanie stałej oraz ostatnie rozgałęzienie, pod warunkiem, że wykorzystujemy maszynę udostępniającą zastosowane instrukcje oraz dodatkowo udostępniającą instrukcje zwracające informację o zerowej wartości wyniku, niejako „przy okazji".

Inne możliwości obejmują wyliczenie wartości pomocniczej z:

$$z \leftarrow (x \oplus \mathbf{0x80000000}) \mid (y + \mathbf{1})$$

Przypisanie to wykorzystuje trzy instrukcje na większości maszyn.

Następnie należy przeprowadzić sprawdzenie i rozgałęzienie w przypadku, gdy $y = 0 \mid z = 0$. W tym celu można zastosować jedną z poniższych formuł:

$$((y \mid -y) \,\&\, (z \mid -z)) \geq 0$$

$$(\mathrm{nabs}(y) \,\&\, \mathrm{nabs}(z)) \geq 0$$

$$((\mathrm{nlz}(y) \mid \mathrm{nlz}(z)) \overset{u}{>>} 5) \neq 0$$

Powyższe formuły wymagają zastosowania, odpowiednio, dziewięciu, siedmiu oraz ośmiu instrukcji na maszynach obsługujących odpowiednie instrukcje. Ostatnia z formuł jest dobrym wyborem w przypadku procesora PowerPC.

W przypadku dzielenia bez znaku $x \overset{u}{\div} y$ przepełnienie występuje wyłącznie w przypadku, gdy $y = 0$.

2.13. Kod warunkowy operacji dodawania, odejmowania i mnożenia

Wiele procesorów udostępnia kod warunkowy (ang. *condition code*), charakteryzujący wynik operacji arytmetycznych na liczbach całkowitych. Często istnieje tylko jedna instrukcja wykonująca *dodawanie* a kod warunkowy określa wynik operacji w przypadku interpretacji operandów jako wartości ze znakiem i bez znaku (lecz nie obejmuje przypadków operandów z różnymi znakami). Najczęściej dostępne są następujące zdarzenia:

- wystąpienie przeniesienia (przepełnienia bez znaku);
- wystąpienie przepełnienia ze znakiem;
- znak 32-bitowego wyniku, interpretowanego jako liczba całkowita w formacie uzupełnienia dwójkowego ze znakiem, ignorujący przeniesienie i przepełnienie. Znak jest rozumiany jako liczba ujemna, 0 lub liczba dodatnia.

Niektóre starsze maszyny udostępniają również informację dotyczącą znaku (ujemny, 0 lub dodatni) wyniku o nieograniczonej precyzji (tzn. 33-bitowego wyniku dla operacji *dodawania* i *odejmowania*). Informacja ta jest jednak trudna do wykorzystania w kompilatorach języków wyższego poziomu, zatem znacznie straciła na popularności.

W przypadku dodawania możliwe jest wystąpienie dziewięciu z dwunastu możliwych kombinacji powyższych zdarzeń. Niemożliwe kombinacje obejmują „brak przeniesienia, przepełnienie i wynik > 0", „brak przeniesienia, przepełnienie i wynik = 0" oraz „przeniesienie, przepełnienie i wynik < 0". Dlatego w celu zapisania kodu warunkowego są potrzebne cztery bity. Dwie z możliwych kombinacji są unikatowe w tym sensie, że są wywoływane przez tylko jeden zestaw wartości zmiennych wejściowych. Dodawanie 0 + 0 jest jedynym sposobem wywołania sytuacji „brak przeniesienia, brak

przepełnienia, wynik = 0". Dodanie maksymalnej wartości ujemnej do samej siebie jest jedynym sposobem na wywołanie kombinacji „przeniesienie, przepełnienie, wynik = 0". Uwagi te są prawdziwe również w przypadku wykorzystania wejściowego bitu przeniesienia (ang. *carry-in*), to znaczy w przypadku operacji typu $x + y + 1$.

W przypadku odejmowania załóżmy, że operacja $x - y$ tak naprawdę oznacza operację $x + \overline{y} + 1$, z przeniesieniem tworzonym tak samo jak w przypadku *dodawania*, z tą różnicą, że w przypadku odejmowania bit przeniesienia ma odwrotne znaczenie, 1 oznacza, że wynik mieści się w pojedynczym słowie procesora, 0 oznacza, że wynik powoduje przepełnienie). W ten sposób w przypadku odejmowania możliwe jest tylko siedem kombinacji zdarzeń. Nie mogą wystąpić te same kombinacje, które są niemożliwe dla operacji *dodawania*, jak również „brak przeniesienia, brak przepełnienia, wynik = 0" oraz „przeniesienie, przepełnienie, wynik = 0".

W przypadku, gdy instrukcja obliczająca iloczyn generuje wynik zapisywany w dwóch słowach, potrzebne są dwie instrukcje *mnożenia*: jedna dla liczb całkowitych ze znakiem i druga dla liczb całkowitych bez znaku (w hipotetycznej maszynie czterobitowej w zapisie szesnastkowym ze znakiem **F** × **F** = **01**, natomiast bez znaku **F** × **F** = **E1**). W tym przypadku nie wystąpi ani sytuacja przeniesienia, ani przepełnienia, ponieważ wynik zawsze będzie mieścić się w dwusłowie.

W przypadku instrukcji mnożenia zapisujących wynik w pojedynczym słowie (to znaczy zapisujących tylko mniej znaczące słowo wyniku) przeniesienie oznacza sytuację, gdy wynik nie mieści się w słowie, w przypadku wyniku i operandów interpretowanych jako liczby całkowite bez znaku. Przepełnienie występuje wtedy, gdy wynik nie wieści się w słowie a operandy są interpretowane jako liczby ze znakiem w reprezentacji uzupełnienia dwójkowego. W tym przypadku również mamy do czynienia z dziewięcioma możliwymi kombinacjami wyniku. Niemożliwymi kombinacjami w tym przypadku są „brak przeniesienia, przepełnienie, wynik > 0", „brak przeniesienia, przepełnienie, wynik = 0" oraz „przeniesienie, brak przepełnienia, wynik = 0". Biorąc pod uwagę operacje *dodawania*, *odejmowania* oraz *mnożenia* łącznie możliwe jest wystąpienie dziesięciu kombinacji zdarzeń (kodów warunkowych).

2.14. Przesunięcia cykliczne

Są to dość proste operacje. Być może będzie to zaskoczeniem, lecz kod ten działa poprawnie dla n z przedziału od 0 do 32 włącznie, nawet w przypadku, gdy przesunięcia są wykonywane modulo 32.

Przesunięcie cykliczne w lewo o n pozycji: $y \leftarrow (x << n) \mid (x \overset{u}{>>} (32 - n))$

Przesunięcie cykliczne w prawo o n pozycji: $y \leftarrow (x \overset{u}{>>} n) \mid (x << (32 - n))$

2.15. Dodawanie i odejmowanie liczb o podwójnej długości

Wykorzystując wyrażenia przedstawione na stronie 45 możemy z łatwością zaimplementować dodawanie liczb podwójnej długości bez potrzeby wykorzystywania bitu przeniesienia procesora. Aby zademonstrować tę technikę dla dodawania liczb o podwójnej długości, przyjmijmy, że operandami są (x_1, x_0) oraz (y_1, y_0), natomiast wynik zostanie zapisany w (z_1, z_0). Cyfra 1 w indeksie dolnym oznacza bardziej znaczącą połowę wartości, natomiast 0 mniej znaczącą połowę. Zakładamy, że wykorzystywane są wszystkie 32 bity rejestrów. Mniej znaczące elementy wartości są liczbami całkowitymi bez znaku.

$$z_0 \leftarrow x_0 + y_0$$

$$c \leftarrow [(x_0 \,\&\, y_0) \mid ((x_0 \mid y_0) \,\&\, \neg z_0)] \overset{u}{>>} 31$$

$$z_1 \leftarrow x_1 + y_1 + c$$

Powyższą formułę można wykonać w dziewięciu instrukcjach. Drugi wiersz może zostać zastąpiony następującym: $c \leftarrow (z_0 \overset{u}{<} x_0)$. Zamiana ta jest możliwa w przypadku procesorów obsługujących instrukcję porównania zwracającą w rejestrze wartość **1** lub **0** (np. instrukcja SLTU — ang. *set on less than unsigned* — występująca w procesorach MIPS [MIPS]). Dzięki temu kod skraca się do czterech instrukcji.

Podobny kod można wyprowadzić dla odejmowania liczb o podwójnej długości:

$$z_0 \leftarrow x_0 - y_0$$

$$b \leftarrow [(\neg x_0 \,\&\, y_0) \mid ((x_0 \equiv y_0) \,\&\, z_0)] \overset{u}{>>} 31$$

$$z_1 \leftarrow x_1 - y_1 - b$$

Powyższa procedura na maszynie udostępniającej pełny zestaw instrukcji logicznych wykona się w ośmiu instrukcjach. W przypadku procesorów obsługujących instrukcję typu SLTU drugi wiersz można zastąpić $b \leftarrow (x_0 \overset{u}{<} y_0)$, co ponownie da nam procedurę wymagającą wykonania czterech instrukcji.

Dodawanie i odejmowanie liczb podwójnej długości można na większości maszyn wykonać w pięciu instrukcjach. W tym celu mniej znaczącą część operandów zapisujemy na mniej znaczących 31 bitach, najbardziej znaczący bit ustawiając na 0, za wyjątkiem sytuacji, gdy zawiera on bit przeniesienia lub pożyczki.

2.16. Przesunięcia liczb o podwójnej długości

Załóżmy, że (x_1, x_0) jest parą 32-bitowych słów, którą mamy przesunąć w lewo lub w prawo, tak jakby były wartością 64-bitową, gdzie x_1 jest bardziej znaczącą połową. Załóżmy również, że (y_1, y_0) jest wynikiem, interpretowanym w ten sam sposób. Załóżmy również, że przesunięcie n jest liczbą z przedziału od 0 do 63. Załóżmy na koniec, że przesunięcia w maszynie są wykonywane modulo 64 lub więcej, to znaczy, że przesunięcie słowa o n z przedziału od 32 do 63 lub z przedziału od –32 do –1 zwróci słowo o wszystkich bitach równych zero, chyba że przesunięcie to jest przesunięciem w prawo ze znakiem. W tym przypadku wynik będzie zawierał 32 bity znaku oryginalnej wartości. Prezentowany kod nie będzie działał na maszynach z rodziny Intel x86, które posiadają przesunięcia modulo 32.

Przyjąwszy powyższe założenia operacja *przesunięcia w lewo liczb o podwójnej długości* (*shift left double*) może zostać zaimplementowana w następujący sposób:

$$y_1 \leftarrow x_1 << n \mid x_0 \overset{u}{>>} (32-n) \mid x_0 << (n-32)$$
$$y_0 \leftarrow x_0 << n$$

W pierwszym przypisaniu zastosowaliśmy *alternatywę*, nie *dodawanie*, ponieważ dzięki temu uzyskamy poprawny wynik w przypadku, gdy $n = 32$. Jeśli założymy, że $0 \le n \le 32$, można pominąć ostatni element pierwszego przypisania, dzięki czemu całe rozwiązanie wymaga zastosowania sześciu instrukcji.

Operację *przesunięcia w prawo liczby podwójnej długości bez znaku* (*shift right double unsigned*) można zrealizować w następujący sposób:

$$y_0 \leftarrow x_0 \overset{u}{>>} n \mid x_1 << (32-n) \mid x_1 \overset{u}{>>} (n-32)$$
$$y_1 \leftarrow x_1 \overset{u}{>>} n$$

Operacja *przesunięcia w prawo liczby podwójnej długości ze znakiem* (*shift right double signed*) jest nieco trudniejszym zadaniem z powodu niepożądanej propagacji bitu znaku w jednej z operacji. Możemy zastosować następujące rozwiązanie:

jeśli $n < 32$, to $\quad y_0 \leftarrow x_0 \overset{u}{>>} n \mid x_1 << (32-n)$

w przeciwnym wypadku: $\quad y_0 \leftarrow x_1 \overset{s}{>>} (n-32)$

$$y_0 \leftarrow x_1 \overset{s}{>>} n$$

W przypadku, gdy procesor obsługuje instrukcje *przesłania warunkowego* (ang. *conditional move*) powyższą formułę można zaimplementować w ośmiu instrukcjach bez potrzeby stosowania rozgałęzień. W przypadku, gdy nie są dostępne instrukcje przesłania warunkowego, operację można wykonać w dziesięciu instrukcjach, wykorzystując

znany pomysł maski bitowej z operacją *przesunięcia w prawo ze znakiem* o 31 pozycji. Maska ta ma na celu usunięcie niepożądanych bitów znaku powielonych w prawo w wyniku przesunięcia ze znakiem:

$$y_0 \leftarrow x_0 \overset{s}{>>} n \mid x_1 << (32-n) \mid [(x_1 \overset{s}{>>} (n-32)) \,\&\, ((32-n) \overset{s}{>>} 31)]$$

$$y_1 \leftarrow x_1 \overset{s}{>>} n$$

2.17. Operacje dodawania, odejmowania i wyznaczania wartości bezwzględnej na wartościach wielobajtowych

Niektóre aplikacje pracują na tablicach krótkich liczb całkowitych (z reguły są to bajty lub półsłowa) a przetwarzanie takich wartości jest z reguły szybsze, jeśli łączy się je w słowa. Dla celów demonstracyjnych w przykładach posłużymy się czteroelementowymi tablicami jednobajtowych wartości całkowitych, spakowanymi w słowo. Zastosowane techniki mogą jednak zostać zaadaptowane do innych wariantów, takich jak upakowanie jednej 12-bitowej liczby i dwóch liczb 10-bitowych w słowo itp. Techniki tego typu mają większą wartość na maszynach 64-bitowych, ponieważ na nich można równolegle wykonywać większą liczbę operacji na takich składowych słowa.

Dodawanie należy skonstruować w sposób, który uniemożliwi propagację przeniesienia do sąsiedniego pola (będącego inną liczbą). Można tego dokonać za pomocą dwuetapowej procedury.

1. Najpierw maskujemy najbardziej znaczący bit w każdym bajcie każdego z operandów i wykonujemy *dodawanie* (w ten sposób przeniesienia nie rozpropagują się przez ograniczenia pól).
2. Poprawiamy najbardziej znaczący bit każdego bajtu używając 1-bitowych *sum* odpowiednich bitów operandów oraz bitów przeniesienia na tych pozycjach.

Bit przeniesienia na pozycji najbardziej znaczącego bitu każdego z bajtów jest oczywiście uzyskiwany z najbardziej znaczącego bitu sumy wyliczonej w punkcie 1. Podobna metoda ma zastosowanie również w przypadku różnicy:

Dodawanie:

$$s \leftarrow (x \,\&\, \mathbf{0x7F7F7F7F}) + (y \,\&\, \mathbf{0x7F7F7F7F})$$

$$s \leftarrow ((x \oplus y) \,\&\, \mathbf{0x80808080}) \oplus s$$

Odejmowanie:

$$d \leftarrow (x \mid \mathbf{0x80808080}) - (y \,\&\, \mathbf{0x7F7F7F7F})$$

$$d \leftarrow ((x \oplus y) \mid \mathbf{0x7F7F7F7F}) \equiv d$$

Powyższe formuły na maszynach udostępniających pełny zestaw instrukcji logicznych wykonują się w ośmiu instrukcjach, wliczając w to załadowanie wartości bezpośrednich **0x7F7F7F7F** (można zamienić instrukcje *koniunkcji* i *alternatywy* z wartością **0x80808080** na, odpowiednio, instrukcje koniunkcji z zaprzeczeniem — *and not* oraz alternatywy z zaprzeczeniem — *or not* — z wartością **0x7F7F7F7F**).

W przypadku, gdy słowo jest podzielone na dwie części można zastosować prostszą technikę. Wykonujemy normalne dodawanie liczb 32-bitowych a następnie odejmujemy niepożądane przeniesienie. Na stronie 44 zauważyliśmy, że wyrażenie $(x + y) \oplus x \oplus y$ wyznacza przeniesienia na każdej z pozycji. Wykorzystując to spostrzeżenie oraz podobne, możemy skonstruować następujący kod, wykonujący w siedmiu instrukcjach dodawanie i odejmowanie dwóch półsłów modulo 2^{16}.

Dodawanie:

$$s \leftarrow x + y$$

$$x \leftarrow (s \oplus x \oplus y) \ \&\ \mathbf{0x00010000}$$

$$s \leftarrow s - c$$

Odejmowanie:

$$d \leftarrow x - y$$

$$b \leftarrow (d \oplus x \oplus y) \ \&\ \mathbf{0x00010000}$$

$$d \leftarrow d + b$$

Wyznaczenie *wartości bezwzględnej* wielobajtowych wartości można z łatwością zaimplementować za pomocą operacji uzupełnienia oraz dodania wartości 1 do każdego bajtu, zawierającego liczbę o wartości ujemnej (to znaczy takiego, którego najbardziej znaczący bit ma wartość 1). Następujący kod ustawia wartość każdego bajtu słowa y na wartość bezwzględną każdego bajtu słowa x (osiem instrukcji):

$a \leftarrow x \ \&\ \mathbf{0x80808080}$	// wyodrębniamy bity znaku
$b \leftarrow a \overset{u}{>>} 7$	// wartość 1 dla każdej liczby x ujemnej
$m \leftarrow (a - b) \mid a$	// 0xFF dla każdej liczby x ujemnej
$y \leftarrow (x \oplus m) + b$	// Uzupełnienie i dodanie 1 w przypadku wartości ujemnej

Trzeci wiersz można zastąpić następującym: $m \leftarrow a + a - b$. Dodanie wartości b w czwartym wierszu nie powoduje przeniesień na granicach bajtów, ponieważ wynik wyrażenia $x \oplus m$ posiada wartość 0 w najbardziej znaczącym bicie każdego bajtu.

2.18. Doz, Max oraz Min

Funkcja „doz", czyli „różnica lub zero" (ang. *difference or zero*) dla argumentów ze znakiem zdefiniowana jest w następujący sposób:

$$\mathrm{doz}(x, y) = \begin{cases} x - y, & x \geq y, \\ 0, & x < y \end{cases}$$

Funkcję tę nazwano „odejmowaniem pierwszego stopnia", ponieważ wynik jest równy zero w przypadku, gdy odjemnik jest większy od odjemnej. Funkcja ta jest pomocna w celu wyliczenia wartości funkcji max(x, y) oraz min(x, y). W tym miejscu należy zaznaczyć, że wartość funkcji „doz" może być liczbą ujemną. Dzieje się tak w przypadku, gdy operacja *odejmowania* spowoduje przepełnienie. Funkcja „doz" może zostać bezpośrednio zastosowana do implementacji funkcji IDIM języka Fortran, choć w tym języku w przypadku przepełnienia wyniki są z reguły niezdefiniowane.

Nie są znane dobre sposoby implementacji funkcji doz(x, y), max(x, y) oraz min(x, y) w sposób niewykorzystujący rozgałęzień i który można zastosować w większości procesorów. Prawdopodobnie najlepszy sposób implementacji funkcji dox(x, y) polega na wykorzystaniu jednej z formuł wyliczania predykatu nierówności $x < y$, przedstawionych na stronie 36. Na podstawie tego wyniku można wyliczyć wyniki funkcji max(x, y) oraz min(x, y):

$$d \leftarrow x - y$$

$$\mathrm{doz}(x, y) = d \,\&\, [(d \equiv ((x \oplus y) \,\&\, (d \oplus x)))\overset{s}{>>}31]$$

$$\max(x, y) = y + \mathrm{doz}(x, y)$$

$$\min(x, y) = x - \mathrm{doz}(x, y)$$

W przypadku, gdy maszyna udostępnia instrukcję *równoważności*, powyższe obliczenia wykorzystują siedem instrukcji w celu wyliczenia doz(x, y), w przeciwnym wypadku potrzeba ośmiu instrukcji. Wyliczenie max(x, y) oraz min(x, y) wymaga dodatkowo po jednej instrukcji.

Wersje powyższych funkcji dla argumentów bez znaku wyglądają następująco:

$$d \leftarrow x - y$$

$$\mathrm{dozu}(x, y) = d \,\&\, \neg[((\neg x \,\&\, y) \mid (x \equiv y) \,\&\, d))\overset{s}{>>}31]$$

$$\mathrm{maxu}(x, y) = y + \mathrm{dozu}(x, y)$$

$$\mathrm{minu}(x, y) = x - \mathrm{dozu}(x, y)$$

Komputer IBM RS/6000 oraz jego poprzednik, oznaczony kodem 801, obsługują instrukcję obliczającą wynik funkcji doz(x, y). Dzięki temu wyliczenie wyniku max(x, y) oraz min(x, y) dla całkowitoliczbowych argumentów ze znakiem wymaga na tych maszynach tylko dwóch instrukcji, a czasem funkcja ta jest użyteczna i do innych celów. Zaimplementowanie funkcji max(x, y) oraz min(x, y) w bezpośredni sposób jest bardziej kosztowne, chyba że maszyna udostępniałaby możliwość przekazania wartości z rejestru wyjściowego z powrotem na wejście z pominięciem ALU (jednostki arytmetyczno-logicznej).

Maszyny udostępniające instrukcję *warunkowego przesłania wartości* (ang. *conditional move*) pozwalają na implementację funkcji max(x, y) oraz min(x, y) w sposób destrukcyjny[2] w następujący sposób:

```
cmplt  z,x,y      Jeśli x < y ustawiamy z = 1, w przeciwnym wypadku z = 0
movne  x,z,y      Jeśli z jest różne od zera, ustawiamy x = y
```

2.19. Wymiana wartości między rejestrami

Istnieje bardzo stara sztuczka umożliwiająca wymianę zawartości dwóch rejestrów bez potrzeby wykorzystania trzeciego [IBM]:

$$x \leftarrow x \oplus y$$

$$y \leftarrow y \oplus x$$

$$x \leftarrow x \oplus y$$

Powyższa sztuczka działa doskonale na maszynie dwuadresowej. Działa również w przypadku, gdy operację $\oplus$ zastąpimy operacją $\equiv$ (która jest dopełnieniem różnicy symetrycznej) i może być zastosowana z operacjami *dodawania* i *odejmowania* na wiele różnych sposobów:

$$\begin{array}{lll} x \leftarrow x + y & x \leftarrow x - y & x \leftarrow y - x \\ y \leftarrow x - y & y \leftarrow y + x & y \leftarrow y - x \\ x \leftarrow x - y & x \leftarrow y - x & x \leftarrow x + y \end{array}$$

Niestety każde z tych zastosowań wykorzystuje instrukcję, która nie może zostać zastosowana na maszynach dwuadresowych, chyba że obsługiwana jest instrukcja „odwróconego odejmowania" (ang. *reverse subtract*).

Ta niepozorna sztuczka może znaleźć zastosowanie w podwójnym buforowaniu, gdy naprzemian są wymieniane dwa wskaźniki. Pierwsza instrukcja może zostać przesunięta poza pętlę wykonującą wymianę (lecz wymaga to zastosowania dodatkowego rejestru, zatem zaprzecza to idei wymiany zawartości rejestrów bez konieczności użycia dodatkowego):

$$\begin{array}{rl} \text{Na zewnątrz pętli:} & t \leftarrow x \oplus y \\ \text{Wewnątrz pętli:} & x \leftarrow x \oplus t \\ & y \leftarrow y \oplus t \end{array}$$

[2] Operacja destrukcyjna powoduje zniszczenie (nadpisanie) oryginalnej wartości jednego z jej argumentów.

Wymiana odpowiadających sobie pól rejestrów

Kolejny problem polega na wymianie zawartości rejestrów x i y z wykorzystaniem maski $\boldsymbol{m}$. Wymieniamy i-ty bit rejestrów w przypadku, gdy $\boldsymbol{m}_i = 1$. Natomiast gdy $\boldsymbol{m}_i = 0$, odpowiednie bity rejestrów x i y pozostają bez zmian. Nie wykorzystujemy przesunięć a bloki jedynek w masce $\boldsymbol{m}$ nie muszą być ciągłe. Bezpośrednia metoda rozwiązania tego zadania może być następująca:

$$x' \leftarrow (x \,\&\, \overline{\boldsymbol{m}}) \mid (y \,\&\, \boldsymbol{m})$$
$$y \leftarrow (y \,\&\, \overline{\boldsymbol{m}}) \mid (x \,\&\, \boldsymbol{m})$$
$$x \leftarrow x'$$

Powyższa procedura wykorzystuje siedem instrukcji pod warunkiem wykorzystania wartości tymczasowych w czterech instrukcjach *koniunkcji* i przy możliwości załadowania wartości $\boldsymbol{m}$ oraz $\overline{\boldsymbol{m}}$ za pomocą pojedynczej instrukcji, jak również dostępności instrukcji *koniunkcji z zaprzeczeniem* (ang. *and not*). W przypadku, gdy procesor jest w stanie wykonać cztery jednoczesne, niezależne instrukcje *koniunkcji*, procedura ta wykona się w trzech cyklach.

Poniżej, w kolumnie (a) zaprezentowano metodę lepszą dla procesorów bez superskalarności (pięć instrukcji, choć komputery z nieograniczoną możliwością jednoczesnego wykonywania niezależnych instrukcji potrzebują do jej wykonania czterech instrukcji). Metoda ta powstała przez modyfikację techniki trzech różnic symetrycznych.

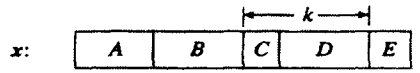
(a)	(b)	(c)
$x \leftarrow x \oplus y$	$x \leftarrow x \equiv y$	$t \leftarrow (x \oplus y) \,\&\, \boldsymbol{m}$
$y \leftarrow y \oplus (x \,\&\, \boldsymbol{m})$	$y \leftarrow y \equiv (x \mid \overline{\boldsymbol{m}})$	$x \leftarrow x \oplus t$
$x \leftarrow x \oplus y$	$x \leftarrow x \equiv y$	$y \leftarrow y \oplus t$

Metoda zaprezentowana w kolumnie (b) polega na wykorzystaniu tej samej wymiany co metoda w kolumnie (a), z tą różnicą, że metoda (b) jest użyteczna w przypadku, gdy wartość $\boldsymbol{m}$ nie mieści się w polu wartości bezpośredniej, natomiast $\overline{\boldsymbol{m}}$ mieści się a maszyna udostępnia instrukcję *równoważności* (ang. *equivalence*).

Jeszcze inna metoda [GLS1] jest zademonstrowana w kolumnie (c). Metoda ta również wymaga zastosowania pięciu instrukcji (zakładając jedną instrukcje na załadowanie wartości $\boldsymbol{m}$ do rejestru) lecz na superskalarnej maszynie z odpowiednim poziomem równoległości może wykonać się w trzech cyklach.

Wymiana dwóch pól w tym samym rejestrze

Załóżmy, że rejestr x ma dwa pola (jednakowej długości) które należy zamienić miejscami bez naruszania pozostałych bitów w rejestrze. To znaczy chcemy zamienić miejscami pola B i D bez naruszania zawartości pól A, C i E (patrz rysunek). Pola są oddalone od siebie o odległość k.



Prosta metoda wykorzystuje przesunięcie pól D oraz B na nowe pozycje a następnie połączenie wartości za pomocą *koniunkcji* i *alternatywy* w następujący sposób:

$$t_1 = (x \,\&\, \boldsymbol{m}) << k$$

$$t_2 = (x \overset{u}{>>} k) \,\&\, \boldsymbol{m}$$

$$x' = (x \,\&\, \boldsymbol{m}') \mid t_1 \mid t_2$$

W powyższym kodzie $\boldsymbol{m}$ oznacza maskę pola D (wartości 1 określają obszar pola D, wartości 0 pozostałe bity), natomiast $\boldsymbol{m}'$ określa maskę pól A, C i E. Kod ten wymaga zastosowania dziewięciu instrukcji i czterech cykli na maszynie o nieograniczonym poziomie równoległości niezależnych operacji, przy założeniu dwóch instrukcji na załadowanie wartości masek.

Alternatywna metoda wymaga zastosowania tylko siedmiu instrukcji i wykonuje się w pięciu cyklach przy tych samych założeniach. Algorytm ten został zademonstrowany przez [GLS1]. Jest on podobny do kodu w kolumnie (c) w tabeli na stronie 57. Ponownie, $\boldsymbol{m}$ jest maską separującą pole D.

$$t_1 = [x \oplus (x \overset{u}{>>} k)] \,\&\, \boldsymbol{m}$$

$$t_2 = t_1 << k$$

$$x' = x \oplus t_1 \oplus t_2$$

Pomysł polega na tym, że t_1 zawiera $B \oplus D$ w miejscu pola D (w pozostałych obszarach zawiera same zera), natomiast t_2 zawiera $B \oplus D$ w miejscu pola B. Kod ten, jak również wcześniejszy, działa poprawnie w przypadku, gdy B i D są polami nieciągłymi, to znaczy jedynki w masce $\boldsymbol{m}$ nie stanowią ciągłego bloku.

Wymiana warunkowa

Metody wymiany wartości opisane wyżej, opierające się na operacji *różnicy symetrycznej*, w przypadku, gdy $\boldsymbol{m} = 0$, nie wykonają żadnej operacji. Fakt ten można wykorzystać do warunkowego wykonania wymiany całych rejestrów (lub odpowiadających sobie pól dwóch rejestrów lub dwóch pól tego samego rejestru), ustawiając wartość wszystkich bitów maski $\boldsymbol{m}$ na jedynki w przypadku spełnienia pewnego warunku c, lub na wartość samych zer w przeciwnym wypadku. Dzięki temu uzyskamy kod bez rozgałęzień, pod warunkiem, że wartość $\boldsymbol{m}$ możemy ustawić bez zastosowania rozgałęzień.

2.20. Wymiana dwóch lub większej liczby wartości

Załóżmy, że zmienna x może przyjmować dwie wartości: a lub b i że chcemy przypisać zmiennej x drugą z możliwych wartości. Otrzymany kod powinien być niezależny od wartości a i b. Na przykład w kompilatorze zmienna x mogłaby być kodem operacji (ang. *opcode*), o której wiadomo, że może być *rozgałęzieniem w przypadku wyniku prawda* (ang. *branch true*) lub *rozgałęzieniem w przypadku wyniku fałsz* i niezależnie od tego, jaką ma aktualnie wartość, chcemy zmienić ją na drugą z możliwych. Wartości możliwych kodów operacji nie są wstępnie zdefiniowane i mogą zależeć na przykład od deklaracji `#define` preprocesora języka C lub od deklaracji wyliczenia `enum` w pliku nagłówkowym.

Najprostszym kodem wykonującym taką zamianę będzie następujący:

```
if (x == a) x = b;
else x = a;
```

Inny sposób stanowi dość częsty zabieg, stosowany przez programistów języka C:

```
x = x == a ? b : a;
```

O wiele lepszy sposób (w każdym razie bardziej wydajny) polega na wykonaniu jednej z następujących operacji:

$$x \leftarrow a + b - x$$

$$x \leftarrow a \oplus b \oplus x$$

W przypadku, gdy a oraz b są stałymi, każda z powyższych formuł wymaga dwóch instrukcji z podstawowego zestawu RISC. Oczywiście przepełnienie wywołane instrukcją $a + b$ można bezpiecznie zignorować.

Narzuca się następujące pytanie: czy można znaleźć wydajny sposób wymiany trzech wartości? To znaczy, załóżmy, że mamy trzy dowolne, różne wartości stałych a, b i c i poszukujemy prostej do wyliczenia funkcji f spełniającej następujące założenia:

$$f(a) = b$$

$$f(b) = c$$

$$f(c) = a$$

Być może interesującym spostrzeżeniem będzie możliwość znalezienia wielomianu takiej funkcji f. W przypadku trzech stałych wielomian ten będzie miał następującą postać:

$$f(x) = \frac{(x-a)(x-b)}{(c-a)(c-b)}a + \frac{(x-b)(x-c)}{(a-b)(a-c)}b + \frac{(x-c)(x-a)}{(b-c)(b-a)}c \qquad (4)$$

Pomysł polega na tym, że w przypadku, gdy $x = a$ znikają pierwszy i ostatni składnik wielomianu, drugi natomiast zostaje uproszczony do b i tak dalej. Przetworzenie wielomianu wymaga 14 operacji arytmetycznych i istnieje możliwość przepełnienia rozmiaru słowa. Jest to jednak wielomian kwadratowy, po zapisaniu go w standardowej postaci wielomianu i wyliczeniu za pomocą schematu Hornera[3] będziemy potrzebowali pięciu instrukcji arytmetycznych (cztery w celu wyliczenia wielomianu kwadratowego z współczynnikami całkowitymi plus jedno dzielenie). Po odpowiednim przekształceniu formuły (4) uzyskamy następującą:

$$\begin{aligned} f(x) = \frac{1}{(a-b)(a-c)(b-c)} \{ & [(a-b)a+(b-c)b+(c-a)c]x^2 \\ & +[(a-b)b^2+(b-c)c^2+(c-a)a^2]x \\ & +[(a-b)a^2b+(b-c)b^2c+(c-a)ac^2]\}. \end{aligned}$$

Wielomian ten staje się zbyt skomplikowany, aby nadal nas interesować, jak również traci walory praktyczne.

Inna metoda jest podobna do równania (4) o tyle, że zostaje zachowany tylko jeden z trzech składników:

$$f(x) = ((-(x = c)) \,\&\, a) + ((-(x = a)) \,\&\, b) + ((-(x = b)) \,\&\, c).$$

Powyższa formuła wykorzystuje 11 instrukcji pod warunkiem udostępnienia przez maszynę predykatu równości, nie licząc ładowania stałych do rejestrów. Ponieważ dwa dodawania stanowią sumowanie dwóch wartości 0 z wartościami niezerowymi, można je zastąpić instrukcjami *alternatywy* lub *różnicy symetrycznej*.

Formułę te można uprościć za pomocą wyliczenia $a - c$ oraz $b - c$ a następnie wykorzystania jednej z następujących formuł [GLS1]:

$$f(x) = ((-x = c)) \,\&\, (a - c)) + ((-(x = a)) \,\&\, (b - c)) + c$$

$$f(x) = ((-(x = c)) \,\&\, (a \oplus c)) \oplus ((-(x = a)) \,\&\, (b \oplus c)) \oplus c.$$

Każda z tych formuł wykorzystuje osiem instrukcji. Na większości maszyn jednak nie da się znaleźć lepszego algorytmu rozwiązującego nasz problem od podanego poniżej krótkiego kodu w języku C. Algorytm ten wykona się w czterech do ośmiu instrukcji w przypadku niewielkich wartości a, b i c:

```
if (x == a) x = b;
else if (x == b) x = c;
else x = a;
```

[3] Reguła Hornera polega na wyciąganiu przed nawias wartości x. Na przykład pozwala na przekształcenie wielomianu $ax^4 + bx^3 + cx^2 + dx + e$ na postać $x(x(x(ax + b) + c) + d) + e$. W przypadku wielomianu stopnia n reguła Hornera wykorzystuje n mnożeń i n dodawań i bardzo dobrze nadaje się do zastosowania instrukcji mnożenia z dodawaniem (*multiply-add*).

Podążając dalej tym tropem można z łatwością wyprowadzić metodę pozwalającą na wykonanie wymiany trzech wartości za pomocą ośmiu instrukcji bez rozgałęzień, nie wykorzystując również predykatu równości [GLS1].

Z faktu, że a, b oraz c są różnymi liczbami można wywnioskować, iż istnieją takie dwie pozycje bitowe n_1 oraz n_2, które nie mają identycznej wartości we wszystkich liczbach a, b i c oraz liczba, która różni się od pozostałych na pozycji bitu n_1 nie jest liczbą, która różni się od pozostałych na pozycji bitu n_2. Sytuację tę przedstawiamy poniżej w zapisie dwójkowym:

$$\begin{array}{cccccc} 1 & 0 & 1 & 0 & 1 & c \\ 1 & 1 & 1 & 1 & 1 & a \\ 1 & 0 & 1 & 0 & 0 & b \\ & n_1 & & & n_2 & \end{array}$$

Przemianujmy zmienne a, b i c w ten sposób, że a będzie liczbą różniącą się od pozostałych na pozycji n_1, natomiast b będzie różniło się od pozostałych na pozycji n_2. Mamy dwie możliwe kombinacje wartości na pozycji n_1, to znaczy $(a_{n_1}, b_{n_1}, c_{n_1}) = (0,1,1) \text{ lub } (1,0,0)$. Podobnie istnieją dwie możliwe kombinacje wartości na pozycji n_2: $(a_{n_2}, b_{n_2}, c_{n_2}) = (0,1,0) \text{ lub } (1,0,1)$. W sumie daje to cztery możliwe kombinacje wartości, które przedstawiamy poniżej:

Przypadek 1 $(a_{n_1}, b_{n_1}, c_{n_1}) = (0,1,1), (a_{n_2}, b_{n_2}, c_{n_2}) = (0,1,0)$:

$$f(x) = x_{n_1} * (a - b) + x_{n_2} * (c - a) + b$$

Przypadek 2 $(a_{n_1}, b_{n_1}, c_{n_1}) = (0,1,1), (a_{n_2}, b_{n_2}, c_{n_2}) = (1,0,1)$:

$$f(x) = x_{n_1} * (a - b) + x_{n_2} * (a - c) + (b + c - a)$$

Przypadek 3 $(a_{n_1}, b_{n_1}, c_{n_1}) = (1,0,0), (a_{n_2}, b_{n_2}, c_{n_2}) = (0,1,0)$:

$$f(x) = x_{n_1} * (b - a) + x_{n_2} * (c - a) + a$$

Przypadek 4 $(a_{n_1}, b_{n_1}, c_{n_1}) = (1,0,0), (a_{n_2}, b_{n_2}, c_{n_2}) = (1,0,1)$:

$$f(x) = x_{n_1} * (b - a) + x_{n_2} * (a - c) + c$$

W powyższych formułach lewym operandem każdego mnożenia jest pojedynczy bit. Mnożenie przez 0 lub 1 może zostać zastąpione operacją *koniunkcji* bitowej odpowiednio z wartością złożoną z samych zer lub z samych jedynek. Dzięki temu formuły te możemy zapisać w następujący sposób:

$$f(x) = ((x << (31 - n_1)) \overset{s}{>>} 31) \,\&\, (a - b) + ((x << (31 - n_2)) \overset{s}{>>} 31) \,\&\, (c - a) + b$$

Z faktu, iż wszystkie wartości za wyjątkiem x są stałymi wynika, że powyższą formułę można wyliczyć za pomocą ośmiu instrukcji z podstawowego zestawu RISC. W tym przypadku ponownie dodawania i odejmowania można zastąpić operacjami *różnicy symetrycznej*.

Powyższą koncepcję można rozwinąć do procedury cyklicznej pomiędzy czterema lub większą liczbą stałych. Osią koncepcji jest odnalezienie pozycji n_1, n_2, ..., na których bity jednoznacznie identyfikują stałe. Dla czterech stałych wystarczy wyszukać trzy pozycje bitowe. Dlatego dla czterech stałych należy wyliczyć następujące równanie dla wartości s, t, u oraz v (to znaczy należy rozwiązać układ czterech równań liniowych, gdzie $f(x)$ jest równe a, b, c oraz d a współczynniki x_{n_i} są równe 0 lub 1):

$$f(x) = x_{n_1} s + x_{n_2} t + x_{n_3} u + v$$

W przypadku, gdy cztery stałe są jednoznacznie zidentyfikowane na dwóch pozycjach bitowych, należy rozwiązać następujące równanie:

$$f(x) = x_{n_1} s + x_{n_2} t + x_{n_1} x_{n_2} u + v .$$

Rozdział 3.

Ograniczenia potęg dwójki

3.1. Zaokrąglanie do wielokrotności znanych potęg liczby 2

Zaokrąglanie liczby całkowitej bez znaku na przykład do następnej mniejszej wielokrotności liczby 8 jest banalnie proste: można tego dokonać za pomocą operacji x & **–8**. Alternatywnym rozwiązaniem tego problemu może być $(x \overset{u}{>>} 3) << 3$. Formuły te działają również dla liczb ze znakiem, pod warunkiem, że przez „zaokrąglenie w dół" rozumiemy zaokrąglenie w kierunku ujemnym, to znaczy (–**37**) & (–**8**) = –**40**.

Zaokrąglanie w górę jest prawie tak samo łatwe. Na przykład liczba całkowita bez znaku x może zostać zaokrąglona w górę do następnej wielokrotności liczby 8 za pomocą następującej formuły:

$$(x + 7) \ \& \ -8.$$

Alternatywnym sposobem jest następującą formuła:

$$x + (-x \ \& \ 7).$$

Wyrażenia te są poprawne również dla liczb całkowitych ze znakiem, pod warunkiem, że przez „zaokrąglenie w górę" rozumiemy zaokrąglenie w kierunku dodatnim. Drugi składnik w drugiej z proponowanych formuł może dodatkowo być użyteczny w przypadku, gdy chcemy wiedzieć ile musimy dodać do x, aby wynik był wielokrotnością liczby 8 [Gold].

W celu zaokrąglenia w kierunku 0 liczby całkowitej ze znakiem do najbliższej wielokrotności liczby 8, można połączyć dwa powyższe wyrażenia:

$$t \leftarrow (x \overset{s}{>>} 31) \ \& \ 7$$

$$(x + t) \ \& \ -8$$

Alternatywą dla pierwszego wyrażenia może być $t \leftarrow (x \overset{s}{>>} 2) \overset{u}{>>} 29$, co może być użyteczne w przypadku, gdy procesor nie obsługuje instrukcji *koniunkcji z wartością bezpośrednią* (ang. *and immediate*) lub gdy stała ma zbyt dużą wartość, niemieszczącą się w polu wartości bezpośredniej.

Czasem współczynnik zaokrąglenia podawany jest jako $\log_2$ *wielkości wyrównania* (ang. *alignment amount*), to znaczy wartość 3 oznacza na przykład zaokrąglenie do 8. W tym przypadku można zastosować poniższą formułę, gdzie $k = \log_2$(wielkość wyrównania).

zaokrąglenie w dół: $x \,\&\, ((-1) << k)$

$(x \overset{u}{>>} k) << k$

zaokrąglenie w górę: $t \leftarrow (1 << k) - 1;\ (x + t) \,\&\, \neg t$

$t \leftarrow (-1) << k;\ (x - t - 1) \,\&\, t$

3.2. Zaokrąglanie w górę lub w dół do następnej potęgi liczby 2

Zdefiniujemy dwie funkcje podobne do funkcji **podłoga** (ang. *floor*) oraz **sufit** (ang. *ceiling*), lecz z tą różnicą, że mają za zadanie dokonać zaokrąglenia do następnej potęgi liczby 2, zamiast do kolejnej liczby całkowitej. W sposób matematycznie funkcje te zostaną zdefiniowane następująco:

$$\mathrm{flp2}(x) \begin{cases} \text{niezdefiniowane}, & x < 0 \\ 0, & x = 0 \\ 2^{\lfloor \log_2 x \rfloor}, & x > 0 \end{cases}$$

$$\mathrm{clp2}(x) \begin{cases} \text{niezdefiniowane}, & x < 0 \\ 0, & x = 0 \\ 2^{\lceil \log_2 x \rceil}, & x > 0 \end{cases}$$

Początkowe litery nazw funkcji mają sugerować nazwy *floor* (podłoga) oraz *ceiling* (sufit). Stąd flp2(x) jest największą potęgą liczby 2, która $\le x$, natomiast clp2(x) jest najmniejszą potęgą dwójki, która $\ge x$. Definicje te mają sens również dla liczb niecałkowitych (np. flp2(0,1) = 0,0625). Funkcje te spełniają szereg relacji podobnych do tych dotyczących prawdziwych funkcji *podłoga* oraz *sufit*. Kilka z nich przedstawiamy poniżej (n jest liczbą całkowitą):

$\lfloor x \rfloor = \lceil x \rceil$ iff x jest liczbą całkowitą	$\mathrm{flp2}(x) = \mathrm{clp2}(x)$ iff x jest potęgą **2** lub $x = \mathbf{0}$
$\lfloor x + n \rfloor = \lfloor x \rfloor + n$	$\mathrm{flp2}(\mathbf{2}^n x) = \mathbf{2}^n \mathrm{flp2}(x)$
$\lceil x \rceil = -\lfloor -x \rfloor$	$\mathrm{clp2}(x) = \mathbf{1} / \mathrm{flp2}(\mathbf{1} / x),\ x \ne \mathbf{0}$

W naszych obliczeniach będziemy mieli do czynienia z liczbami całkowitymi i zakładamy, że są to liczby bez znaku, zatem funkcje te są dobrze zdefiniowane dla wszystkich wartości x. Wymogiem jest to, aby wyliczony wynik był prawidłową wartością modulo 2^{32} (to znaczy, wartość funkcji clp2(x) musi być równa 0 dla $x > 2^{31}$). Poniżej przedstawiamy rozpisane wartości tych funkcji dla wybranych wartości x:

x	flp2(x)	clp2(x)
0	0	0
1	1	1
2	2	2
3	2	4
4	4	4
5	4	8
...	...	...
$2^{31}-1$	2^{30}	2^{31}
2^{31}	2^{31}	2^{31}
$2^{31}+1$	2^{31}	0
...	...	...
$2^{32}-1$	2^{31}	0

Funkcje „flp2" oraz „clp2" są związane ze sobą rozpisanymi poniżej zależnościami. Z uwzględnieniem zastrzeżeń, zależności te mogą zostać użyte w celu wyliczenia wartości jednej z tych funkcji na podstawie wartości drugiej:

$$\begin{aligned}
\text{clp2}(x) &= 2\text{flp2}(x-\mathbf{1}), & x \neq \mathbf{1},\\
&= \text{flp2}(\mathbf{2}x-\mathbf{1}) & 1 \leq x \leq \mathbf{2}^{31},\\
\text{flp2}(x) &= \text{clp2}(x \overset{u}{\div} \mathbf{2}+\mathbf{1}), & x \neq \mathbf{0},\\
&= \text{clp2}(x+\mathbf{1}) \overset{u}{\div} \mathbf{2}, & x < \mathbf{2}^{31}.
\end{aligned}$$

Funkcje zaokrąglenia w górę i w dół można z łatwością zdefiniować za pomocą instrukcji „nlz" (*number of leading zeros* — liczba zer wiodących), co demonstrujemy poniżej. Jednakże w celu generowania prawidłowego wyniku dla $x = \mathbf{0}$ oraz $x > \mathbf{2}^{31}$ komputer musi mieć instrukcje przesunięcia zdefiniowane w ten sposób, aby generowały wartość 0 dla wartości przesunięcia równych –1, 32 oraz 63. Wiele maszyn (np. PowerPC) ma zaimplementowane przesunięcia modulo 64, które działają właśnie w ten sposób. W przypadku, gdy mamy do czynienia z przesunięciem o –1 wystarczy, gdy procesor wykona przesunięcie o 1 w przeciwnym kierunku (to znaczy przesunięcie w lewo o –1 spowoduje wykonanie przesunięcia w prawo o 1).

$$\begin{aligned}
\text{flp2}(x) &= \mathbf{1} << (\mathbf{31} - \text{nlz}(x))\\
&= \mathbf{1} << (\text{nlz}(x) \oplus \mathbf{31})\\
&= \mathbf{0x80000000} \overset{u}{>>} \text{nlz}(x)
\end{aligned}$$

$$\begin{aligned} \mathrm{clp2}(x) &= \mathbf{1} << (\mathbf{32} - \mathrm{nlz}(x - \mathbf{1})) \\ &= \mathbf{0x80000000}^{u} >> (\mathrm{nlz}(x - \mathbf{1}) - \mathbf{1}) \end{aligned}$$

Zaokrąglanie w dół

Listing 3.1 przedstawia algorytm zaokrąglania w dół niewykorzystujący rozgałęzień, który może być przydatny w przypadku, gdy nie jest dostępna instrukcja obliczająca liczbę wiodących zer (ang. *number of leading zeros*). Algorytm ten wykorzystuje propagację lewostronnej jedynki w prawo i wykonuje się w 12 instrukcjach.

Listing 3.1. *Największa potęga 2 mniejsza lub równa x, bez rozgałęzień*

```
unsigned flp2(unsigned x) {
   x = x | (x >> 1);
   x = x | (x >> 2);
   x = x | (x >> 4);
   x = x | (x >> 8);
   x = x | (x >>16);
   return x - (x >> 1);
}
```

Listing 3.2 przedstawia proste pętle wykonujące to samo zadanie. Wszystkie użyte zmienne są liczbami całkowitymi bez znaku. Druga z pętli wyłącza prawostronną jedynkę z liczby x do momentu, gdy $x = \mathbf{0}$ a wtedy zwraca poprzednią wartość liczby x.

Listing 3.2.a *Największa potęga 2 mniejsza lub równa x, prosta pętla*

```
y = 0x80000000;
while (y > x)
   y = y >> 1;
return y;
```

Listing 3.2.b *Największa potęga 2 mniejsza lub równa x, prosta pętla*

```
do {
   y = x;
   x = x & (x - 1);
} while (x != 0);
return y
```

Pętla *a* wykonuje się w $4\mathrm{nlz}(x) + 3$ instrukcjach. Pętla *b* dla $x \neq 0$ wykonuje się $4\mathrm{pop}(x)$ instrukcjach[1], pod warunkiem, że porównanie z liczbą 0 ma zerowy koszt.

Zaokrąglanie w górę

Sztuczka z propagacją bitu w prawo stanowi dobry algorytm służący do zaokrąglania liczby w górę do następnej potęgi liczby 2. Algorytm ten jest przedstawiony na listingu 3.3. Wykonuje się w 12 instrukcjach.

[1] $\mathrm{pop}(x)$ określa liczbę jedynek w liczbie x.

Listing 3.3. *Najmniejsza potęga liczby 2 większa lub równa liczbie x*

```
unsigned clp2(unsigned x) {
   x = x - 1;
   x = x | (x >> 1);
   x = x | (x >> 2);
   x = x | (x >> 4);
   x = x | (x >> 8);
   x = x | (x >>16);
   return x + 1;
}
```

Próba rozwiązania tego problemu za pomocą pętli nie daje najlepszych wyników:

```
y = 1;
while (y < x)      // porównanie bez znaku
   y = 2*y;
return y;
```

Powyższy kod zwraca 1 w przypadku, gdy $x = 0$, co najprawdopodobniej nie jest pożądanym zachowaniem, natomiast dla $x > 2^{31}$ powoduje niemożność opuszczenia pętli. Pętla ta wykonuje się w $4n + 3$ instrukcjach, gdzie n jest potęgą liczby 2 zwróconej liczby całkowitej. Zatem kod ten jest wolniejszy od kodu pozbawionego rozgałęzień już dla $n \geq 3$ ($x \geq 8$).

3.3. Wykrywanie przekroczenia ograniczeń potęgi dwójki

Załóżmy, że pamięć jest podzielona na bloki o rozmiarze określonym potęgą dwójki, rozpoczynające się od adresu 0. Bloki mogą być słowami, dwusłowami, stronami itd. Następnie, założywszy, że jest znany adres początkowy a oraz długość l musimy sprawdzić, czy adres z przedziału od a do $a + l - 1$, $l \geq 2$ przekracza ograniczenie bloku. Wartości a oraz l są liczbami bez znaku i dopuszczalne są wszelkie wartości mieszczące się w rejestrze.

Jeśli $l = 0$ lub 1, przekroczenie zakresu nie następuje, niezależnie od wartości a. W przypadku, gdy l przekracza rozmiar bloku, następuje przekroczenie zakresu, niezależnie od wartości a. Dla bardzo dużych wartości l (możliwe jest zawinięcie wartości) przekroczenie ograniczenia może nastąpić nawet w przypadku, gdy pierwszy i ostatni bajt zakresu znajdują się w tym samym bloku.

W systemie IBM System/370 istnieje zdumiewająco prosty sposób wykrywania przekroczeń ograniczeń potęgi dwójki [CIS]. Metoda ta jest przedstawiona poniżej dla bloków pamięci o rozmiarze 4096 bajtów (często stosowany rozmiar strony pamięci).

```
O   RA,=A(-4096)
ALR RA,RL
BO  CROSSES
```

Pierwsza instrukcja wylicza alternatywę logiczną rejestru RA (zawierającego początkowy adres a) oraz liczby 0xFFFFF000. Druga instrukcja dodaje rozmiar i ustawia 2-bitowy kod warunkowy procesora. Dla instrukcji sumy logicznej pierwszy bit kodu warunku jest ustawiany na wartość 1 w przypadku, gdy wystąpiło przeniesienie, drugi bit przyjmuje wartość 1 w przypadku, gdy 32-bitowy rejestr ma wartość różną od zera. Ostatnia z instrukcji dokonuje rozgałęzienia w przypadku, gdy obydwa bity mają wartość 1. U celu rozgałęzienia rejestr RA będzie zawierać wielkość, o którą sprawdzany rozmiar wykracza poza pierwszą stronę (jest to dodatkowa funkcja, wykraczająca poza nasze oczekiwania).

Jeśli na przykład $a = 0$ oraz $l = 4096$, wtedy wystąpi przeniesienie, lecz rejestr wyniku ma wartość 0, zatem program prawidłowo nie wykona rozgałęzienia do miejsca oznaczonego etykietą CROSSES.

Zastanówmy się nad sposobem adaptacji tej metody do maszyn z rodziny RISC, które z reguły nie posiadają instrukcji rozgałęzienia w sytuacji przeniesienia i niezerowej wartości rejestru. W celu uproszczenia notacji przyjmijmy rozmiar bloku równy 8. Metoda [CJS] powoduje rozgałęzienie do etykiety CROSSES w przypadku, gdy następuje przeniesienie ($(a \mid -8) + l \geq 2^{32}$) oraz rejestr wyniku ma niezerową wartość ($(a \mid -8) + l \neq 2^{32}$). Obydwa warunki są równoważne następującemu predykatowi:

$$(a \mid -\mathbf{8}) + l > \mathbf{2}^{32}.$$

To natomiast jest równoważne odczytaniu przeniesienia w wyrażeniu $((a \mid -8) - 1) + l$. W przypadku, gdy maszyna obsługuje instrukcję *branch on carry* (rozgałęzienie w przypadku przeniesienia) rozwiązanie będzie wymagać pięciu instrukcji, wliczając w to załadowanie stałej -8.

Jeśli maszyna nie obsługuje instrukcji *branch on carry*, możemy wykorzystać fakt, że przeniesienie występuje w działaniu $x + y$, wtedy i tylko wtedy, gdy $\neg x \overset{u}{<} y$ (zob. *Dodawanie i odejmowanie bez znaku* w podrozdziale 2.12.)

$$\neg((a \mid -\mathbf{8}) - \mathbf{1}) \overset{u}{<} l$$

Wykorzystując własności, takie jak $\neg(x - 1) = -x$ możemy wyprowadzić następujące wyrażenia predykatu przekroczenia ograniczenia:

$$-(a \mid -\mathbf{8}) \overset{u}{<} l$$

$$\neg(a \mid -\mathbf{8}) + \mathbf{1} \overset{u}{<} l$$

$$(\neg a \,\&\, \mathbf{7}) + \mathbf{1} \overset{u}{<} l$$

Powyższe wyrażenia mogą być wyliczone za pomocą pięciu do sześciu instrukcji na większości procesorów rodziny RISC.

Kolejne spostrzeżenie prowadzi do następującej zależności, związanej z przekroczeniem 8-bajtowego ograniczenia:

$$(a \,\&\, \mathbf{7}) + l - \mathbf{1} \geq \mathbf{8}.$$

Niestety powyższe wyrażenie nie nadaje się do bezpośredniego wyliczenia z powodu możliwości wystąpienia przepełnienia (w przypadku, gdy l jest bardzo duże). Można je jednak przebudować do postaci $8 - (a \& 7) < l$, które może być bezpośrednio wyliczone na komputerze (nie ma obawy o przepełnienie). Stąd można wyprowadzić wyrażenie:

$$\mathbf{8} - (a \& \mathbf{7}) \overset{u}{<} \boldsymbol{l}\,.$$

Powyższe wyrażenie może zostać wyliczone w pięciu instrukcjach na większości komputerów klasy RISC (czterech w przypadku, gdy procesor obsługuje instrukcję *subtract from immediate* — odejmowanie wartości bezpośredniej). W przypadku, gdy wystąpi przekroczenie ograniczenia, rozmiar przekroczenia można wyliczyć z formuły $\boldsymbol{l} - (\mathbf{8} - (\boldsymbol{a} \& \mathbf{7}))$, którą można wyliczyć za pomocą dodatkowej instrukcji *odejmowania*.

Rozdział 4.

Ograniczenia arytmetyczne

4.1. Kontrola ograniczeń liczb całkowitych

Przez kontrolę ograniczeń liczby x rozumiemy sprawdzenie typu:

$$a \leq x \leq b.$$

Po pierwsze załóżmy, że wszystkie wartości są liczbami całkowitymi ze znakiem.

Ważnym zastosowaniem kontroli ograniczeń jest kontrola poprawności indeksów tabel. Na przykład załóżmy, że wykorzystujemy jednowymiarową tablicę A o indeksach z przedziału od 1 do 10. W celu określenia odwołania $A(i)$ kompilator mógłby wygenerować kod, który skontrolowałby poprawność indeksu i:

$$1 \leq i \leq 10.$$

W przypadku, gdy wartość i nie spełnia warunku, mogłoby być wywoływane odpowiednie *rozgałęzienie*. W niniejszym podrozdziale zostanie zademonstrowany sposób kontroli zakresu za pomocą pojedynczego porównania, zastępując je odpowiednią równoważnością, na przykład dla powyższego przykładu:

$$\overset{u}{i-1} \leq 9.$$

Jest to prawdopodobnie lepszy kod, ponieważ wymaga zastosowania tylko jednego rozgałęzienia (tzw. pułapki porównania — ang. *compare trap*) oraz dlatego, że zapewne wartość $i - 1$ jest i tak potrzebna w związku z przeliczaniem adresów w tabeli. Należy jednak zastanowić się, czy następująca implementacja zawsze działa poprawnie i czy jest odporna na przepełnienia związane z operacją *odejmowania*:

$$a \le x \le b \Rightarrow \boldsymbol{x} - \overset{u}{\boldsymbol{a} \le} \boldsymbol{b} - \boldsymbol{a}\,.$$

Jest to prawda pod warunkiem, że $a \le b$. W przypadku kontroli zakresów w tabelach reguły języka mogą posiadać regułę zakazującą tworzenia tablic z zerową lub ujemną liczbą elementów w którymkolwiek z wymiarów. Reguła ta może być sprawdzana w czasie kompilacji lub w przypadku tablic dynamicznych w czasie alokacji tablicy w pamięci. W takich środowiskach powyższa transformacja jest poprawna, co spróbujemy wykazać.

Wygodnie będzie zastosować lemat, który sam w sobie jest wartościową informacją, którą warto pamiętać.

> *Lemat.* Jeśli a i b są liczbami całkowitymi ze znakiem i $a \le b$, wyliczona wartość $\boldsymbol{b} - \boldsymbol{a}$ prawidłowo reprezentuje wartość arytmetyczną $b - a$ w przypadku, gdy wyliczona wartość jest interpretowana jako wartość bez znaku.

Dowód. Załóżmy maszynę 32-bitową. Z faktu, że $a \le b$ wynika, iż $b - a$ należy do przedziału od 0 do $(2^{31} - 1) - (-2^{31}) = 2^{32} - 1$. W przypadku, gdy rzeczywista różnica należy do przedziału od 0 do $2^{31} - 1$, wynik maszynowy jest poprawny (ponieważ wynik jest reprezentowany w interpretacji ze znakiem) a bit znaku jest wyłączony. Stąd wynik maszynowy jest poprawny zarówno w interpretacji ze znakiem, jak i bez znaku.

W przypadku, gdy różnica należy do przedziału od 2^{31} do $2^{32} - 1$, wynik maszynowy będzie różnił się o wielokrotność liczby 2^{32} (ponieważ nie można go reprezentować za pomocą liczby ze znakiem). Spowoduje to, że wynik (reprezentowany ze znakiem) będzie należał do przedziału od -2^{31} do -1. Wynik maszynowy jest za niski o 2^{32} a bit znaku jest włączony. Interpretacja wyniku jako liczby bez znaku spowoduje zwiększenie wyniku o 2^{32}, ponieważ bit znaku otrzymuje wówczas wagę $+2^{31}$, nie zaś -2^{31}. Stąd reprezentowany wynik jest prawidłowy.

Twierdzenie ograniczeń

Twierdzenie W przypadku, gdy a i b są liczbami ze znakiem, takimi że $a \le b$, wtedy zachodzi następująca własność:

$$a \le x \le b \Rightarrow \boldsymbol{x} - \overset{u}{\boldsymbol{a} \le} \boldsymbol{b} - \boldsymbol{a} \qquad (1)$$

Dowód. Rozpatrzmy trzy przypadki w zależności od wartości $\boldsymbol{x}$. We wszystkich przypadkach, zgodnie z lematem, z faktu, że $a \le b$ wynika, że wartość wyliczona $\boldsymbol{b} - \boldsymbol{a}$ jest równa arytmetycznej wartości działania $b - a$ pod warunkiem, że wartość ta jest interpretowana bez znaku, zgodnie z formułą (1).

Przypadek 1., $x < a$. W tym przypadku $\boldsymbol{x} - \boldsymbol{a}$ interpretowane jako liczba bez znaku jest równoważne $x - a + 2^{32}$. Niezależnie od wartości x oraz b (z zakresu liczb 32-bitowych) zachodzi nierówność:

$$\boldsymbol{x} + 2^{32} > \boldsymbol{b}.$$

Stąd wynika:

$$\boldsymbol{x} - \boldsymbol{a} + 2^{32} > \boldsymbol{b} - \boldsymbol{a},$$

a stąd wynika:

$$\boldsymbol{x - a \overset{u}{>} b - a}\ .$$

W tym przypadku obydwie strony zależności (1) są **fałszywe**.

Przypadek 2., $a \le x \le b$. Arytmetycznie $x - a \le b - a$. Z faktu, że $a \le x$ wynika, iż zgodnie z lematem $x - a$ jest równe $\boldsymbol{x - a}$, w przypadku, gdy ten ostatni wynik jest interpretowane bez znaku, stąd:

$$\boldsymbol{x - a \overset{u}{\le} b - a}\ .$$

W tym przypadku obydwie strony zależności (1) są **prawdziwe**.

Przypadek 3., $x > b$. W tym przypadku $x - a > b - a$. Ponieważ w tym przypadku $x > a$ (gdyż $b \ge a$), z lematu wynika, iż $x - a$ jest równe co do wartości z $\boldsymbol{x - a}$, w przypadku, gdy ten ostatni wynik jest rozpatrywany bez znaku. Stąd:

$$\boldsymbol{x - a \overset{u}{>} b - a}\ .$$

W tym przypadku obydwie strony zależności (1) są **fałszywe**.

Powyższe twierdzenie jest prawdziwe również dla a i b będących liczbami całkowitymi bez znaku. Jest tak, ponieważ liczby całkowite bez znaku stanowią trywialny przypadek, w którym lemat jest spełniony i również przedstawiony dowód jest prawdziwy.

Poniżej przestawiamy podobne transformacje kontrolujące ograniczenia, ponownie została zacytowana transformacja z powyższego twierdzenia. Wszystkie przedstawione formuły są spełnione dla interpretacji zmiennych a, b oraz x ze znakiem lub bez znaku.

$$\text{jeśli } a \le b, \text{ to } a \le x \le b = \boldsymbol{x - a \overset{u}{\le} b - a = b - x \overset{u}{\le} b - a} \qquad (2)$$

$$\text{jeśli } a \le b, \text{ to } a \le x < b = \boldsymbol{x - a \overset{u}{<} b - a}$$

$$\text{jeśli } a \le b, \text{ to } a < x \le b = \boldsymbol{b - x \overset{u}{<} b - a}$$

$$\text{jeśli } a < b, \text{ to } a < x < b = \boldsymbol{x - a - 1 \overset{u}{\le} b - a - 1 = b - x - 1 \overset{u}{\le} b - a - 1}$$

W ostatniej regule $\boldsymbol{b - a - 1}$ można zastąpić $\boldsymbol{b + \neg a}$.

Istnieją również inne transformacje, które mogą być pomocne w przypadku, gdy kontrola zakresu ma postać $-2^{n-1} \le x \le 2^{n-1} - 1$. Jest to sprawdzenie możliwości interpretacji liczby x za pomocą reprezentacji uzupełnienia dwójkowego zbudowanej z liczby bitów równej n. Dla $n = 8$ można tego dokonać za pomocą jednego z poniższych testów:

a. $\boldsymbol{-128 \le x \le 127}$

b. $\boldsymbol{x + 128 \overset{u}{\le} 255}$

c. $\boldsymbol{(x \overset{s}{>>} 7) + 1 \overset{u}{\le} 1}$

d. $x \overset{s}{>>} 7 = x \overset{s}{>>} 31$

e. $(x \overset{s}{>>} 7) + (x \overset{u}{>>} 31) = 0$

f. $(x << 24) \overset{s}{>>} 24 = x$

g. $x \oplus (x \overset{s}{>>} 31) \leq 127$

Nierówność (b) jest po prostu zastosowaniem spostrzeżeń z niniejszego podrozdziału. Również równanie (c) zostało omówione wcześniej. Równania (c) – (f) i prawdopodobnie (g) są użyteczne głównie w przypadku, gdy stałe z równań (a) oraz (g) przekraczają dopuszczalny rozmiar pól wartości bezpośrednich dla instrukcji *porównania* i *dodawania*.

Kolejnym specjalnym przypadkiem obejmującym potęgi dwójki jest następujący:

$$0 \leq x \leq 2^n - 1 \Leftrightarrow (x \overset{u}{>>} n) = 0 .$$

Bardziej ogólna wersja powyższego wzoru wygląda następująco:

$$a \leq x \leq a + 2^n - 1 \Leftrightarrow ((x - a) \overset{u}{>>} n) = 0 .$$

4.2. Ograniczenia zakresów w operacjach sumy i różnicy

Niektóre kompilatory optymalizujące przeprowadzają „analizę zakresu" (ang. *range analysis*) wyrażeń. Jest to proces określania górnego i dolnego ograniczenia wartości każdego z wyrażeń programu. Optymalizacje tego typu z reguły nie dają spektakularnych wyników, pozwalają jednak na pominięcie kontroli zakresów w instrukcji `switch` języka C oraz na pominięcie niektórych kontroli zakresów wykorzystywanych przez kompilatory jako informacje wspomagające proces wykrywania błędów.

Załóżmy, że mamy do czynienia z ograniczeniami dwóch zmiennych x oraz y (rozpatrujemy przypadek liczb całkowitych bez znaku):

$$a \leq x \leq b$$
$$c \leq y \leq d \qquad (3)$$

W jaki sposób policzyć ograniczenia operacji $x + y$, $x - y$ oraz $-x$? Arytmetycznie oczywiście $a + c \leq x + y \leq b + d$, lecz problem polega na tym, że operacje *dodawania* mogą spowodować przepełnienie.

Sposób wyliczenia ograniczeń został określony w następującym twierdzeniu:

twierdzenie

Jeśli a, b, c, d, x oraz y są liczbami całkowitymi bez znaku, takimi że:

$$\overset{u}{a} \le \overset{u}{x} \le \overset{u}{b}$$
$$\overset{u}{c} \le \overset{u}{y} \le \overset{u}{d}$$

w takiej sytuacji zachodzą następujące własności:

Dla $a + c \le 2^{32} - 1$ oraz $b + d \ge 2^{32}$ zachodzi $\overset{u}{0} \le \overset{u}{x} + \overset{u}{y} \le \overset{u}{2^{32} - 1}$

w przeciwnym wypadku $\overset{u}{a} + \overset{u}{c} \le \overset{u}{x} + \overset{u}{y} \le \overset{u}{b} + \overset{u}{d}$ (4)

Dla $a - d < 0$ oraz $b - c \ge 0$ zachodzi $\overset{u}{0} \le \overset{u}{x} - \overset{u}{y} \le \overset{u}{2^{32} - 1}$

w przeciwnym wypadku $\overset{u}{a} - \overset{u}{d} \le \overset{u}{x} - \overset{u}{y} \le \overset{u}{b} - \overset{u}{c}$ (5)

Dla $a = 0$ oraz $b \ne 0$ zachodzi $\overset{u}{0} \le -\overset{u}{x} \le \overset{u}{2^{32} - 1}$

w przeciwnym wypadku $-\overset{u}{b} \le -\overset{u}{x} \le -\overset{u}{a}$ (6)

Nierówności (4) oznaczają, że ograniczeniami operacji $x + y$ jest „zwykłe" $a + c$ oraz $b + d$, lecz w przypadku, gdy operacja $a + c$ *nie* powoduje przepełnienia, natomiast $b + d$ powoduje przepełnienie, ograniczeniami są wartość 0 oraz maksymalna liczba całkowita bez znaku. Formuły (5) należy interpretować podobnie lecz w przypadku, gdy wynik odejmowania jest mniejszy od 0 powstanie przepełnienie (w kierunku ujemnym).

Dowód. W przypadku, gdy $a + c$ ani $b + d$ nie powodują przepełnienia, $x + y$ dla x oraz y spełniających założenia również nie powoduje przepełnienia, co powoduje, że wyliczone wartości są poprawne i jest spełniona druga nierówność formuły (4). Jeśli zarówno $a + c$, jak i $b + d$ powodują przepełnienia, wtedy przepełnienie spowoduje również operacja $x + y$. Z arytmetycznego punktu widzenia oczywiste jest, że zachodzi następująca własność:

$$a + c - 2^{32} \le x + y - 2^{32} \le b + d - 2^{32}.$$

Jest to jednak przypadek, w którym wszystkie trzy wspomniane operacje powodują przepełnienie. W tym przypadku zachodzi również następująca własność:

$$\overset{u}{a} + \overset{u}{c} \le \overset{u}{x} + \overset{u}{y} \le \overset{u}{b} + \overset{u}{d}.$$

W przypadku, gdy operacja $a + c$ nie powoduje przepełnienia lecz wywołuje je operacja $b + d$, zachodzą następujące własności:

$$a + c \le 2^{32} - 1$$
$$b + d \ge 2^{32}$$

Wyrażenie $x + y$ może przyjąć wartości z przedziału od $a + c$ do $b + d$, zatem może również przyjąć wartości $2^{32} - 1$ oraz 2^{32}, to znaczy wynik operacji $x + y$ może przyjąć wartość $2^{32} - 1$ oraz 0 (choć nie może przyjąć wszystkich wartości z tego przedziału).

Przypadek, gdy przepełnieniu ulega operacja $a + c$, natomiast $b + d$ nie powoduje przepełnienia nie może mieć miejsca, ponieważ $a \le b$ oraz $c \le d$.

W tym miejscu kończy się dowód formuły (4). Dowód formuły (5) jest podobny, lecz wszędzie w miejsce kontroli przepełnienia należy wstawić klauzulę, że rzeczywista różnica liczb x i y jest mniejsza od zera.

Nierówności (6) można udowodnić za pomocą (5) dla $a = b = 0$ a następnie dokonać odpowiedniego podstawienia. Wyrażenie $-x$ dla x będącego liczbą całkowitą bez znaku oznacza wartość $2^{32} - x$ lub $\neg x + 1$.

Przepełnienie bez znaku jest łatwe do wykrycia (patrz podrozdział *Dodawanie i odejmowanie bez znaku* na stronie 44), dlatego łatwo można wykorzystać w kodzie kontrolę zakresów. Na listingu 4.1 przedstawiamy kontrolę zakresów dolnego i górnego (odpowiednio, `s` oraz `t`) dla operacji *dodawania* i *odejmowania*.

Listing 4.1. *Przekazywanie ograniczeń zakresów bez znaku przez operacje dodawania i odejmowania*

```
s = a + c;
t = b + d;
if (s >= a && t < b) {
   s = 0;
   t = 0xFFFFFFFF;
}
s = a - d;
t = b - c;
if (s > a && t <= b) {
   s = 0;
   t = 0xFFFFFFFF;
}
```

Liczby ze znakiem

Przypadek liczb ze znakiem nie jest już tak prosty. Podobnie jak poprzednio, załóżmy, że mamy do czynienia z dwoma zmiennymi x oraz y spełniającymi następujące zależności (wszystkie wartości są liczbami całkowitymi ze znakiem)

$$\begin{aligned} a \le x \le b \\ c \le y \le d \end{aligned} \tag{7}$$

Chcemy wyliczyć dokładne ograniczenia operacji $x + y$, $x - y$ oraz $-x$. Rozumowanie jest bardzo podobne do rozumowania przeprowadzonego dla liczb bez znaku. Poniżej przedstawiamy odpowiednie formuły dla operacji *dodawania*:

$$\begin{aligned} a + c < -2^{31},\ b + d < -2^{31} &: \boldsymbol{a + c \le x + y \le b + d} \\ a + c \le -2^{31},\ b + d \ge -2^{31} &: \boldsymbol{-2^{31} \le x + y \le 2^{31} - 1} \\ -2^{31} \le a + c < 2^{31},\ b + d < 2^{31} &: \boldsymbol{a + c \le x + y \le b + d} \\ -2^{31} \le a + c < 2^{31},\ b + d \ge 2^{31} &: \boldsymbol{-2^{31} \le x + y \le 2^{31} - 1} \\ a + c \ge 2^{31},\ b + d \ge 2^{31} &: \boldsymbol{a + c \le x + y \le b + d} \end{aligned} \tag{8}$$

Pierwszy wiersz zależności dotyczy przypadku, gdy zarówno operacja $a + b$, jak i operacja $b + d$ ulegają przepełnieniu w kierunku ujemnym. W tym przypadku suma $x + y$ leży pomiędzy wyliczonymi sumami $a + c$ oraz $b + d$. Dzieje się tak, ponieważ trzy wspomniane sumy mają zbyt duże wartości o tę samą różnicę (2^{32}). Drugi wiersz oznacza, że w przypadku, gdy suma $a + c$ ulega przepełnieniu w kierunku ujemnym a suma $b + d$ nie ulega przepełnieniu lub ulega przepełnieniu w kierunku dodatnim, wtedy wyliczona suma $x + y$ może przyjąć zarówno wartość najmniejszej liczby ujemnej, jak i największej liczby dodatniej (lecz zapewne nie może przyjąć wszystkich wartości leżących pomiędzy nimi), co nie jest trudne do wykazania. Pozostałe wiersze można zinterpretować w podobny sposób.

Zasady przeniesienia ograniczeń w przypadku liczb całkowitych ze znakiem za pomocą operacji *różnicy* można z łatwością wyprowadzić zastępując definicję ograniczeń liczby y następującą:

$$-d \le -y \le -c$$

Ograniczenia te podstawiamy do reguł dotyczących operacji *sumy*. Otrzymamy poniższy zestaw reguł:

$$a - d < -2^{31},\ b - c < -2^{31} : \boldsymbol{a - d \le x - y \le b - c}$$

$$a - d \le -2^{31},\ b - c \ge -2^{31} : \boldsymbol{-2^{31} \le x - y \le 2^{31} - 1}$$

$$-2^{31} \le a - d < 2^{31},\ b - c < 2^{31} : \boldsymbol{a - d \le x - y \le b - c}$$

$$-2^{31} \le a - d < 2^{31},\ b - c \ge 2^{31} : \boldsymbol{-2^{31} \le x - y \le 2^{31} - 1}$$

$$a - d \ge 2^{31},\ b - c \ge 2^{31} : \boldsymbol{a - d \le x - y \le b - c}$$

Reguły dla negacji można wydobyć z reguł dla odejmowania podstawiając odpowiednio $a = b = 0$, pomijając niektóre z kombinacji i upraszczając. Wynik jest następujący:

$$a = -2^{31},\ b = -2^{31} : \boldsymbol{-x = -2^{31}}$$

$$a = -2^{31},\ b \ne -2^{31} : \boldsymbol{-2^{31} \le -x = 2^{31} - 1}$$

$$a \ne -2^{31} : \boldsymbol{-b \le -x \le -a}$$

Kod w języku C dla liczb ze znakiem byłby dość skomplikowany. Weźmy pod uwagę tylko dodawanie. Wydaje się, że najprościej będzie rozpatrzyć dwa przypadki z (8), w których wynik $x + y$ będzie najmniejszą oraz największą dopuszczalną liczbą. Przepełnienie w kierunku ujemnym występuje w przypadku, gdy obydwa operandy mają znak ujemny a ich suma nie jest ujemna (zob. podrozdział *Dodawanie i odejmowanie ze znakiem*, rozdział 2., strona 41). Dlatego w celu sprawdzenia, czy $a + b < -2^{31}$ przyjmiemy `s = a + c`, następnie posłużymy się następującym sprawdzeniem: `if (a < 0 && c < 0 && s >= 0)`.... Wydajniejszym sposobem będzie jednak wykonywanie operacji logicznych bezpośrednio na zmiennych arytmetycznych, co spowoduje, że bit znaku

będzie przechowywał wynik logiczny operacji[1]. W ten sposób powyższy warunek zapiszemy następująco: `if ((a & c & ~s) < 0)`.... Te przemyślenia zostały wykorzystane w kodzie przedstawionym na listingu 4.2.

Listing 4.2. *Przenoszenie ograniczeń operacji ze znakiem przez operacje sumy*

```
s = a + c;
t = b + d;
u = a & c & ~s & ~(b & d & ~t);
v = ((a ^ c) | ~(a ^ s)) & (~b & ~d & t);
if ((u | v) < 0) {
   s = 0x80000000;
   t = 0x7FFFFFFF;}
```

W powyższym kodzie `u` ma wartość **prawda** (bit znaku ma wartość 1), jeśli suma `a + c` ulega przepełnieniu w kierunku ujemnym a suma `b + d` *nie ulega* przepełnieniu w kierunku ujemnym. Zmienna `v` ma wartość **prawda**, jeśli `a + c` nie ulega przepełnieniu a `b + d` ulega przepełnieniu w kierunku dodatnim. Pierwszy warunek można opisać słowami: „`a` oraz `c` mają przeciwne znaki lub `a` oraz `s` mają ten sam znak". Zastosowany warunek w instrukcji `if` jest równoważny następującemu: `if (u < 0 || v < 0)`. Oznacza to, że `u` lub `v` muszą mieć wartość **prawda**, aby warunek został spełniony.

4.3. Ograniczenia zakresów w operacjach logicznych

Załóżmy, podobnie jak w ostatnim podrozdziale, że mamy określone następujące ograniczenia zakresów zmiennych x i y, będących liczbami całkowitymi bez znaku:

$$a \le x \le b$$
$$c \le y \le d \tag{9}$$

W jaki sposób możemy określić ograniczenia wyników instrukcji $x \mid y$, $x \,\&\, y$, $x \oplus y$ oraz $\neg x$? Po połączeniu nierówności (9) z wybranymi zależnościami z podrozdziału 2.3 oraz po zastosowaniu spostrzeżenia, że $\neg x = 2^{32} - 1 - x$, otrzymamy następujące zależności:

$$\begin{aligned}
\max(a,c) &\le (x \mid y) \le b + d \\
0 &\le (x \,\&\, y) \le \min(b, d) \\
0 &\le (x \oplus y) \le b + d \\
\neg b &\le \neg x \le a
\end{aligned}$$

[1] Możemy uzyskać kod bardziej wydajny, wykorzystujący mniejszą liczbę rozgałęzień, jeśli na początku sprawdzimy, czy w ogóle występują przepełnienia, pod warunkiem, że ograniczenia nie mają dużych wartości.

Zakładamy, że operacja $b + d$ nie powoduje przepełnienia. Wymienione zależności są dość łatwe do wyliczenia i mogą być wystarczająco dobrym rozwiązaniem dla wspomnianego wyżej kompilatora optymalizującego. Niestety ograniczenia w pierwszych dwóch nierównościach nie są ścisłe. Załóżmy na przykład, że mamy do czynienia z następującymi stałymi (zapisanymi dwójkowo):

$$\mathbf{00010} \le x \le \mathbf{00100}$$

$$\mathbf{01001} \le y \le \mathbf{10100} \qquad (10)$$

Metodą sprawdzenia wszystkich możliwych wartości x i y dochodzimy do wniosku, iż $\mathbf{01010} \le (x \mid y) \le \mathbf{10111}$. Ograniczeniem dolnym nie jest zatem $\max(a,c)$ ani $a \mid c$ a górnym ograniczeniem nie jest $b + d$ ani $b \mid d$.

W jaki sposób można uzyskać ścisłe ograniczenia wyrażeń logicznych? Rozpatrzmy minimalne wartości operacji $x \mid y$. Rozsądnym założeniem byłoby przyjęcie dolnego ograniczenia wyniku jako $a \mid c$, czyli alternatywa wartości minimalnych tych zmiennych. Przykład (10) pokazuje jednak, że minimum wartości takiej operacji może być jednak mniejsze.

Aby znaleźć minimum, rozpoczniemy od $x = a$ i $y = c$ a następnie wyznaczymy wartość, o którą należy zwiększyć x lub y, aby zmniejszyć wartość operacji $x \mid y$. Wynikiem naszych poszukiwań będzie właśnie ta wielkość. Zamiast przypisywać wartości a i b zmiennym x i y będziemy działać bezpośrednio na tych wartościach zwiększając wartość jednej z nich w przypadku, gdy spełni to nasze założenia, to znaczy zmniejszy wartość operacji $a \mid c$.

Procedura polega na przeglądaniu bitów a i c od lewej do prawej. Jeśli bity na odpowiadających sobie pozycjach w a oraz c mają wartość 0, wynik będzie zawierał 0 na tej pozycji. Jeśli obydwa bity mają wartość 1, wynik również będzie zawierał jedynkę na tej pozycji. W przypadku, gdy jeden z bitów ma wartość 1 a odpowiadający mu bit w drugim ograniczeniu ma wartość 0, wtedy możliwe jest, że zmiana 0 na 1 i ustawienie wszystkich następujących bitów na wartość 0 zmniejszy wartość $a \mid c$. Wiemy jednak na pewno, że taka zmiana nie zwiększy wartości $a \mid c$, ponieważ wynik i tak na tej pozycji będzie zawierać bit o wartości 1 (pochodzący z drugiej zmiennej). Tę ze zmiennych, która zawiera na danej pozycji wartość 0 modyfikujemy w ten sposób, że wartość 0 zmieniamy na 1 a kolejne bity zmieniamy na 0. Możemy tak zrobić wyłącznie wtedy, gdy uzyskana wartość jest mniejsza lub równa górnemu ograniczeniu danej zmiennej. Wynikiem końcowym (ograniczeniem dolnym operacji $x \mid y$) będzie wynik alternatywy bitowej tak skonstruowanej wartości z drugim (niezmienionym) ograniczeniem. Jeśli nie możemy dokonać opisanej zmiany (ponieważ zmieniona wartość przekroczyłaby górne ograniczenie danej zmiennej), kontynuujemy analizę na kolejnej pozycji.

I to już wszystko. Można by podejrzewać, że po dokonaniu zmiany należy kontynuować przeglądanie, poszukując kolejnej możliwości zredukowania wartości $a \mid c$. Nawet jeśli jednak uda się znaleźć pozycję, na której 0 można zamienić na 1, zmiana następujących po niej bitów na 0 nie wpłynie na dalsze zmniejszenie wartości $a \mid c$, ponieważ bity te już mają wartość zero (z pierwszego zastąpienia).

Kod tego algorytmu w języku C przedstawiamy na listingu 4.3. Zakładamy, że kompilator przeniesie poza pętlę wyliczenie podwyrażeń `~a & c` oraz `a & ~c`. Co więcej, w przypadku, gdy na danej platformie jest dostępna operacja wyliczająca *liczbę zer wiodących*, program można przyśpieszyć inicjując wartość `m` w następujący sposób:

```
m = 0x80000000 >> nlz(a ^ c);
```

Dzięki temu możemy pominąć analizę pozycji, na których zarówno `a`, jak i `c` mają identyczną wartość 0 lub 1. Aby to przyśpieszenie rzeczywiście miało miejsce w przypadku, gdy `a ^ c = 0` (to znaczy, gdy `a = c`) operacja *przesunięcia w prawo* musi działać modulo 64. Jeśli instrukcja odczytująca *liczbę zer wiodących* nie jest dostępna, być może warto zastosować na wyniku operacji `a ^ c` którąś z przedstawionych wersji funkcji flp2 (zob. podrozdział 3.2, strona 64).

Listing 4.3. *Minimalna wartość x | y z ograniczeniami na x i y*

```
unsigned minOR(unsigned a, unsigned b,
               unsigned c, unsigned d) {
   unsigned m, temp;

   m = 0x80000000;
   while (m != 0) {
      if (~a & c & m) {
         temp = (a | m) & -m;
         if (temp <= b) {a = temp; break;}
      }
      else if (a & ~c & m) {
         temp = (c | m) & -m;
         if (temp <= d) {c = temp; break;}
      }
      m = m >> 1;
   }
   return a | c;
}
```

Teraz rozpatrzmy sposób określenia maksymalnej wartości operacji $x \mid y$ za pomocą zmiennych oszacowanych nierównościami (9). Algorytm, który posłuży nam w tym celu jest podobny do tego, który wykorzystaliśmy do oszacowania ograniczenia dolnego. Różnica polega na tym, że tym razem rozpatrujemy wartości b i d i przeglądając je (od strony lewej do prawej) poszukujemy pozycji, na których obydwie zmienne mają bit o wartości 1. Po znalezieniu takiej pozycji staramy się zwiększyć wartość wyrażenia $\boldsymbol{b} \mid \boldsymbol{d}$ przez zastąpienie wartości bitu w jednej ze zmiennych z 1 na 0, natomiast wszystkie kolejne bity zamieniając na wartość 1. Jeśli zmiana taka jest dopuszczalna (wartość zmiennej jest większa lub równa od dolnego ograniczenia tej zmiennej), zostaje ona wykonana a wynikiem będzie wartość alternatywy uzyskanej wartości z oryginalną wartością drugiej ze zmiennych. W przypadku, gdy nie można dokonać zmiany, próbujemy postąpić analogicznie z drugą ze zmiennych. jeśli nie można dokonać zmiany na drugiej ze zmiennych, kontynuujemy poszukiwania na kolejnych pozycjach. Kod tego algorytmu w języku C został przedstawiony na listingu 4.4.

Listing 4.4. *Maksymalna wartość x | y z ograniczeniami na x i y*

```
unsigned maxOR(unsigned a, unsigned b,
               unsigned c, unsigned d) {
   unsigned m, temp;

   m = 0x80000000;
   while (m != 0) {
      if (b & d & m) {
         temp = (b - m) | (m - 1);
         if (temp >= a) {b = temp; break;}
         temp = (d - m) | (m - 1);
         if (temp >= c) {d = temp; break;}
      }
      m = m >> 1;
   }
   return b | d;
}
```

W tym przypadku na zewnątrz pętli można wyprowadzić wyrażenie `b & d`. Ponownie algorytm można przyśpieszyć za pomocą instrukcji „nlz":

```
m = 0x80000000 >> nlz(b & d);
```

Istnieją dwa sposoby na które możemy określić ograniczenia w nierównościach (9) dla operacji $x \& y$: algebraicznie i za pomocą bezpośrednich obliczeń. Metoda algebraiczna wykorzystuje prawo DeMorgana:

$$x \& y = \neg(\neg x \mid \neg y)$$

Sposób określania ograniczeń w wyrażeniach *alternatywy* jest dobrze znany, natomiast oszacowanie ich dla instrukcji *dopełnienia bitowego* jest trywialne ($a \overset{u}{\le} x \overset{u}{\le} b \Leftrightarrow \neg b \overset{u}{\le} \neg x \overset{u}{\le} \neg a$), zatem otrzymamy następujące zależności:

$$\text{minAND}(a, b, c, d) = \neg\text{maxOR}(\neg b, \neg a, \neg d, \neg c)$$

$$\text{maxAND}(a, b, c, d) = \neg\text{minOR}(\neg b, \neg a, \neg d, \neg c)$$

Metoda bezpośrednich obliczeń jest podobna do wykorzystywanej w przypadku wyznaczania ograniczeń operacji *alternatywy*. Dla górnego i dolnego ograniczenia przedstawiamy ją, odpowiednio, na listingach 4.5 oraz 4.6.

Listing 4.5. *Minimalna wartość x & y, z ograniczeniami na x i y*

```
unsigned minAND(unsigned a, unsigned b,
                unsigned c, unsigned d) {
   unsigned m, temp;

   m = 0x80000000;
   while (m != 0) {
      if (~a & ~c & m) {
         temp = (a | m) & -m;
         if (temp <= b) {a = temp; break;}
```

```
            temp = (c | m) & -m;
            if (temp <= d) {c = temp; break;}
         }
         m = m >> 1;
      }
      return a & c;
   }
```

Listing 4.6. *Maksymalna wartość x & y, z ograniczeniami na x i y*

```
   unsigned maxAND(unsigned a, unsigned b,
                   unsigned c, unsigned d) {
      unsigned m, temp;

      m = 0x80000000;
      while (m != 0) {
         if (b & ~d & m) {
            temp = (b & ~m) | (m - 1);
            if (temp >= a) {b = temp; break;}
         }
         else if (~b & d & m) {
            temp = (d & ~m) | (m - 1);
            if (temp >= c) {d = temp; break;}
         }
         m = m >> 1;
      }
      return b & d;
   }
```

Algebraiczna metoda wyznaczania ograniczeń wyrażeń *koniunkcji*, *alternatywy* oraz *dopełnienia bitowego* działa dla wszystkich bitowych wyrażeń logicznych, za wyjątkiem *różnicy symetrycznej* oraz *równoważności*. Powodem problemów z oszacowaniem tych wyrażeń jest to, że w przypadku rozpisania ich za pomocą wyrażeń *koniunkcji*, *alternatywy* i *dopełnienia* uzyskujemy dwa podwyrażenia zawierające x oraz y. Na przykład można rozważyć następującą zależność:

$$\min_{\substack{a \le x \le b \\ c \le y \le d}} (x \oplus y) = \min_{\substack{a \le x \le b \\ c \le y \le d}} ((x \,\&\, \neg y) \mid (\neg x \,\&\, y))$$

Dwa podwyrażenia zawierające *koniunkcję* nie mogą być osobno zminimalizowane (bez dowodu, że jest tak naprawdę, choć w rzeczywistości jest to prawdziwe twierdzenie), ponieważ poszukujemy jednej wartości x oraz jednej wartości y, minimalizujących wartość całego wyrażenie.

W celu określenia ograniczeń wyrażenia *różnicy symetrycznej* można zastosować następujące własności:

$$\mathrm{minXOR}(\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}, \boldsymbol{d}) = \mathrm{minAND}(\boldsymbol{a}, \boldsymbol{b}, \neg\boldsymbol{d}, \neg\boldsymbol{c}) \mid \mathrm{minAND}(\neg\boldsymbol{b}, \neg\boldsymbol{a}, \boldsymbol{c}, \boldsymbol{d})$$

$$\mathrm{maxXOR}(\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}, \boldsymbol{d}) = \mathrm{maxOR}(\mathbf{0}, \mathrm{maxAND}(\boldsymbol{a}, \boldsymbol{b}, \neg\boldsymbol{d}, \neg\boldsymbol{c}), \mathbf{0}, \mathrm{maxAND}(\neg\boldsymbol{b}, \neg\boldsymbol{a}, \boldsymbol{c}, \boldsymbol{d}))$$

Wyliczenie wartości minXOR oraz maxXOR jest już proste. Kod wyliczający minXOR jest taki sam jak przedstawiony na listingu 4.3 kod funkcji minOR, za wyjątkiem tego, że instrukcje `break` zostają usunięte a wartość zwracaną z funkcji zmieniamy na `a ^ c`. Kod funkcji maxXOR jest taki sam jak kod funkcji maxOR przedstawiony na listingu 4.4, za wyjątkiem czterech wierszy w instrukcji `if`, które zastępujemy następującymi:

```
temp = (b - m) | (m - 1);
if (temp >= a) b = temp;
else {
   temp = (d - m) | (m - 1);
   if (temp >= c) d = temp;
}
```

Zwracaną wartość natomiast zmieniamy na `b ^ d`.

Ograniczenia ze znakiem

W przypadku, gdy ograniczenia są liczbami całkowitymi ze znakiem wyliczanie ich dla wyrażeń logicznych jest znacznie bardziej skomplikowanym zadaniem. W przypadku, gdy ograniczenia a i b lub c i d są przeciwnych znaków, obliczenia są nieregularne. Jeden ze sposobów wyliczenia dolnego i górnego ograniczenia operacji *alternatywy* $x \mid y$ został przedstawiony w tabeli 4.1. Znak + oznacza, że ograniczenie w danej kolumnie ma znak dodatni, znak - oznacza, że dane ograniczenie ma znak ujemny. Kolumna opisana „minOR (ze znakiem)" zawiera wyrażenia wyliczające dolne ograniczenie $x \mid y$. Ostatnia kolumna zawiera wyrażenia wyliczające górne ograniczenie $x \mid y$. Jednym ze sposobów implementacji jest zbudowanie z bitów znaku ograniczeń a, b, c i d wartości od 0 do 15 i zastosowanie instrukcji `switch` na tych wartościach. Nie są wykorzystane wszystkie wartości od 0 do 15, ponieważ niemożliwe są sytuacje $a > b$ czy też $c > d$.

Tabela 4.1. *Obliczanie funkcji minOR oraz maxOR ze znakiem za pomocą operacji bez znaku*

a	*b*	*c*	*d*	minOR (bez znaku)	maxOR (ze znakiem)
–	–	–	–	minOR(a, b, c, d)	maxOR(a, b, c, d)
–	–	–	+	a	–1
–	–	+	+	minOR(a, b, c, d)	maxOR(a, b, c, d)
–	+	–	–	c	–1
–	+	–	+	min(a, c)	maxOR(0, b, 0, d)
–	+	+	+	minOR(a, 0xFFFFFFFF, c, d)	maxOR(0, b, c, d)
+	+	–	–	minOR(a, b, c, d)	maxOR(a, b, c, d)
+	+	–	+	minOR(a, b, c, 0xFFFFFFFF)	maxOR(a, b, 0, d)
+	+	+	+	minOR(a, b, c, d)	maxOR(a, b, c, d)

Dla liczb ze znakiem prawdziwa jest następująca relacja:

$$a \le x \le b \Leftrightarrow \neg b \le \neg x \le \neg a$$

W celu rozwinięcia wyrażeń z tabeli 4.1 do obsługi innych wyrażeń logicznych (za wyjątkiem *różnicy symetrycznej* oraz *równoważności*) można zatem zastosować metodę algebraiczną. Zagadnienie to zostawiamy jako ćwiczenie dla Czytelnika.

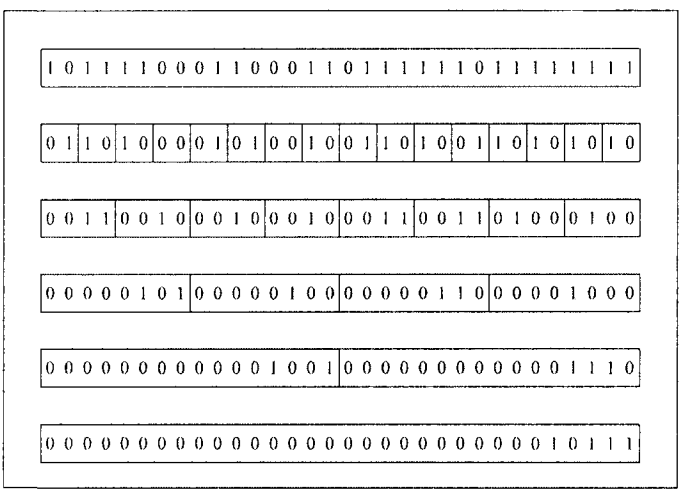
Rozdział 5.

Zliczanie bitów

5.1. Zliczanie jedynek

Komputer IBM Stretch (ok. 1960 r.) posiadał mechanizm zliczający liczbę jedynek w słowie, jak również liczbę wiodących zer. Obydwie te wartości były dostępne jako produkt uboczny wszystkich operacji logicznych! Funkcja zliczająca liczbę jedynek zwana jest również *wielkością populacji* (ang. *population count*) i była dostępna na przykład na komputerach Stretch czy też SPARCv9.

W komputerach nieobsługujących tej funkcji dobrym sposobem zliczania liczby jedynek w słowie jest ustawienie każdego z 2-bitowych pól na sumę jedynek w każdym z tych pól, następnie zliczenie liczby jedynek w każdym z 4-bitowych pól i zapisanie jej w tym polu itd. Więcej informacji na temat tej sztuczki można znaleźć w pracy [RND]. Metoda ta została zilustrowana na rysunku 5.1. Pierwszy wiersz ilustruje słowo, w którym zliczamy jedynki. Ostatni wiersz zawiera wynik (liczba 23).



Rysunek 5.1. *Zliczanie jedynek w słowie za pomocą strategii „dziel i zwyciężaj"*

Jest to przykład strategii *dziel i zwyciężaj* (ang. *divide and conquer*), w której oryginalny problem (zsumowania 32 bitów) zostaje podzielony na dwa problemy (zsumowanie dwóch pól 16-bitowych), z których każdy jest rozwiązywany osobno a następnie wyniki są łączone (w tym przypadku zsumowane). Strategia ta jest stosowana rekurencyjnie, następnie 16 bitów zostaje podzielonych na pola 8-bitowe itd.

W naszym przypadku najmniejsze problemy (sumowanie sąsiednich bitów) mogą być wykonywane równolegle i łączenie sąsiednich sum również może być dokonywane równolegle w stałej liczbie kroków. W wyniku uzyskujemy algorytm wykonujący się w $\log_2(32) = 5$ krokach.

Innym przykładem wykorzystania strategii *dziel i zwyciężaj* jest dobrze znana technika przeszukiwania binarnego, metoda sortowania szybkiego (*quicksort*) oraz metoda odwracania bitów w słowie, którą opisujemy w podrozdziale 7.1.

Omówiona metoda (której ilustrację przedstawiamy na rysunku 5.1.) może zostać zapisana w postaci następującego kodu w języku C:

```
x = (x & 0x55555555) + ((x >> 1) & 0x55555555);
x = (x & 0x33333333) + ((x >> 2) & 0x33333333);
x = (x & 0x0F0F0F0F) + ((x >> 4) & 0x0F0F0F0F);
x = (x & 0x00FF00FF) + ((x >> 8) & 0x00FF00FF);
x = (x & 0x0000FFFF) + ((x >>16) & 0x0000FFFF);
```

Pierwszy wiersz kodu wykorzystuje operację `(x >> 1) & 0x55555555` zamiast bardziej naturalnego zapisu `(x & 0xAAAAAAAA) >> 1`, ponieważ proponowany kod zapobiega zapisywaniu w rejestrze dwóch dużych stałych. To spowodowałoby stratę jednej instrukcji w przypadku, gdyby maszyna nie obsługiwała instrukcji *and not*. Podobna uwaga dotyczy pozostałych wierszy kodu.

Oczywiście ostatnia operacja *koniunkcji* jest zbyteczna, pozostałe operacje *koniunkcji* mogą również zostać pominięte pod warunkiem, że nie ma obawy o wystąpienie przeniesienia sumy do sąsiedniego sumowanego pola. Co więcej, istnieje sposób na zakodowanie pierwszego wiersza z zastosowaniem o jeden mniejszej liczby instrukcji. To prowadzi nas do uproszczenia kodu przedstawionego na listingu 5.1, który wykonuje się w 21 instrukcjach i nie wykorzystuje rozgałęzień.

Listing 5.1. *Zliczanie jedynek w słowie*

```
int pop(unsigned x) {
   x = x - ((x >> 1) & 0x55555555);
   x = (x & 0x33333333) + ((x >> 2) & 0x33333333);
   x = (x + (x >> 4)) & 0x0F0F0F0F;
   x = x + (x >> 8);
   x = x + (x >> 16);
   return x & 0x0000003F;
}
```

Pierwsze przypisanie zmiennej x zostało oparte na pierwszych dwóch składnikach stosunkowo zaskakującej formuły:

$$\mathrm{pop}(x) = x - \left\lfloor \frac{x}{2} \right\rfloor - \left\lfloor \frac{x}{4} \right\rfloor - \ldots - \left\lfloor \frac{x}{2^{31}} \right\rfloor \tag{1}$$

W równaniu (1) x musi być większe lub równe 0. Traktując x jako liczbę bez znaku, równanie (1) można zrealizować za pomocą 31 *przesunięć w prawo* o wartość bezpośrednią 1 oraz 31 *odejmowań*. Funkcja przedstawiona na listingu 5.1 wykorzystuje równolegle dwa pierwsze elementy tego równania na każdym 2-bitowym polu.

Istnieje prosty dowód równania (1), który przedstawiamy poniżej dla przypadku 4-bitowego słowa. Załóżmy, że słowo jest postaci $b_3b_2b_1b_0$, dla każdego $b_i = 0$ lub 1. Mamy zatem:

$$\begin{aligned}
x - \left\lfloor \frac{x}{2} \right\rfloor - \left\lfloor \frac{x}{4} \right\rfloor - \left\lfloor \frac{x}{8} \right\rfloor &= b_3 \cdot 2^3 + b_2 \cdot 2^2 + b_1 \cdot 2^1 + b_0 \cdot 2^0 \\
&\quad - (b_3 \cdot 2^2 + b_2 \cdot 2^1 + b_1 \cdot 2^0) \\
&\quad - (b_3 \cdot 2^1 + b_2 \cdot 2^0) \\
&\quad - (b_3 \cdot 2^0) \\
&= b_3(2^3 - 2^2 - 2^1 - 2^0) + b_2(2^2 - 2^1 - 2^0) + b_1(2^1 - 2^0) + b_0(2^0) \\
&= b_3 + b_2 + b_1 + b_0
\end{aligned}$$

Ponadto można zauważyć, że równanie (1) można wyprowadzić dzięki spostrzeżeniu, że i-ty bit binarnej reprezentacji nieujemnej liczby całkowitej x można zapisać następująco:

$$b_i = \left\lfloor \frac{x}{2^i} \right\rfloor - 2\left\lfloor \frac{x}{2^{i+1}} \right\rfloor$$

Powyższy wzór dotyczy wartości i od 0 do 31. Ostatni element jest równy zeru, ponieważ $x < 2^{32}$. Równanie (1) można również zastosować do liczb o innych podstawach. Na przykład dla podstawy dziesiętnej mamy:

$$\mathrm{sum_digits}(x) = x - 9\left\lfloor \frac{x}{10} \right\rfloor - 9\left\lfloor \frac{x}{100} \right\rfloor - \ldots$$

Poszczególne elementy powyższej formuły są rozwijane do momentu, gdy osiągną wartość 0. Dowód powyższej formuły można wyprowadzić za pomocą tej samej techniki, która została zastosowana poprzednio.

Odmiana powyższego algorytmu wykorzystuje analogię równania (1) dla podstawy czwórkowej. W zastępstwie drugiego wiersza funkcji z listingu 5.1 wstawiamy następujący kod:

```
x = x - 3 * ((x >> 2) & 0x33333333)
```

Kod ten wymaga jednak tyle samo instrukcji co poprzednio a ponadto wymaga szybkiej instrukcji *mnożenia przez 3*.

Algorytm przedstawiony w periodyku HAKMEM [HAK, pozycja 169] zlicza jedynki w słowie wykorzystując pierwsze trzy składniki formuły (1) tworząc słowo podzielone na 3-bitowe pola, z których każde zawiera tę wartość liczby bitów, która znajdowałaby się w odpowiadających im polach słowa źródłowego. Następnie algorytm łączy sąsiednie 3-bitowe pola w pola o rozmiarze 6 bitów, sumując ich zawartości, potem te 6-bitowe pola są sumowane, zaś wynik zwracany za pomocą dzielenia słowa modulo 63. W języku C algorytm ten można zapisać następująco:

```
int pop(unsigned x) {
   unsigned n;

   n = (x >> 1) & 033333333333;        // zliczamy jedynki
   x = x - n;                          // w każdym 3-bitowym
   n = (n >> 1) & 033333333333;        // polu
   x = x - n;
   x = (x + (x >> 3)) & 030707070707;  // sumujemy pola 3-bitowe
   return x%63;                        // sumujemy pola 6-bitowe
}
```

Spostrzeżenie, że dzielenie modulo powoduje zsumowanie 6-bitowych pól stanie się bardziej oczywiste, gdy przyjmiemy słowo x jako liczbę zapisaną w systemie o podstawie 64. Reszta z dzielenia liczby całkowitej o podstawie b przez liczbę $b - 1$ jest, dla $b \geq 3$, przystająca modulo b do sumy cyfr w tej liczbie i jest oczywiście mniejsza od b. Ponieważ w tym przypadku suma cyfr musi być mniejsza lub równa 32, x % 63 musi być równe sumie cyfr w x, co jest z kolei równe liczbie jedynek w oryginalnej liczbie x.

Powyższy algorytm wymaga zaledwie 10 instrukcji na maszynie DEC PDP-10, ponieważ posiada ona instrukcję *wyliczającą resztę z dzielenia* z drugim operandem wskazującym słowo w pamięci. Na maszynie obsługującej podstawowy zestaw instrukcji RISC algorytm ten potrzebuje 13 instrukcji, z założeniem, że procesor obsługuje instrukcję *reszty z dzielenia bez znaku* jako pojedynczą instrukcję (lecz nie odwołującej się w jednym z operandów do wartości bezpośredniej lub wartości w pamięci). Nie jest to zawrotna wydajność, ponieważ dzielenie jest z reguły powolną operacją. Nie da się jej również zastosować do słów o długości 64 bitów za pomocą prostej modyfikacji wartości stałych, lecz można tym sposobem zmodyfikować algorytm do obsługi słów o długości maksymalnie 62 bitów.

Inną odmianą algorytmu przedstawionego w HAKMEM jest wykorzystujący równanie (1) w celu zsumowania jedynek w każdym z 4-bitowych pól, pracując na wszystkich ośmiu polach jednocześnie [Hay1]. Następnie 4-bitowe sumy łatwo można zsumować do pól ośmiobitowych. Powstałe w ten sposób cztery bajty można łatwo zsumować mnożąc słowo przez stałą 0x01010101.

```
int pop(unsigned x) {
   unsigned n;

   n = (x >> 1) & 0x77777777;          // Zliczamy jedynki
   x = x - n;                          // w każdym 4-bitowym
   n = (n >> 1) & 0x77777777;          // polu.
   x = x - n;
   n = (n >> 1) & 0x77777777;
```

```
    x = x - n;
    x = (x + (x >> 4)) & 0x0F0F0F0F;  // Wyliczamy podsumy o rozmiarze bajtu.
    x = x*0x01010101;                 // Sumujemy bajty.
    return x >> 24;
}
```

Powyższy kod wymaga 19 instrukcji z podstawowego zestawu RISC. Algorytm ten działa dobrze w przypadku wykorzystania dwuadresowej maszyny, ponieważ pierwsze sześć wierszy kodu można wykonać z wykorzystaniem tylko jednej instrukcji *przeniesienia wartości rejestru* (ang. *move register*). Ponadto wielokrotne zastosowanie maski `0x77777777` pozwala na załadowanie tej wartości do jednego z rejestrów i wykorzystanie w instrukcjach operujących na rejestrach. Co równie istotne, większość przesunięć odbywa się zaledwie o jedną pozycję.

Zupełnie inna technika zliczania jedynek w słowie została przedstawiona na listingu 5.2. Polega ona na kolejnym wyłączaniu prawostronnych jedynek aż do momentu uzyskania wyniku zerowego [Weg, RND]. Jest to bardzo szybki algorytm, pod warunkiem, że w słowie występuje niewielka liczba jedynek. Wymaga 2 + 5pop(x) instrukcji.

Listing 5.2. *Zliczanie jedynek w słowie o rzadkiej populacji*

```
int pop(unsigned x) {
   int n;

   n = 0;
   while (x != 0) {
      n = n + 1;
      x = x & (x - 1);
   }
   return n;
}
```

Algorytm ten posiada wersję pokrewną, zakładającą występowanie dużej liczby jedynek w słowie. Polega on na odwracaniu prawostronnych zer za pomocą wyrażenia `x = x | (x + 1)` do momentu, gdy wynik składa się z samych jedynek (–1). Następnie jest wykonywana operacja 32 – n. Alternatywnie można zainicjować n do wartości 32 i odliczać w dół.

Raczej zaskakujący algorytm zliczania jedynek polega na przesunięciu cyklicznym słowa w lewo 31 razy dodając poszczególne wyniki. Wynik ostateczny będzie równy –pop(x)! Mamy zatem następującą formułę:

$$\mathrm{pop}(x) = -\sum_{i=0}^{31} (x \overset{rot}{<<} i) \tag{2}$$

W powyższej formule operacje dodawania są wykonywane modulo wielkość słowa a wynik końcowy jest liczbą całkowitą w formacie uzupełnieniu dwójkowego. Algorytm ten prezentujemy jako ciekawostkę. Na większości komputerów byłby zupełnie bezużyteczny, ponieważ wymaga wykonania 31 przebiegów pętli, co daje 63 instrukcje plus dodatkowo instrukcje kontrolne pętli.

Aby sprawdzić, dlaczego formuła (2) działa, zastanówmy się co dzieje się z każdym bitem liczby x. Zostaje on kolejno przesunięty na każdą z pozycji słowa a jeśli dodamy do siebie te wszystkie 32 liczby uzyskamy w wyniku słowo składające się z samych jedynek — czyli o wartości –1. Aby to zademonstrować, zastosujmy 6-bitowe słowo x = **001001** (binarnie):

$$\begin{array}{ll} \mathbf{001001} & x \\ \mathbf{010010} & x \overset{rot}{<<} \mathbf{1} \\ \mathbf{100100} & x \overset{rot}{<<} \mathbf{2} \\ \mathbf{001001} & x \overset{rot}{<<} \mathbf{3} \\ \mathbf{010010} & x \overset{rot}{<<} \mathbf{4} \\ \mathbf{100100} & x \overset{rot}{<<} \mathbf{5} \end{array}$$

Oczywiście równie dobrze zadziałałaby operacja *przesunięcia cyklicznego w prawo*.

Metoda z formuły (1) jest bardzo podobna do tej techniki „przesuwania cyklicznego i sumowania", co stanie się jasne, gdy przekształcimy nieco formułę (1):

$$\mathrm{pop}(x) = x - \sum_{i=0}^{31} (x \overset{u}{>>} i)$$

Daje to nam nieco lepszy algorytm od równania (2). Jest lepszy, ponieważ wykorzystuje instrukcję *przesunięcia w prawo*, która jest powszechniej dostępna od instrukcji *przesunięcia cyklicznego* a także dlatego, że pętla może zostać zatrzymana w przypadku, gdy przesuwana wartość osiągnie wartość 0. Dzięki temu kod kontrolny pętli znacznie się uprości i w niektórych przypadkach możliwe jest zaoszczędzenie kilku iteracji. Obydwa algorytmy zestawiamy dla porównania na listingu 5.3.

Listing 5.3. *Dwa podobne algorytmy zliczające jedynki*

```
int pop(unsigned x) {
   int i, sum;

// metoda przesunięcia          // metoda przesunięcia w prawo i odejmowania
// cyklicznego i dodawania
   sum = x;                     // sum = x;
   for (i = 1; i <= 31; i++) {  // while (x != 0) {
      x = rotatel(x, 1);        //    x = x >> 1;
      sum = sum + x;            //    sum = sum - x;
   }                            // }
   return -sum;                 // return sum;
}
```

Nieco mniej interesujący algorytm możemy uzyskać wykorzystując *tabelę przeglądową* (ang. *lookup table*) zawierającą wartości instrukcji pop(x) dla określonego przedziału, na przykład od 0 do 255. W celu zliczenia liczby jedynek w słowie musimy dla każdego pola 8-bitowego odczytać wartość z tej tablicy (cztery razy) a następnie dodać odczytane wartości. Wersja tego algorytmu bez rozgałęzień może mieć następującą postać:

```
int pop(unsigned x) {                // tablica przeglądowa.
   static char table[256] = {
      0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4,
      ...
      4, 5, 5, 6, 5, 6, 6, 7, 5, 6, 6, 7, 6, 7, 7, 8};

   return table[x         & 0xFF] +
          table[(x >>  8) & 0xFF] +
          table[(x >> 16) & 0xFF] +
          table[(x >> 24)];
}
```

Pod pozycją 167 w [HAK] znajduje się krótki algorytm zliczający liczbę jedynek w 9-bitowym polu wyrównanym do prawej i wyizolowanym w rejestrze. Algorytm ten działa wyłącznie na maszynach o rozmiarze rejestru równym 36 bitów lub większym. Poniżej prezentujemy wersję tego algorytmu działającą na maszynach 32-bitowych, ale wyłącznie dla 8-bitowych wartości:

```
x = x * 0x08040201;  // robimy 4 kopie
x = x >> 3;          // kolejny krok wydobywa właściwe bity
x = x & 0x11111111;  // odczytujemy co 4 bit
x = x * 0x11111111;  // sumujemy cyfry (0 lub 1)
x = x >> 28;         // wyrównujemy wynik
```

Wersja dla 7-bitowych bloków:

```
x = x * 0x02040810;  // robimy 4 kopie, wyrównane do lewej
x = x & 0x11111111;  // odczytujemy co 4 bit
x = x * 0x11111111;  // sumujemy cyfry (0 lub 1)
x = x >> 28;         // wyrównujemy wynik
```

W powyższych procedurach ostatni krok można zastąpić obliczeniem reszty z dzielenia `x` przez 15.

Nie są to zbyt dobre algorytmy. Większość programistów zapewne wybrałoby metodę opierającą się na tablicy przeglądowej. Jednakże drugi z powyższych algorytmów posiada wersję wykorzystującą 64-bitową arytmetykę, która może być użyteczna w przypadku wykorzystania 64-bitowej maszyny posiadającej szybkie instrukcje *mnożenia*. Argumentem tego mnożenia jest 15-bitowa liczba (nie sądzę, żeby mogła istnieć wersja tego algorytmu dla liczb 16-bitowych, chyba że istniałaby pewność, iż nie wszystkie bity w tej liczbie są równe 1). Typ danych `long long` jest dostępny w rozszerzeniach standardu wprowadzonych w GNU C [Stall]. Jest to typ danych o dwukrotnie większym rozmiarze w stosunku do typu `int`, w naszym przypadku ma rozmiar 64 bitów. Przyrostek `ULL` w literałach liczbowych sygnalizuje, że liczba jest stałą typu `unsigned long long`.

```
int pop(unsigned x) {
   unsigned long long y;
   y = x * 0x0002000400080010ULL;
   y = y & 0x1111111111111111ULL;
   y = y * 0x1111111111111111ULL;
   y = y >> 60;
   return y;
}
```

Zliczanie jedynek w tablicy

Najprostszy sposób zliczenia jedynek w tablicy słów (w przypadku, gdy nie są dostępne instrukcje *zliczania populacji*) polega na zastosowaniu algorytmu z listingu 5.1 na stronie 86 dla każdego ze słów w tablicy, a następnie prostym zsumowaniu wyników jednostkowych.

Kolejny sposób, być może szybszy, polega na zastosowaniu dwóch wierszy tej funkcji na grupach trzech słów z tablicy, sumując jednocześnie trzy wyniki jednostkowe. Ponieważ każdy wynik jednostkowy w każdym z 4-bitowych pól może wynosić co najwyżej 4, zsumowanie trzech wyników w każdym z 4-bitowych pól da liczbę o maksymalnej wartości równej 12 a więc nie powodujących przepełnienia zakresu do pola po lewej stronie. Następnie każdy z wyników częściowych można przekształcić w słowo zawierające w każdym z czterech 8-bitowych pól liczbę o maksymalnej wartości 24. W tym celu zastosujemy następujące wyrażenie:

```
x = (x & 0x0F0F0F0F) + ((x >> 4) & 0x0F0F0F0F);
```

W trakcie generowania tych sum pośrednich możemy dodawać je do siebie dopóki ich suma jest mniejsza od 255 a więc pozwala to na bezpieczne zsumowanie 10 takich słów ($\lfloor 255/24 \rfloor$), zanim nie zostaniemy zmuszeni do przekształcenia tej sumy na słowo zawierające pola 16-bitowe przechowujące sumy częściowe o maksymalnej wartości 240. Możemy tego dokonać za pomocą następującego wyrażenia:

```
x = (x & 0x00FF00FF) + ((x >> 8) & 0x00FF00FF);
```

W ten sposób możemy dodać co najwyżej 273 takie słowa ($\lfloor 65535/240 \rfloor$), zanim nie zostaniemy zmuszeni do przekształcenia tego słowa do słowa zawierającego pojedyncze, 32-bitowe pole zawierające sumę bitów we wszystkich przetworzonych do tej pory słowach macierzy:

```
x = (x & 0x0000FFFF) + (x >> 16);
```

W praktyce instrukcje kontrolne pętli drastycznie uszczuplają wszelkie zaoszczędzone instrukcje, zatem zapewne zastosowanie tej procedury na opisaną skalę jest przerostem formy nad treścią. Listing 5.4 przedstawia opisaną koncepcję wykorzystując tylko jeden poziom pośredni. W pierwszej kolejności tworzy cztery 8-bitowe sumy pośrednie. Następnie, po dodaniu maksymalnej ilości tych sum tworzona jest suma całkowita. Maksymalną liczbą słów, które można zsumować w ten sposób bez obawy o przepełnienie 8-bitowego pola sumy częściowej jest $\lfloor 255/8 \rfloor = 31$.

Listing 5.4. *Zliczanie jedynek w macierzy*

```
int pop_array(unsigned A[], int n) {

   int i, j, lim;
   unsigned s, s8, x;

   s = 0;
   for (i = 0; i < n; i = i + 31) {
```

```
        lim = min(n, i + 31);
        s8 = 0;
        for (j = i; j < lim; j++) {
           x = A[j];
           x = x - ((x >> 1) & 0x55555555);
           x = (x & 0x33333333) + ((x >> 2) & 0x33333333);
           x = (x + (x >> 4)) & 0x0F0F0F0F;
           s8 = s8 + x;
        }
        x = (s8 & 0x00FF00FF) + ((s8 >> 8) & 0x00FF00FF);
        x = (x & 0x0000FFFF) + (x >> 16);
        s = s + x;
     }
     return s;
  }
```

Algorytm z listingu 5.4 został porównany z algorytmem wykorzystującym prostą pętlę. Obydwa kody skompilowano za pomocą kompilatora GCC na maszynie bardzo przypominającej nasz wzorcowy procesor RISC. W wyniku uzyskano średnio 22 instrukcje na słowo w przypadku metody wykorzystującej kilka pętli oraz 17,6 instrukcji na słowo w przypadku kodu z listingu 5.4, czyli uzyskano oszczędność rzędu 20%.

Zastosowania

Funkcja *zliczania populacji* może znaleźć zastosowanie w wyliczaniu „odległości Hamminga" (ang. *Hamming distance*) pomiędzy dwoma wektorami bitowymi. Jest to koncepcja z teorii *kodów korekcyjnych* (ang. *error-correcting codes*). Odległość Hamminga jest po prostu liczbą pozycji, na których dwa wektory różnią się wzajemnie. A konkretnie:

$$\mathrm{dist}(x, y) = \mathrm{pop}(x \oplus y).$$

Więcej szczegółów na temat tej teorii można znaleźć w pozycji [Dewd], w rozdziale dotyczącym kodów korekcyjnych.

Innym zastosowaniem może być technika dość szybkiego dostępu do bezpośrednio indeksowanej, stosunkowo rzadko wypełnionej tablicy A zapisanej w zwartej reprezentacji. Reprezentacja ta polega na tym, że zapisywane są wartości wyłącznie niepustych elementy takiej tablicy. Dodatkowo wykorzystujemy tablicę ciągów bitowych `bits`, składającą się z 32-bitowych słów, z których każde posiada jedynkę na pozycji i, takiej, że $A[i]$ jest zdefiniowane. Jako mechanizm przyśpieszający wykorzystujemy dodatkowo tablicę słów `bitsum` taką, że `bitsum[j]` zawiera liczbę jedynek we wszystkich elementach macierzy `bits` o indeksach mniejszych od j. Poniżej przedstawiamy przykład takiej macierzy, w której zdefiniowane są elementy 0, 2, 32, 47, 48 oraz 95.

bits	bitsum	dane
0x00000005	0	$A[0]$
0x00018001	2	$A[2]$

bits	bitsum	dane
0x80000000	5	$A[32]$ $A[47]$ $A[48]$ $A[95]$

Dla indeksu i, $0 \le i \le 95$, dostępny jest odpowiadający mu indeks `sparse_i` w tabeli danych. Indeks ten obliczamy na podstawie liczby jedynek w tabeli `bits` poprzedzających bit na pozycji i. Możemy to wyliczyć w następujący sposób:

```
j = i >> 5;                    // j = i/32.
k = i & 31;                    // k = rem(i, 32);
mask = 1 << k;                 // jedynka na poz. k
if ((bits[j] & mask) == 0) goto no_such_element;
mask = mask - 1;               // jedynki na prawo od k
sparse_i = bitsum[j] + pop(bits[j] & mask);
```

Koszt takiej reprezentacji wynosi dwa bity na element pełnej tablicy.

Jeszcze innym zastosowaniem funkcji zliczania populacji jest wyliczanie liczby końcowych zer w słowie. Więcej informacji na ten temat można znaleźć w podrozdziale 5.4 *Zliczanie zer końcowych* na stronie 104.

5.2. Parzystość

Parzystość (ang. *parity*) ciągu bitów polega na określeniu, czy zawiera on parzystą, czy też nieparzystą liczbę jedynek. Ciąg jest nieparzysty, gdy zawiera nieparzystą liczbę jedynek, w przeciwnym wypadku jest parzysty.

Wyliczanie parzystości słowa

W tym przypadku wygenerujemy liczbę 1, jeśli słowo x jest nieparzyste a liczbę 0 — jeśli jest parzyste. Jest to suma modulo 2 bitów w słowie x — innymi słowy jest to *różnica symetryczna* wszystkich bitów w x.

Jednym ze sposobów obliczenia tej wartości jest wyliczenie pop(x). Parzystość słowa x określa najmniej znaczący bit wyniku. Metoda ta byłaby zadowalająca w przypadku, gdy wykorzystywany procesor obsługuje instrukcję *zliczania populacji*, jeśli tak nie jest, istnieją lepsze sposoby określania parzystości słowa.

Dość bezpośredni sposób polega na wyliczeniu różnicy symetrycznej bitów w słowie:

$$y \leftarrow \bigoplus_{i=0}^{n-1} (x \overset{u}{>>} i)$$

W powyższej formule n określa rozmiar słowa, parzystość jest określona najmniej znaczącym bitem zmiennej y. W powyższym wzorze symbol $\oplus$ oznacza *różnicę symetryczną*, lecz dla tej konkretnej formuły wystarczy zwykłe dodawanie.

Parzystość słowa o większej długości n można określić o wiele szybciej w następujący sposób. Poniżej prezentujemy odpowiedni algorytm, dla $n = 32$. Przesunięcia mogą być ze znakiem lub bez znaku:

```
y = y ^ (x >> 1);
y = y ^ (y >> 2);
y = y ^ (y >> 4);                                    (3)
y = y ^ (y >> 8);
y = y ^ (y >> 16);
```

Powyższy kod wymaga 10 instrukcji, natomiast metoda różnicy symetrycznej wymaga ich aż 62, nawet bez zastosowania pętli. Ponownie bitem parzystości jest tutaj najmniej znaczący bit wyniku y. W rzeczywistości w przypadku zastosowania przesunięcia bez znaku i-ty bit zmiennej y określa parzystość bitów w x na lewo od i wraz z bitem o indeksie i. Ponadto z faktu, że operacja różnicy symetrycznej stanowi swoją inwersję, $x_i \oplus x_j$ określa parzystość przedziału od $i - 1$ do j, gdzie $i \geq j$.

Jest to przykład operacji „równoległego prefiksu" (ang. *parallel prefix*) lub „skanowania" (ang. *scan*), która ma zastosowanie w przetwarzaniu równoległym [KRS, HS]. Mając do dyspozycji odpowiednią liczbę procesorów za pomocą tej operacji można przekształcić procesy o czasie wykonania rzędu $O(n)$ na proces o czasie wykonania rzędu $O(\log_2 n)$. Na przykład w przypadku tablicy słów, na której chcemy wykonać operację skanowania za pomocą *różnicy symetrycznej*, możemy zastosować formułę (3) dla każdego ze słów w tablicy a następnie zastosować praktycznie tę samą technikę na tablicy, wykonując instrukcje *różnicy symetrycznej* na słowach z tablicy. Operacja tego typu wykorzystuje więcej elementarnych (operujących na argumentach o długości słowa) operacji *różnicy symetrycznej* od prostej operacji typu „od lewej do prawej", dlatego nie stanowi dobrego pomysłu w przypadku maszyn jednoprocesorowych. Lecz na komputerze o dużej liczbie procesorów zadanie to może zostać wykonane w czasie $O(\log_2 n)$ zamiast $O(n)$, gdzie n jest liczbą słów w tablicy.

Bezpośrednim zastosowaniem formuły (3) jest przekształcenie liczby całkowitej na zapis w kodzie Graya (zob. rozdział 13., strona 263).

W przypadku, gdyby formułę (3) zmodyfikować tak, aby wykorzystywała przesunięcia w lewo, parzystość całego słowa x określałby najbardziej znaczący bit, zatem i-ty bit w zmiennej y określałby parzystość bitów w x *na prawo* i razem z pozycją i.

Gdyby natomiast zastosować *przesunięcie cykliczne*, w wyniku uzyskalibyśmy liczbę złożoną z samych jedynek dla przypadku nieparzystości liczby x lub też liczbę o wartości 0 w przeciwnym wypadku.

Następujący kod wykorzystuje dziewięć instrukcji i wylicza parzystość liczby x w postaci liczby 0 lub 1 (przesunięcia są bez znaku):

```
x = x ^ (x >> 1);
x = (x ^ (x >> 2)) & 0x11111111;
x = x*0x11111111;
p = (x >> 28) & 1;
```

Po wykonaniu drugiego wiersza w powyższym kodzie każda cyfra szesnastkowa w x ma wartość 0 lub 1, w zależności od parzystości bitów w tej cyfrze szesnastkowej. *Mnożenie* powoduje zsumowanie tych cyfr, umieszczając wynik w bardziej znaczącej cyfrze szesnastkowej. Nie ma obawy o wystąpienie przeniesienia w kolumnie cyfry szesnastkowej, ponieważ maksymalna suma w takiej kolumnie wynosi 8.

Instrukcje *mnożenia* i *przesunięcia* można zastąpić instrukcją *wyliczania reszty z dzielenia* x przez 15, co dałoby rozwiązanie problemu w ośmiu instrukcjach (z poprawką na czas wykonania instrukcji *reszty z dzielenia*) pod warunkiem, że maszyna obsługuje instrukcję *reszty z dzielenia przez wartość bezpośrednią*.

Dodawanie bitu parzystości do wartości 7-bitowej

Pozycja 167 w [HAK] zawiera nowatorskie wyrażenie pozwalające na uzupełnienie o bit parzystości 7-bitowej wartości wyrównanej do prawej i odseparowanej w rejestrze. Polega to na umieszczeniu bitu na lewo od siedmiu bitów w celu utworzenia liczby 8-bitowej z parzystą liczbą jedynek. Kod oryginalnej w publikacji dotyczył maszyny 36-bitowej, lecz działa również na maszynach 32-bitowych.

$$\text{modu}((x * \mathbf{0x10204081}) \,\&\, \mathbf{0x888888FF}, \mathbf{1920})$$

W powyższym wzorze modu(a, b) oznacza resztę z dzielenia liczby a przez b, argumenty i wynik dzielenia są traktowane jak liczby całkowite bez znaku. Znak „*" określa mnożenie modulo 2^{32}, natomiast stała 1920 jest równa $15 \cdot 2^7$. W rzeczywistości formuła ta oblicza sumę bitów w liczbie x i umieszcza ją na lewo od siedmiu bitów, z których składa się liczba x. Wyrażenie to przekształca na przykład liczbę **0x0000007F** w liczbę **0x000003FF**, a liczbę **0x00000055** przekształca w liczbę **0x00000255**.

Inna niesamowita formuła z [HAK] uzupełnia liczbę 7-bitową do 8-bitowej liczby z nieparzystą liczbą jedynek:

$$\text{modu}((x * \mathbf{0x00204081}) \mid \mathbf{0x3DB6DB00}, \mathbf{1152})$$

Liczba $1152 = 9 \cdot 2^7$. Aby zrozumieć działanie tej formuły, należy uświadomić sobie, że potęgi 8 są ± 1 modulo 9. Gdybyśmy liczbę **0x3DB6DB00** w tym wzorze zastąpili liczbą **0xBDB6DB00**, formuła ta uzupełniałaby liczbę do liczby z parzystą liczbą bitów.

Metody te nie są praktyczne na współczesnych maszynach, gdzie pamięć jest tania a instrukcje *dzielenia* nadal są powolne. Większość programistów wyliczyłaby te funkcje za pomocą prostej techniki tablicy przeglądowej.

Zastosowanie

Operacja *parzystości* jest użyteczna podczas mnożenia macierzy bitowych w GF(2)[1] (gdzie operacja *sumy* jest równoważna operacji *różnicy symetrycznej*).

5.3. Zliczanie zer wiodących

Istnieje kilka prostych sposobów na zliczanie zer wiodących za pomocą przeszukiwania binarnego. Poniżej prezentujemy model posiadający kilka odmian. Algorytmy te wykonują się w 20 do 29 instrukcji na podstawowym zestawie instrukcji RISC. Porównania są typu „logicznego" (liczby całkowite bez znaku).

```
if (x == 0) return(32);
n = 0;
if (x <= 0x0000FFFF) {n = n +16; x = x <<16;}
if (x <= 0x00FFFFFF) {n = n + 8; x = x << 8;}
if (x <= 0x0FFFFFFF) {n = n + 4; x = x << 4;}
if (x <= 0x3FFFFFFF) {n = n + 2; x = x << 2;}
if (x <= 0x7FFFFFFF) {n = n + 1;}
return n;
```

Jedna z odmian polega na zastąpieniu porównań *koniunkcjami*:

```
if ((x & 0xFFFF0000) == 0) {n = n + 16; x = x << 16;}
if ((x & 0xFF000000) == 0) {n = n + 8; x = x << 8;}
...
```

Inna z odmian stosuje *przesunięcia w prawo*, dzięki czemu unika się stosowania dużych wartości bezpośrednich.

Ostatnie wyrażenie `if` polega na dodaniu 1 do `n` jeśli najbardziej znaczący bit `n` jest równy 0, zatem możemy zastosować rozwiązanie alternatywne, które oszczędza nam jednego rozgałęzienia:

```
n = n + 1 - (x >> 31);
```

W powyższym przypisaniu możemy pominąć fragment „+ 1", jeśli `n` zainicjujemy wartością 1 zamiast 0. Wszystkie powyższe obserwacje doprowadzają nas do algorytmu (potrzebującego 12 do 20 instrukcji z podstawowego zestawu RISC) przedstawione na listingu 5.5.

Listing 5.5. *Zliczanie zer wiodących, przeszukiwanie binarne*

```
int nlz(unsigned x) {
  int n;
```

[1] Czyli w polu Galois (ang. *Galois field*) GF(2) — *przyp. red.*

```
if (x == 0) return(32);
   n = 1;
   if ((x >> 16) == 0) {n = n +16; x = x <<16;}
   if ((x >> 24) == 0) {n = n + 8; x = x << 8;}
   if ((x >> 28) == 0) {n = n + 4; x = x << 4;}
   if ((x >> 30) == 0) {n = n + 2; x = x << 2;}
   n = n - (x >> 31);
   return n;
}
```

Kolejne usprawnienie jest możliwe w przypadku, gdy najbardziej znaczący bit w x ma wartość 1. W tym celu zmieniamy pierwszy wiersz kodu na następujący:

```
if ((int)x <= 0) return (~x >> 26) & 32;
```

Listing 5.6 zawiera algorytm w pewnym sensie odwrotny do powyższego. Im więcej zer w badanej liczbie, tym mniej instrukcji wykorzystuje. Ponadto uniknięto w nim stosowania dużych wartości bezpośrednich oraz dużych przesunięć. Na standardowym zestawie instrukcji RISC algorytm ten wykorzystuje 12 do 20 instrukcji.

Listing 5.6. *Zliczanie zer wiodących, przeszukiwanie binarne, odliczanie w dół*

```
int nlz(unsigned x) {
   unsigned y;
   int n;

   n = 32;
   y = x >>16;  if (y != 0) {n = n -16;  x = y;}
   y = x >> 8;  if (y != 0) {n = n - 8;  x = y;}
   y = x >> 4;  if (y != 0) {n = n - 4;  x = y;}
   y = x >> 2;  if (y != 0) {n = n - 2;  x = y;}
   y = x >> 1;  if (y != 0) return n - 2;
   return n - x;
}
```

Algorytm ten można zmodyfikować o wykorzystanie tablicy przeglądowej. Ostatnie cztery wiersze kodu należy w nim zastąpić przez następujące:

```
static char table[256] = {0, 1, 2, 2, 3, 3, 3, 3, 4, 4, ..., 8};
return n - table[x];
```

Wiele algorytmów można usprawnić za pomocą podobnej tablicy, lecz nie będziemy często o tym wspominać w niniejszej książce.

W celu zmniejszenia rozmiaru kodu przedstawione powyżej algorytmy można zakodować z wykorzystaniem pętli. Na przykład algorytm z listingu 5.6 można zastąpić algorytmem z listingu 5.7. Wymaga on 23 do 33 instrukcji z podstawowego zestawu RISC, z czego dziesięć to instrukcje rozgałęzień warunkowych.

Listing 5.7. *Zliczanie zer wiodących, przeszukiwanie binarne, odliczanie w dół w formie pętli*

```
int nlz(unsigned x) {
   unsigned y;
   int n, c;

   n = 32;
   c = 16;
   do {
      y = x >> c;  if (y != 0) {n = n - c;  x = y;}
      c = c >> 1;
   } while (c != 0);
   return n - x;
}
```

Można oczywiście przesuwać w lewo o jedno miejsce zliczając zera aż do momentu, gdy wystąpi jedynka w bicie znaku. Można również przesuwać w prawo o jedno miejsce na raz aż do momentu, gdy słowo ma wartość 0. Algorytmy te są zwięzłe i działają sprawnie w przypadku, gdy liczba wiodących zer jest, odpowiednio, mała lub duża. Można również połączyć te metody, tak jak na listingu 5.8.

Listing 5.8. *Zliczanie zer wiodących, jednoczesne zliczanie od obydwu stron*

```
int nlz(int x) {
   int y, n;

   n = 0;
   y = x;
L: if (x < 0) return n;
   if (y == 0) return 32 - n;
   n = n + 1;
   x = x << 1;
   y = y >> 1;
   goto L;
}
```

Algorytm z listingu 5.8 demonstruje ogólną sytuację, gdy użyteczne jest zastosowanie kombinacji dwóch algorytmów, wybierając wynik tego z nich, który pierwszy zakończy działanie. Technika taka pozwala na utworzenie kodu, który działa bardzo szybko na superskalarnych maszynach lub też na maszynach klasy VLIW. Dzieje się tak, ponieważ instrukcje niezależne od siebie znajdują się w bezpośrednim sąsiedztwie. Maszyny we wspomnianych rodzajach architektury są w stanie wykonywać dwie lub większą liczbę instrukcji jednocześnie, pod warunkiem, że są one wzajemnie niezależne.

Na maszynie typu RISC z podstawowym zestawem instrukcji kod ten wykonuje się w $\min(3 + 6\text{nlz}(x), 5 + 6(32 - \text{nlz}(x)))$ instrukcjach, w najgorszym przypadku w 99. Można sobie jednak wyobrazić skrajny przypadek, gdy komputer superskalarny lub klasy VLIW wykonuje cały ten algorytm w jednym cyklu, pod warunkiem, że porównania zostaną zrealizowane jako produkt uboczny operacji *przesunięcia*, w przeciwnym wypadku w dwóch cyklach, plus nakład na rozgałęzienie.

Przekształcenie algorytmów z listingu 5.5 lub 5.6 do wersji pozbawionej rozgałęzień jest raczej intuicyjne. Listing 5.9 zawiera wersję algorytmu wykonującego się w 28 instrukcjach podstawowego zestawu instrukcji RISC.

Listing 5.9. *Zliczanie zer wiodących, przeszukiwanie binarne bez rozgałęzień*

```
int nlz (unsigned x) {
   int y, m, n;

   y = -(x >> 16);       // Jeśli lewa połowa x jest równa 0,
   m = (y >> 16) & 16;   // ustawiamy n = 16. W przeciwnym wypadku
   n = 16 - m;           // ustawiamy n = 0 a następnie
   x = x >> m;           // przesuwamy x w prawo o 16.
                         // Teraz x ma postać 0000xxxx.
   y = x - 0x100;        // Jeśli pozycje 8-15 są równe 0,
   m = (y >> 16) & 8;    // dodajemy 8 do n i przesuwamy x w lewo o 8.
   n = n + m;
   x = x << m;

   y = x - 0x1000;       // Jeśli pozycje 12-15 są równe 0,
   m = (y >> 16) & 4;    // dodajemy 4 do n i przesuwamy x w lewo o 4.
   n = n + m;
   x = x << m;

   y = x - 0x4000;       // jeśli pozycje 14-15 są równe 0,
   m = (y >> 16) & 2;    // dodajemy 2 do n i przesuwamy x w lewo o 2.
   n = n + m;
   x = x << m;

   y = x >> 14;          // Ustawiamy y = 0, 1, 2, lub 3.
   m = y & ~(y >> 1);    // Odpowiednio m = 0, 1, 2, lub 2.
   return n + 2 - m;
}
```

Jeśli wykorzystywana maszyna udostępnia instrukcję *zliczania liczebność populacji*, dobrym sposobem wyliczenia liczby zer wiodących jest funkcja przedstawiona na listingu 5.10.

Listing 5.10. *Zliczanie zer wiodących, propagacja w prawo i zliczanie jedynek*

```
int nlz(unsigned x) {
   int pop(unsigned x);

   x = x | (x >> 1);
   x = x | (x >> 2);
   x = x | (x >> 4);
   x = x | (x >> 8);
   x = x | (x >>16);
   return pop(~x);
}
```

Kolejność pięciu przypisań do x może zostać odwrócona lub wręcz mieć zupełnie dowolną kolejność. Kod ten jest napisany bez rozgałęzień i wykorzystuje 11 instrukcji. Nawet w przypadku, gdy w procesorze nie jest dostępna instrukcja *zliczania liczebność*

populacji, algorytm ten może nadal być użyteczny. Wykorzystując kod z wydruku 5.1 (strona 86) działający w 21 instrukcjach, algorytm ten będzie wykonywać się w 32 instrukcjach podstawowego zestawu RISC, bez rozgałęzień.

Metody zmiennoprzecinkowe

W celu zliczania liczby zer wiodących można wykorzystać mechanizmy normalizacyjne obliczeń zmiennoprzecinkowych. Sztuczka ta działa całkiem dobrze z wykorzystaniem liczb zmiennoprzecinkowych w formacie IEEE. Pomysł polega na przekształceniu liczby całkowitej bez znaku na liczbę zmiennoprzecinkową podwójnej precyzji, wyodrębnieniu wykładnika i odjęciu go od stałej. Na listingu 5.11 przedstawiona została kompletna procedura wykorzystująca ten pomysł.

Listing 5.11. *Zliczanie zer wiodących, wykorzystanie funkcji zmiennoprzecinkowych IEEE*

```
int nlz(unsigned k) {
   union {
      unsigned asInt[2];
      double asDouble;
   };
   int n;

   asDouble = (double)k + 0.5;
   n = 1054 - (asInt[LE] >> 20);
   return n;
}
```

Kod wykorzystuje opcję języka C++ polegającą na zastosowaniu *anonimowej unii* (ang. *anonymous union*), w której nakładamy na liczbę całkowitą liczbę zmiennoprzecinkową podwójnej precyzji. Zmienna LE musi mieć wartość 1 podczas wykonywania kodu na maszynie typu *little-endian*, 0 na maszynie *big-endian*[2]. Dodanie do liczby wartości 0,5 lub innej wartości mniejszej od 1 jest niezbędne w przypadku, gdy $k = 0$.

Nie ma możliwości oszacowania czasu wykonania tego kodu, ponieważ maszyny różnią się znacznie pod względem możliwości operacji zmiennoprzecinkowych. Na przykład wiele maszyn posiada rejestry zmiennoprzecinkowe niezależnie od rejestrów całkowitych i na takich maszynach w celu przekształcenia wartości całkowitej na zmiennoprzecinkową może być konieczny transfer wartości do rejestru zmiennoprzecinkowego za pośrednictwem pamięci a następnie ponowne przeniesienie wyniku do rejestru całkowitoliczbowego.

Kod na wydruku 5.11 nie jest prawidłowy w świetle standardów ANSI C oraz C++, ponieważ odwołuje się do tego samego adresu w pamięci w interpretacji dwóch różnych typów liczbowych. Z tego powodu nie ma pewności, że kod ten skompilowany

[2] *Little-endian*, czyli z rosnącym porządkiem bitów (najpierw młodszy); *big-endian*, czyli z malejącym porządkiem bitów (najpierw starszy) — *przyp. red.*

określonym kompilatorem będzie działać na określonej maszynie. Kod działa skompilowany kompilatorem XLC w systemie AIX firmy IBM, skompilowany kompilatorem GCC w systemie AIX i pod Windows 2000, z wykorzystaniem wszystkich poziomów optymalizacji (w każdym razie w wersjach tych narzędzi dostępnych w okresie tworzenia książki). Procedura przestanie jednak działać, jeśli zmodyfikujemy kod w następujący sposób:

```
xx = (double)k + 0.5;
n = 1054 - (*((unsigned *)&xx + LE) >> 20);
```

Kod ten dodatkowo jest w konflikcie z innym z wymogów standardu ANSI, mówiącym, iż działania arytmetyczne na wskaźnikach są dopuszczalne wyłącznie w odniesieniu do wskaźników do elementów tablic [Cohen]. Błąd wykonania jednak wynika z pierwszego pogwałcenia standardu, związanego z definicją nakładania się zmiennych.

Pomimo niewielkiej przydatności kodu[3] prezentujemy kilka wersji tego algorytmu:

```
asDouble = (double)k;
n = 1054 - (asInt[LE] >> 20);
n = (n & 31) + (n >> 9);      /* korekta problemu z k = 0 */

k = k & ~(k >> 1);            /* korekta problemu z zaokrągleniem */
asFloat = (float)k + 0.5f;
n = 158 - (asInt >> 23);

k = k & ~(k >> 1);            /* korekta problemu z zaokrągleniem */
asFloat = (float)k;
n = 158 - (asInt >> 23);
n = (n & 31) + (n >> 6);      /* korekta problemu z k = 0 */
```

W pierwszej modyfikacji problem z $k = 0$ został rozwiązany nie za pomocą dodania wartości 0,5, lecz za pomocą arytmetyki całkowitej na wyniku (który bez poprawki miałby wartość 1054, czyli 0x41E).

Kolejne dwie wersje wykorzystują liczby zmiennoprzecinkowe o pojedynczej precyzji z odpowiednią zmianą konstrukcji anonimowej unii. Pojawia się jednak nowy problem. Zaokrąglenia mogą spowodować utratę wyniku w przypadku, gdy aktywny jest tryb zaokrągleń do najbliższej liczby (najczęściej stosowany) lub też do $+\infty$. W trybie zaokrąglenia do najbliższej liczby problem występuje w przypadkach k należącego do zakresu od 0xFFFFFF80 do 0xFFFFFFFF, od 0x7FFFFFC0 do 7FFFFFFF, 0x3FFFFFE0 do 0x3FFFFFFF itd. Przy zaokrągleniu dodanie 1 powoduje przeniesienia w lewo do końca, co powoduje, że pozycja najbardziej znaczącej jedynki zmienia się. Proponowana poprawka powoduje wyczyszczenie bitu na prawo od najbardziej znaczącej jedynki, co powstrzymuje cykl przeniesień.

Kompilator GNU C/C++ posiada unikalną możliwość pozwalającą na zakodowanie całych podanych formuł w postaci makra, co powoduje, że w miejscu wywołania go uzyskujemy funkcję osadzoną (ang. *in-line*). Sekwencja instrukcji powinna zwykle

[3] Mała przydatność kodu związana jest ze sposobem zastosowania języka C. Przedstawione metody byłyby w pełni użyteczne, gdyby zostały zaimplementowane w języku maszynowym lub gdyby zostały wygenerowane przez kompilator zoptymalizowany dla konkretnej maszyny.

kończyć się wyrażeniem, które jest uznawane za wynik zwracany przez konstrukcję. Definicja makra tego typu została przedstawiona poniżej. Jest to reprezentacja pierwszej wersji algorytmu wykorzystującej wartość zmiennoprzecinkową pojedynczej precyzji. W języku C zwyczajowo nazwy makr zapisuje się wielkimi literami.

```
#define NLZ(kp) \
   ({union {unsigned _asInt; float _asFloat;}; \
     unsigned _k = (kp), _kk = _k & ~(_k >> 1); \
     _asFloat = (float)_kk + 0.5f; \
     158 - (_asInt >> 23);})
```

Podkreślenia w nazwach zostały zastosowane w celu uniknięcia konfliktów z parametrem k. Zakładamy, że nazwy zdefiniowane przez użytkownika nie rozpoczynają się od podkreśleń.

Związek z funkcją obliczającą logarytm

Funkcja „nlz" jest silnie związana z funkcją obliczającą logarytm o podstawie 2. Dla liczby całkowitej bez znaku $x \neq \mathbf{0}$ mamy następujące zależności:

$$\lfloor \log_2(x) \rfloor = \mathbf{31} - \mathrm{nlz}(x)$$

$$\lceil \log_2(x) \rceil = \mathbf{32} - \mathrm{nlz}(x - \mathbf{1})$$

Więcej informacji na ten temat można znaleźć w podrozdziale 11.4 *Logarytm całkowitoliczbowy*, strona 215.

Kolejną własnością silnie związaną z funkcją „nlz" jest *rozmiar bitowy* (ang. *bitsize*), to znaczy liczba bitów niezbędna do zapisania argumentu jako liczby ze znakiem w formacie uzupełnienia dwójkowego. Definicja funkcji bitsize(x) wygląda następująco:

$$\mathrm{bitsize}(x) = \begin{cases} 1, & x = -1 \text{ lub } 0, \\ 2, & x = -2 \text{ lub } 1, \\ 3, & -4 \le x \le -3 \text{ lub } 2 \le x \le 3, \\ 4, & -8 \le x \le -5 \text{ lub } 4 \le x \le 7, \\ \ldots & \ldots \\ 32, & -2^{31} \le x \le -2^{30} \text{ lub } 2^{30} \le x \le 2^{31} - 1, \end{cases}$$

Z tej definicji wynika, że bitsize(x) = bitsize($-x - 1$). Z własności $-x - \mathbf{1} = \neg x$ wynika jednak następujący algorytm (wykorzystujący przesunięcie ze znakiem):

```
x = x ^ (x >> 31);      // jeśli (x < 0) to x = -x - 1;
return 33 - nlz(x);
```

Zastosowania

Dwoma ważnymi zastosowaniami funkcji obliczającej *liczbę zer wiodących* w słowie są mechanizmy symulujące operacje zmiennoprzecinkowe oraz różne algorytmy dzielenia

(zob. listing 9.1 na stronie 141 oraz listing 9.3 na stronie 177). Instrukcja ta ma jednak wiele innych, bardzo różnorodnych zastosowań.

Funkcję te można zastosować w celu zbudowania predykatu $x = y$ w zaledwie trzech instrukcjach (zob. podrozdział 2.11 *Predykaty porównań*, strona 21) oraz jako mechanizm pomocniczy w tworzeniu pewnych funkcji elementarnych (zob. rozdział 11.).

Oryginalnym zastosowaniem jest generowanie liczb pseudolosowych o rozkładzie wykładniczym za pomocą generowania pseudolosowych liczb całkowitych o rozkładzie jednostajnym i obliczanie wyniku operacji „nlz" na takiej liczbie [GLS1]. Wynik równy 0 ma prawdopodobieństwo 1/2, 1 ma prawdopodobieństwo 1/4, 2 ma prawdopodobieństwo 1/8 itd. Jeszcze innym zastosowaniem operacji zliczania liczby zer wiodących jest przeszukiwanie słowa na okoliczność występowania ciągłych bloków jedynek (lub zer) o określonej długości. Jest to element niektórych algorytmów alokacji bloków na dysku. W przypadku tych dwóch ostatnich zastosowań można również wykorzystać funkcję *zliczającą liczbę zer końcowych*.

5.4. Zliczanie zer końcowych

W przypadku, gdy na maszynie dostępna jest operacja *zliczania zer wiodących*, najlepszym sposobem na *zliczenie zer końcowych* jest odwrócenie tego problemu:

$$\mathbf{32} - \mathrm{nlz}(\neg x \,\&\, (x - \mathbf{1})).$$

Jeśli dostępna jest funkcja *zliczania wielkości populacji* w słowie, nieco lepszym sposobem jest utworzenie maski identyfikującej zera końcowe a następnie zliczenie jedynek w tej masce [Hay2], na przykład w następujący sposób:

$$\mathrm{pop}(\neg x \,\&\, (x - 1))$$

lub:

$$\mathbf{32} - \mathrm{pop}(x \mid -x).$$

Istnieją również odmiany tych algorytmów wykorzystujące wyrażenia, które tworzą maskę identyfikującą zera końcowe w x, na przykład takie, jak przedstawiliśmy w podrozdziale 2.1 *Manipulowanie prawostronnymi bitami*, na stronie 35. Metody te są jak najbardziej sensowne, nawet w przypadku, gdy maszyna nie obsługuje żadnej z instrukcji zliczających bity. Wykorzystując algorytm funkcji pop(x) z listingu 5.1 (strona 86), pierwsze z powyższych wyrażeń potrzebuje 3 + 21 = 24 instrukcji bez rozgałęzień.

Listing 5.12 przedstawia algorytm obliczający liczbę końcowych zer w 12 do 20 instrukcji z podstawowego zestawu RISC (dla $x \neq 0$).

Listing 5.12. *Zliczanie zer końcowych, wyszukiwanie binarne*

```
int ntz(unsigned x) {
   int n;
```

```
    if (x == 0) return(32);
      n = 1;
      if ((x & 0x0000FFFF) == 0) {n = n +16; x = x >>16;}
      if ((x & 0x000000FF) == 0) {n = n + 8; x = x >> 8;}
      if ((x & 0x0000000F) == 0) {n = n + 4; x = x >> 4;}
      if ((x & 0x00000003) == 0) {n = n + 2; x = x >> 2;}
      return n - (x & 1);
    }
```

Jeśli to wyda się pomocne (a kompilator okaże się za mało inteligentny, żeby zrobić to automatycznie), wyrażenie `n + 16` możemy zastąpić liczbą 17. Nie będzie to jednak miało większego wpływu na liczbę wykonywanych instrukcji.

Listing 5.13 przedstawia odmianę algorytmu „ntz" wykorzystującą mniejsze wartości bezpośrednie i prostsze operacje. Algorytm ten wymaga od 12 do 21 instrukcji podstawowego zestawu RISC. W przeciwieństwie do listingu 5.12, w przypadku, gdy liczba zer końcowych jest mała, algorytm z listingu 5.13 wymaga większej liczby instrukcji, zawiera również większą liczbę rozgałęzień warunkowych.

Listing 5.13. *Zliczanie zer końcowych, mniejsze wartości bezpośrednie*

```
    int ntz(unsigned x) {
      unsigned y;
      int n;

      if (x == 0) return 32;
      n = 31;
      y = x <<16;  if (y != 0) {n = n -16;  x = y;}
      y = x << 8;  if (y != 0) {n = n - 8;  x = y;}
      y = x << 4;  if (y != 0) {n = n - 4;  x = y;}
      y = x << 2;  if (y != 0) {n = n - 2;  x = y;}
      y = x << 1;  if (y != 0) {n = n - 1;}
      return n;
    }
```

Wiersz poprzedzający instrukcję `return` możemy zastąpić następującym:

```
    n = n - ((x << 1) >> 31);
```

Dzięki temu możemy zaoszczędzić jedno rozgałęzienie, lecz nie oszczędzamy ani jednej instrukcji.

Trudno jest jednak znaleźć algorytm lepszy od tego, który jest oparty na *drzewie przeszukiwania* (ang. *search tree*) [Aus2], jeśli chodzi o liczbę wykonywanych instrukcji. Listing 5.14 przedstawia procedurę tego typu dla 8-bitowych argumentów. Procedura ta wykonuje się w siedmiu instrukcjach w większości przypadków, za wyjątkiem ostatnich dwóch (`return 7` oraz `return 8`), kiedy to wykonuje się w dziewięciu instrukcjach. Wersja 32-bitowa tego algorytmu będzie się wykonywała w od 11 do 13 instrukcjach. Niestety dla dużych rozmiarów słów prezentowany program znacznie się rozrasta. Przedstawiona wersja algorytmu dla wartości 8-bitowych wymaga 12 wierszy kodu źródłowego a po skompilowaniu zajmie około 41 instrukcji. Wersja 32-bitowa zajmie 48 wierszy kodu i około 164 instrukcji a wersja 64-bitowa będzie jeszcze dwukrotnie większa.

Listing 5.14. *Zliczanie zer końcowych, drzewo przeszukiwania binarnego*

```
int ntz(char x) {
   if (x & 15) {
      if (x & 3) {
         if (x & 1) return 0;
         else return 1;
      }
      else if (x & 4) return 2;
      else return 3;
   }
   else if (x & 0x30) {
      if (x & 0x10) return 4;
      else return 5;
   }
   else if (x & 0x40) return 6;
   else if (x) return 7;
   else return 8;
}
```

W przypadku, gdy oczekujemy, że liczba końcowych zer jest mała lub duża, całkiem szybkie rozwiązanie można uzyskać za pomocą prostych pętli (listing 5.15). Algorytm z lewej strony wymaga 5 + 3ntz(x) instrukcji, algorytm z prawej (w komentarzach) wymaga 3 + 3 (32 – ntz(x)) instrukcji z podstawowego zestawu RISC.

Listing 5.15. *Zliczanie zer końcowych, proste pętle zliczające*

```
int ntz(unsigned x) {
   int n;

   x = ~x & (x - 1);
   n = 0;                    // n = 32;
   while(x != 0) {           // while (x != 0) {
      n = n + 1;             //    n = n - 1;
      x = x >> 1;            //    x = x + x;
   }                         // }
   return n;                 // return n;
}
```

Interesującym spostrzeżeniem jest to, że w przypadku jednostajnego rozkładu liczb x średnia liczba zer końcowych wyniesie prawie dokładnie 1,0. Aby tego dowieść, zsumujmy iloczyny $p_i n_i$, gdzie p_i jest prawdopodobieństwem, że istnieje dokładnie n_i zer końcowych:

$$S \cong \frac{1}{2}\cdot 0 + \frac{1}{4}\cdot 1 + \frac{1}{8}\cdot 2 + \frac{1}{16}\cdot 3 + \frac{1}{32}\cdot 4 + \frac{1}{64}\cdot 5 + \ldots \cong \sum_{n=0}^{\infty} \frac{n}{2^{n+1}}$$

W celu wyliczenia tej sumy rozpatrzmy następującą tablicę:

1/4	1/8	1/16	1/32	1/64	...
	1/8	1/16	1/32	1/64	...
		1/16	1/32	1/64	...
			1/32	1/64	...
				1/64	...
					...

Suma w każdej z kolumn jest składnikiem szeregu S. Stąd S jest sumą wszystkich elementów w tablicy. W poszczególnych wierszach sumy są następujące:

$$1/4 + 1/8 + 1/16 + 1/32 + \ldots = 1/2$$

$$1/8 + 1/16 + 1/32 + 1/64 + \ldots = 1/4$$

$$1/16 + 1/32 + 1/64 + 1/128 + \ldots = 1/8$$

Suma tych sum wyniesie $1/2 + 1/4 + 1/8 + \ldots = 1$. Zbieżność bezwzględna oryginalnych szeregów uzasadnia przegrupowanie.

Czasem zdarza się, że potrzebna jest funkcja podobna do ntz(x), lecz różniąca się tym, że argument o wartości 0 stanowi specjalny przypadek, być może błąd, który powinien być zidentyfikowany za pomocą wyniku funkcji różniącego się znacznie od zwykłego wyniku funkcji. Załóżmy, że w następujący sposób zdefiniujemy funkcję „liczba czynników 2 w x":

$$\text{nfact2}(x) = \begin{cases} \text{ntz}(x), & x \neq 0, \\ -1, & x = 0. \end{cases}$$

Powyższą formułę można wyliczyć w następujący sposób:

$$31 - \text{nlz}(x \,\&\, -x)$$

Zastosowania

Źródło [GLS1] wskazuje kilka ciekawych zastosowań funkcji *zliczania liczby zer końcowych*. Jedno z nich nazwano „funkcją linijki" (ang. *ruler function*), ponieważ pozwala na określenie wysokości podziałki na linijce, którą podzielono na połówki, następnie na ćwiartki, części ósme itd.

Funkcja ta znajduje zastosowanie w algorytmie *wykrywania cykli* (ang. *loop detection*) Gospera. Opiszemy go dość szczegółowo, ponieważ jest to przykład eleganckiego rozwiązania, które może znaleźć zastosowanie w większej liczbie sytuacji, niż może się początkowo wydawać.

Załóżmy, że mamy do czynienia z ciągiem X_0, X_1, X_2 itd., zdefiniowanym zależnością $X_{n+1} = f(X_n)$. W przypadku, gdy zakres funkcji f jest ograniczony, ciąg ten jest z konieczności cykliczny. Oznacza to, że składa się z początkowego ciągu $X_0, X_1, \ldots, X_{\mu-1}$,

po którym następuje cykl $X_\mu, X_{\mu+1}, \ldots X_{\mu+\lambda-1}$ powtarzający się bez końca ($X_\mu = X_{\mu+\lambda}$, $X_{\mu+1} = X_{\mu+\lambda+1}$ itd., gdzie λ jest okresem tego cyklu). Mając daną funkcję f problem wykrywania cykliczności polega na odnalezieniu indeksu μ pierwszego elementu cyklu oraz okresu cyklu λ. Wykrywanie cykli ma zastosowanie w generatorach liczb pseudolosowych oraz w celu wykrywania cykli w listach powiązanych.

Można zapisać kolejne wartości ciągu i porównywać każdy kolejny element z poprzednimi. Dzięki temu można określić rozpoczęcie się nowego cyklu.

Prawdopodobnie najprostszy algorytm wykrywania cykli został zaproponowany przez R. W. Floyda [Knu2, podrozdz. 3.1. prob. 6]. Algorytm ten polega na analizie następującego procesu:

$$x = f(x)$$

$$y = f(f(y))$$

Zmienne x oraz y inicjujemy wartością X_0. Po n krokach $x = X_n$ oraz $y = X_{2n}$. Wartości te porównujemy i w przypadku, gdy są równe wiemy już, że X_n oraz X_{2n} są od siebie oddalone o wielokrotność okresu λ, to znaczy $2n - n = n$ jest wielokrotnością liczby λ. Następnie możemy określić μ, odtwarzając ciąg od początku i porównując X_0 z X_n, następnie X_1 z X_{n+1} itd. Równość wystąpi przy porównaniu X_μ z $X_{n+\mu}$. Na końcu określamy λ odtwarzając więcej elementów i porównując X_μ z $X_{\mu+1}, X_{\mu+2}, \ldots$. Algorytm ten wymaga niewielkiej ilości miejsca niezależnego od rozmiaru ciągu, lecz wymaga wielokrotnego wyliczania wartości f.

Algorytm Gospera [HAK poz. 132, Knu2, odpowiedzi do ćwiczeń w podrozdziale 3.1 problem 7] odnajduje okres λ, lecz nie odnajduje miejsca początkowego μ pierwszego cyklu. Podstawową zaletą tego algorytmu jest to, że nie musi on rozpoczynać od nowa wyliczania wartości f i jest dość efektywny pod względem czasu i miejsca. Rozmiar danych nie jest jednak stały, wymagana jest tablica o rozmiarze $\log_2(\Lambda) + 1$, gdzie Λ jest rozmiarem najdłuższego dopuszczalnego okresu. Nie jest to dużo miejsca, na przykład w przypadku okresu o maksymalnej długości 2^{32} wystarczy tablica o rozmiarze 33 słów.

Algorytm Gospera napisany w języku C można znaleźć na listingu 5.16. Ta funkcja w C oczekuje jako argumentów wskaźnika do analizowanej funkcji f oraz początkowej wartości X_0. Zwraca górne i dolne ograniczenie miejsca μ oraz okres λ.

Listing 5.16. *Algorytm Gospera wykrywający cykle*

```
void ld_Gosper(int (*f)(int), int X0, int *mu_l,
                              int *mu_u, int *lambda) {
   int Xn, k, m, kmax, n, lgl;
   int T[33];

   T[0] = X0;
   Xn = X0;
   for (n = 1; ; n++) {
      Xn = f(Xn);
```

```
        kmax = 31 - nlz(n);            // Floor(log2 n).
        for (k = 0; k <= kmax; k++) {
           if (Xn == T[k]) goto L;
        }
        T[ntz(n+1)] = Xn;              // brak dopasowania.
     }
  L:
     // obliczamy m = max{i | i < n and ntz(i+1) = k}.

     m = ((((n >> k) - 1) | 1) << k) - 1;
     *lambda = n - m;
     lgl = 31 - nlz(*lambda - 1);      // Ceil(log2 lambda) - 1.
     *mu_u = m;                        // górne ograniczenie mu.
     *mu_l = m - max(1, 1 << lgl) + 1;// dolne ograniczenie mu.
  }
```

Pomimo że algorytm Gospera nie wylicza wartości μ, potrafi określić zakres, w którym wartość ta występuje, taki że $\mu_u - \mu_l + 1 \leq \max(\lambda - 1, 1)$. Algorytm ten porównuje wartości X_n dla $n = 1, 2, \ldots$ z podzbiorem o rozmiarze $\lfloor \log_2 n \rfloor + 1$ elementów ciągu poprzedzających X_n. Porównanie odbywa się z wartością najbliższego elementu poprzedzającego X_i, takiego że reprezentacja binarna $i + 1$ kończy się jedynką (to znaczy i jest liczbą parzystą), następnie takiego najbliższego elementu poprzedzającego X_i, że reprezentacja binarna $i + 1$ kończy się zerem, następnie takiego najbliższego elementu poprzedzającego X_i, że reprezentacja binarna $i + 1$ kończy się dwoma zerami itd.

Z tego wynika następujący przebieg sekwencji porównań:

$X_1 : X_0$	$X_7 : X_6, X_5, X_3$	$X_{13} : X_{12}, X_9, X_{11}, X_7$
$X_2 : X_0, X_1$	$X_8 : X_6, X_5, X_3, X_7$	$X_{14} : X_{12}, X_{13}, X_{11}, X_7$
$X_3 : X_2, X_1$	$X_9 : X_8, X_5, X_3, X_7$	$X_{15} : X_{14}, X_{13}, X_{11}, X_7$
$X_4 : X_2, X_1, X_3$	$X_{10} : X_8, X_9, X_3, X_7$	$X_{16} : X_{14}, X_{13}, X_{11}, X_7, X_{15}$
$X_5 : X_4, X_1, X_3$	$X_{11} : X_{10}, X_9, X_3, X_7$	$X_{17} : X_{16}, X_{13}, X_{11}, X_7, X_{15}$
$X_6 : X_4, X_5, X_3$	$X_{12} : X_{10}, X_9, X_{11}, X_7$	$X_{18} : X_{16}, X_{17}, X_{11}, X_7, X_{15}$

Można dowieść, że algorytm zawsze zostaje przerwany w miejscu n w drugim cyklu, to znaczy dla $n < \mu + 2\lambda$. Więcej szczegółów na ten temat można znaleźć w pozycji [Knu2].

Funkcja linijki pomaga rozwiązać łamigłówkę wież Hanoi. Ponumerujmy n dysków od 0 do $n - 1$. W każdym kroku k, gdzie k przyjmuje wartości od 1 do $2^n - 1$ przenosimy dysk ntz(k) o minimalną dopuszczalną odległość w prawo w sposób cykliczny.

Funkcja linijki może posłużyć również do utworzenia odbiciowego binarnego kodu Graya (zob. podrozdział 13.1 na stronie 263). Rozpoczynamy od dowolnej liczby n-bitowej i w każdym kroku k, gdzie k przyjmuje wartości od 1 do $2^n - 1$ odwracamy bit o indeksie ntz(k).

Rozdział 6.

Przeszukiwanie słów

6.1 Wyszukiwanie pierwszego bajtu o wartości 0

Funkcja tego typu znajduje zastosowanie głównie z powodu sposobu reprezentacji napisów w języku C. Nie jest przechowywana żadna informacja o długości napisu, natomiast zakończenie napisu jest zaznaczone bajtem o wartości 0. W celu określenia długości napisu w języku C wykorzystywana jest funkcja *strlen* (ang. *string length* — długość napisu). Funkcja ta przeszukuje napis od lewej do prawej w poszukiwaniu bajtu o wartości 0 i zwraca liczbę przeszukanych znaków, nie licząc bajtu zerowego.

Szybka implementacja funkcji *strlen* umożliwiłaby ładowanie i sprawdzanie wartości pojedynczych bajtów aż do osiągnięcia granicy słowa a następnie ładować po jednym słowie danych do rejestru i sprawdzać zawartość rejestru pod kątem występowania bajtu o wartości 0. Na maszynach klasy *big-endian* (zapisujących liczby z bardziej znaczącymi bajtami z lewej strony) potrzebna jest funkcja zwracająca indeks pierwszego bajtu zerowego licząc od lewej strony. Wygodnym sposobem kodowania wyniku jest określenie liczby od 0 do 3 lub liczby 4 w przypadku, gdy bajt zerowy nie występuje w badanym słowie. Na maszynach typu *little-endian* (bardziej znaczące bajty znajdują się z prawej strony) jest potrzebny pierwszy bajt zerowy licząc od prawej, ponieważ procesory tego typu odwracają kolejność bajtów przy ładowaniu ich do rejestru. Interesują nas następujące funkcje, gdzie „00" określa bajt zerowy, „nn" określa bajt niezerowy, natomiast „xx" określa bajt o dowolnej wartości.

$$\text{zbytel}(x) = \begin{cases} 0, & x = \text{00xxxxxx}, \\ 1, & x = \text{nn00xxxx}, \\ 2, & x = \text{nnnn00xx}, \\ 3, & x = \text{nnnnnn00}, \\ 4, & x = \text{nnnnnnnn}. \end{cases}$$

$$\text{zbyter}(x) = \begin{cases} 0, & x = \text{xxxxxx00}, \\ 1, & x = \text{xxxx00nn}, \\ 2, & x = \text{xx00nnnn}, \\ 3, & x = \text{00nnnnnn}, \\ 4, & x = \text{nnnnnnnn}. \end{cases}$$

Pierwsza z procedur wyszukujących pierwszy bajt zerowy, przedstawiona na listingu 6.1, po prostu sprawdza wartość każdego z bajtów w słowie w kolejności od lewej do prawej i zwraca wynik po napotkaniu bajtu zerowego.

Listing 6.1. *Wyszukiwanie lewostronnego bajtu zerowego za pomocą sekwencji porównań*

```
int zbytel(unsigned x) {
   if            ((x >> 24) == 0) return 0;
   else if ((x & 0x00FF0000) == 0) return 1;
   else if ((x & 0x0000FF00) == 0) return 2;
   else if ((x & 0x000000FF) == 0) return 3;
   else return 4;
}
```

Procedura ta wymaga wykonania od dwóch do 11 instrukcji RISC: 11 instrukcji wykonuje się w przypadku, gdy słowo nie zawiera bajtu zerowego (jest to bardzo typowa sytuacja w przypadku funkcji *strlen*). W celu wyszukania prawostronnego bajtu (dla architektury *big-endian*) można skonstruować bardzo podobny algorytm.

Listing 6.2 zawiera wersję tej funkcji bez rozgałęzień. Pomysł polega na tym, żeby każdy bajt zerowy zastąpić wartością 0x80, natomiast każdy bajt niezerowy wartością 0x00 a następnie *wyliczyć liczbę zer wiodących*. W przypadku, gdy maszyna obsługuje instrukcję *wyznaczającą liczbę zer wiodących* oraz instrukcję *nor* procedura ta wymaga wykonania ośmiu instrukcji. Podobne sztuczki można znaleźć w pracy [Lamp].

Listing 6.2. *Wyszukiwanie lewostronnego bajtu zerowego, bez rozgałęzień*

```
int zbytel(unsigned x) {
   unsigned y;
   int n;
                        // Oryginalny bajt: 00 80 inne
   y = (x & 0x7F7F7F7F) + 0x7F7F7F7F;     // 7F 7F 1xxxxxxx
   y = ~(y | x | 0x7F7F7F7F);             // 80 00 00000000
   n = nlz(y) >> 3;              // n = 0 ... 4, 4 gdy x
   return n;                     // nie ma bajtu zerowego
}
```

Pozycję prawostronnego zera można określić za pomocą liczby końcowych zer w wyniku końcowym, (wyliczonym powyższym sposobem) przez podzielenie tej liczby przez 8 (z odrzuceniem części ułamkowej). Wykorzystując technikę wyliczania zer końcowych na podstawie liczby zer wiodących (zob. podrozdział 5.4. *Zliczanie zer końcowych* na

stronie 104) można wyliczyć tę wartość, zastępując następującym wyrażeniem przypisanie liczbie n w funkcji na listingu 6.2:

```
n = (32 - nlz(~y & (y - 1))) >> 3;
```

Jest to rozwiązanie wymagające wykonania 12 instrukcji, pod warunkiem, że maszyna obsługuje instrukcje *nor* oraz *and not*.

W większości przypadków na platformie PowerPC procedura wyszukująca prawostronny bajt zerowy nie będzie potrzebna. Zamiast tego wystarczy załadować słowa za pomocą instrukcji *ładowania słowa z odwróceniem bajtów* (ang. *load word byte reversed — lwbrx*).

Procedura zaprezentowana na listingu 6.2 jest bardziej przydatna na maszynach 64-bitowych niż na maszynach 32-bitowych, ponieważ na 64-bitowej maszynie procedura ta (po odpowiednich, raczej oczywistych zmianach) wykorzystuje tę samą liczbę instrukcji (siedem lub dziesięć w zależności od techniki generowania stałej), podczas gdy technika z listingu 6.1 wykorzystuje w najgorszym przypadku 23 instrukcje.

Jeśli potrzebne jest tylko sprawdzenie *występowania bajtu zerowego* (bez konieczności zwracania indeksu bajtu w słowie), po drugim przypisaniu do zmiennej y można wykorzystać instrukcję rozgałęzienia w przypadku *wartości zero* (ang. *branch on zero*).

W przypadku, gdy nie jest dostępna instrukcja *nlz*, nie jest znany żaden dobry sposób *wyznaczenia pierwszego bajtu zerowego*. Listing 6.3 przedstawia możliwe rozwiązanie (prezentujemy tylko fragment wykonawczy tej funkcji).

Listing 6.3. *Wyszukiwanie lewostronnego bajtu zerowego, bez użycia funkcji nlz*

```
                          // Oryginalny bajt: 00 80 inne
    y = (x & 0x7F7F7F7F) + 0x7F7F7F7F; // 7F 7F 1xxxxxxx
    y = ~(y | x | 0x7F7F7F7F);        // 80 00 00000000
                                      // Poszczególne kroki:
    if (y == 0) return 4;             // 00000000 ==> 4,
    else if (y > 0x0000FFFF)          // 80xxxxxx ==> 0,
       return (y >> 31) ^ 1;          // 0080xxxx ==> 1,
    else                              // 000080xx ==> 2,
       return (y >> 15) ^ 3;          // 00000080 ==> 3.
}
```

Funkcja ta wymaga wykonania 13 instrukcji RISC lub 10 w przypadku, gdy zero nie występuje. Jest to zatem nieco gorszy kod od przedstawionego na listingu 6.1, jednak wykorzystuje mniejszą liczbę instrukcji rozgałęzień. Niestety algorytm ten nie da się łatwo przenieść na architekturę 64-bitową.

Istnieją również inne sposoby uniknięcia konieczności stosowania funkcji *nlz*. Wartość zmiennej y wyliczonej za pomocą kodu z listingu 6.3 składa się z czterech bajtów, każdy z nich ma wartość 0x00 lub 0x80. Resztą z dzielenia takiej liczby przez liczbę 0x7F jest oryginalna wartość z liczbą jedynek równą maksymalnie 4. Jedynki te są przesunięte i skompresowane do skrajnie prawych pozycji. Stąd reszta ta przyjmuje wartości od 0 do 15 i jednoznacznie identyfikuje oryginalną liczbę. Na przykład:

remu(**0x80808080**, **127**) = **15**,

remu(**0x80000000**, **127**) = **8**,

remu(**0x00008080**, **127**) = **3**, itd.

Wartość tę można wykorzystać w charakterze indeksu w tabeli o rozmiarze 16 bajtów. Dzięki temu kod od miejsca `if (y == 0)` można zastąpić następującym:

```
static char table[16] = {4, 3, 2, 2, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0};
return table[y % 127];
```

W powyższym fragmencie `y` jest liczbą całkowitą bez znaku. W miejsce liczby 127 można wykorzystać liczbę 31, lecz w tym celu należałoby zmodyfikować tabelę.

Metody wykorzystujące dzielenie przez 127 czy 31 zostały przedstawione w charakterze ciekawostki, ponieważ *obliczenie reszty z dzielenia* może zająć do 20 cykli zegara, nawet w przypadku, gdyby była zaimplementowana bezpośrednio w procesorze. Poniżej prezentujemy jednak dwa nieco wydajniejsze zamienniki kodu od miejsca `if (y == 0)` w listingu 6.3:

```
return table[hopu(y, 0x02040810) & 15];
```

oraz:

```
return table[y * 0x00204081 >> 28];
```

W powyższym kodzie funkcja `hopu(a, b)` określa starsze 32 bity iloczynu bez znaku liczb `a` i `b`. W drugiej wersji zakładamy, że stosowana jest zwyczajowa konwencja HLL, polegająca na tym, że w 32-bitowym wyniku mnożenia liczb zostaną zwrócone mniej znaczące 32 bity pełnego wyniku. Może okazać się, że druga z metod jest bardziej praktyczna, na przykład w przypadku, gdy maszyna udostępnia szybkie instrukcje *mnożenia* lub w przypadku, gdy mnożenie przez 0x204081 jest wykonywane za pomocą sekwencji *przesunięć* i *dodawań*. Mnożenie to można wykonać za pomocą czterech instrukcji:

$$y(1 + 2^7 + 2^{14} + 2^{21}) = y(1 + 2^7)(1 + 2^{14})$$

Wykorzystując powyższe mnożenie w czterech cyklach cały czas wykonania procedury obejmuje 13 cykli (7 w celu obliczenia zmiennej `y`, plus cztery na *przesunięcia* i *dodawania* plus 2 na *przesunięcie w prawo* o 28 i odwołanie do indeksu w tabeli). Oczywiście kod ten nie wykorzystuje rozgałęzień.

Metoda ta daje się łatwo zmodyfikować dla maszyn 64-bitowych. W charakterze metody *modulus* należy zastosować następującą operację:

```
return table[y % 511];
```

Tabela w tym przypadku będzie miała rozmiar 256 bajtów i wartości 8, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 4,... (tzn. `table[i]` = liczba zer wiodących w *i*).

Metody polegające na mnożeniu wyglądają następująco:

```
return table[hopu(y, 0x0204081020408100) & 255];
```

oraz:

```
return table[(y * 0x0002040810204081 >> 56];
```

W tym przypadku tabela będzie miała rozmiar 256 bajtów i wartości 8, 7, 6, 6, 5, 5, 5, 5, 4, 4, 4, 4, 4, 4, 4, 3,....

Mnożenie przez 0x2040810204081 można zastąpić zestawem operacji:

$$t_1 \leftarrow y(1 + 2^7)$$

$$t_2 \leftarrow t_1(1 + 2^{14})$$

$$t_3 \leftarrow t_2(1 + 2^{28})$$

Powyższe rozwiązanie wymaga 13 cykli procesora.

Wszystkie powyższe algorytmy wykorzystujące tablice mogą oczywiście zostać zaimplementowane za pomocą operacji *wyszukiwania prawostronnego bajtu zerowego*. W tym celu należy odpowiednio zmodyfikować wartości w tabelach.

Jeśli procesor nie obsługuje instrukcji `nor`, w przypadku maszyny 32-bitowej można pominąć operację *negacji* w drugim przypisaniu zmiennej `y` na listingu 6.3. Wystarczy zastosować jedno z prezentowanych wyrażeń `return table[...]`, dla `table[i]` = 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 2, 2, 3, 4. Metoda ta nie zadziała na maszynach 64-bitowych.

Istnieje ciekawa odmiana procedury z listingu 6.2, również przeznaczona dla maszyn nieobsługujących instrukcji wyznaczającej *liczbę zer wiodących*. Załóżmy, że a, b, c i d są 1-bitowymi zmiennymi predykatów „pierwszy bajt x jest równy zero", „drugi bajt x jest równy zero" itd. W tym przypadku:

$$\text{zbytel}(x) = a + ab + abc + abcd.$$

Mnożenia można zastąpić operacjami *koniunkcji*, co prowadzi do kodu przedstawionego na listingu 6.4 (przedstawiamy jedynie kod wykonawczy):

Listing 6.4. *Wyszukiwanie lewostronnego bajtu zerowego za pomocą wyliczenia wielomianu*

```
y = (x & 0x7F7F7F7F) + 0x7F7F7F7F;
y = y | x;            // wiodąca 1 w niezerowych bajtach

t1 =  y >> 31;                // t1 = a.
t2 = (y >> 23) & t1;          // t2 = ab.
t3 = (y >> 15) & t2;          // t3 = abc.
t4 = (y >>  7) & t3;          // t4 = abcd.
return t1 + t2 + t3 + t4;
```

Algorytm ten wymaga zastosowania 15 instrukcji z podstawowego zestawu RISC, co nie jest bardzo szybkim rozwiązaniem, lecz kod ten daje możliwość równoległego wykonywania. Na maszynie superskalarnej, która jest w stanie wykonać jednocześnie do trzech niezależnych instrukcji, kod ten wykona się w zaledwie dziesięciu cyklach.

Prosta odmiana tego algorytmu pozwala na skonstruowanie algorytmu wyszukiwania *prawostronnego bajtu zerowego* za pomocą operacji:

$$\text{zbyter}(x) = \boldsymbol{abcd} + \boldsymbol{bcd} + \boldsymbol{cd} + \boldsymbol{d}$$

Operacja ta wykorzystuje o jedną operację *koniunkcji* więcej od kodu z listingu 6.4.

Proste uogólnienia

Funkcje „zbytel" oraz „zbyter" można wykorzystać w celu wyszukiwania bajtu o określonej wartości, w pierwszej kolejności wykonując operację *różnicy symetrycznej* na liczbie x oraz słowie złożonym z powtórzonej odpowiednią liczbę razy poszukiwanej wartości. Na przykład w celu odszukania znaku spacji (0x20) w słowie x należy wyszukać bajt zerowy w wyniku działania $x \oplus$ **0x20202020**.

Podobnie w celu odszukania bajtu, w którym słowa x oraz y mają identyczną wartość, należy wyszukać bajt zerowy w wyniku działania $x \oplus y$.

Nie ma nic nadzwyczajnego w ograniczeniach zastosowanych w listingu 6.2 oraz jego odmianach. Na przykład w celu wyszukania wartości zerowej w pierwszych czterech bitach słowa, kolejnych dwunastu oraz ostatnich szesnastu należy zastosować kod z listingu 6.2 z maską o wartości 0x77FF7FFF [PHO]. W przypadku, gdy poszukiwana długość pola jest równa 1, na odpowiedniej pozycji maski należy zastosować wartość 0.

Wyszukiwanie wartości w określonym przedziale

Kod z listingu 6.2 można z łatwością zmodyfikować w sposób pozwalający na wyszukiwanie bajtu o określonej wartości w przedziale od 0 do 128. Poniższy kod wyszukuje indeks bajtu (od lewej) o wartości od 0 do 9:

```
y = (x & 0x7F7F7F7F) + 0x76767676;
y = y | x;
y = y | 0x7F7F7F7F;          // Bajty > 9 to 0xFF.
y = ~y;                      // Bajty > 9 to 0x00.
                             // Bajty <= 9 to 0x80.
n = nlz(y) >> 3;
```

Ogólnie: załóżmy, że chcemy wyszukać pierwszy bajt od lewej, którego wartość jest z zakresu od a do b, gdzie różnica pomiędzy a i b jest mniejsza od 128. Jako przykład przyjmijmy wielkie litery o kodach ASCII od 0x41 do 0x5A. Aby wyszukać pierwszą wielką literę w słowie odejmujemy od wartości słowa liczbę 0x41414141 w taki sposób, aby pożyczka nie przeszła poza granice bajtów. Następnie wykorzystujemy powyższy

kod, aby odnaleźć bajty o wartości z przedziału od 0 do 0x19 (0x5A – 0x41). Wykorzystując formuły odejmowania, które zostały przedstawione w podrozdziale 2.17 *Operacje dodawania, odejmowania i wyznaczania wartości bezwzględnej na wartościach wielobajtowych* na stronie 53, z oczywistym uproszczeniem dla przypadku y = 0x41414141, otrzymujemy:

```
d = (x | 0x80808080) - 0x41414141;
d = ~((x | 0x7F7F7F7F) ^ d);
y = (d & 0x7F7F7F7F) + 0x66666666;
y = y | d;
y = y | 0x7F7F7F7F;     // Bajty z poza przedziału 41-5A mają wartość FF.
y = ~y;                 // Bajty z poza przedziału 41-5A mają wartość 00,
                        // bajty z przedziału 41-5A mają wartość 80.
n = nlz(y) >> 3;
```

W przypadku niektórych zakresów wartości można znaleźć również prostszy kod. Na przykład załóżmy, że chcemy wyszukać pierwszy bajt, którego wartość wynosi od 0x30 do 0x39 (cyfra dziesiętna zakodowana w ASCII). Wystarczy obliczyć różnicę symetryczną słowa wejściowego i wartości 0x30303030 a następnie wykorzystać powyższy kod w celu wyszukania bajtów o wartościach od 0 do 9. Uproszczenie to jest możliwe do zastosowania w przypadku, gdy górne i dolne ograniczenie posiada wspólne n bardziej znaczących bitów oraz gdy ograniczenie dolne kończy się liczbą zer równą $8 - n$.

Opisane techniki można dostosować do obsługi zakresów powyżej 128 bez konieczności dodawania instrukcji. Na przykład w celu wyszukania lewostronnego indeksu bajtu o wartości od 0 do 137 (0x89) wystarczy w kodzie wyszukującym wartości od 0 do 9 zastąpić wiersz kodu o treści `y = y | x` na następujący: `y = y & x`.

Podobnie zastępując wiersz `y = y | d` na `y | y & d` w kodzie wyszukującym wartości z zakresu od 0x41 do 0x5A utworzymy kod wyszukujący lewostronny indeks bajtu o wartości z przedziału od 0x41 do 0xDA.

6.2. Wyszukiwanie pierwszego ciągu jedynek o zadanej długości

Kolejny problem polega na wyszukaniu ciągu jedynek o określonej długości (lub dłuższego) w słowie w rejestrze. W przypadku, gdy ciąg zostanie odnaleziony, należy zwrócić jego pozycję, jeśli nie, należy odpowiednio zasygnalizować ten fakt. Istnieją również odmiany tego algorytmu, zwracające jedynie informację o tym, czy taki ciąg istnieje (o znaczeniu *tak* lub *nie*), jak również zwracające pierwsze wystąpienie liczby jedynek dokładnie o zadanej długości. Problem tego typu jest wykorzystywany w programach realizujących alokację miejsca na dysku, szczególnie w celu defragmentacji dysku (to znaczy takiej modyfikacji położenia danych na dysku, aby każdy plik zajmował ciągły obszar na dysku). Problem ten został zasygnalizowany Autorowi przez Alberta Changa, który zasugerował, iż jest to jedno z możliwych zastosowań funkcji *zliczającej liczbę zer wiodących*.

Załóżmy, że w komputerze jest dostępna instrukcja *zliczająca liczbę zer wiodących* w słowie, lub jej bezpośredni odpowiednik w postaci procedury.

Algorytmem, który przychodzi do głowy w pierwszej chwili jest obliczenie liczby zer wiodących i przesunięcie wartości w lewo o odpowiednią liczbę miejsc. Następnie zliczamy liczbę jedynek za pomocą *dopełnienia* (ang. *complement*) wartości słowa i ponownego zliczenia zer wiodących. Jeśli wynik spełnia założenia, mamy gotowy wynik. W przeciwnym wypadku przesuwamy wynik w lewo o odpowiednią liczbę miejsc i ponownie zaczynamy od początku (dopełnienie wartości i zliczenie liczby zer wiodących). Algorytm ten przedstawiamy poniżej. W przypadku, gdy zostanie odnalezionych n następujących po sobie jedynek, algorytm ten zwraca liczbę od 0 do 31, określającą indeks, od którego w słowie wejściowym rozpoczyna się ten ciąg (licząc od lewej strony). W przeciwnym wypadku zostanie zwrócona liczba 32, oznaczająca, że ciąg jedynek o zadanej długości nie został odnaleziony.

```
int ffstrl(unsigned x, int n) {
   int k, p;

   p = 0;                  // Inicjalizacja wartości zwracanej pozycji.
   while (x != 0) {
      k = nlz(x);          // Pominięcie początkowych zer
      x = x << k;          // (o ile istnieją).
      p = p + k;
      k = nlz(~x);         // Zliczenie pierwszej (i kolejnych) grupy jedynek.
      if (k >= n)          // Jeśli wartość dostatecznie duża,
         return p;         // zwrócenie jej.
      x = x << k;          // Jeśli za mało jedynek,
      p = p + k;           // pominięcie ich.
   }
   return 32;
}
```

Algorytm ten jest rozsądny w przypadku, gdy oczekujemy, że nie będzie on wykonywany wielokrotnie, to znaczy wtedy, gdy w słowie x występują długie sekwencje samych zer i jedynek. Założenie takie może być sensowne na przykład w przypadku aplikacji zajmującej się alokacją miejsca na dysku. W najgorszym przypadku nie jest to jednak najlepszy algorytm, na przykład dla liczby x = 0x55555555 i $n \geq 2$ wymaga on 178 instrukcji z pełnego zestawu RISC.

Algorytmem, który daje lepsze wyniki w gorszych przypadkach jest algorytm oparty na instrukcjach *przesunięcia w lewo* oraz *koniunkcji*. Aby sprawdzić jego działanie przyjmijmy, że poszukujemy ciągu ośmiu lub więcej jedynek w 32-bitowym słowie x. Można tego dokonać w następujący sposób:

$$x \leftarrow x \,\&\, (x << 1)$$

$$x \leftarrow x \,\&\, (x << 2)$$

$$x \leftarrow x \,\&\, (x << 4)$$

Po pierwszym przypisaniu jedynki w wyniku określają pozycje ciągów o długości 2. Po drugim przypisaniu jedynki w x określają pozycje ciągów o długości 4 (to znaczy ciągu jedynek o długości 2, po którym natychmiast następuje kolejny ciąg jedynek o długości 2). Po trzecim przypisaniu jedynki w x określają miejsca, od których rozpoczynają się ciągi jedynek o długości 8. Po wyliczeniu *liczby zer wiodących* w tym wyniku otrzymamy liczbę oznaczającą indeks pierwszego ciągu jedynek o długości 8 (lub większej) lub liczbę 32 w przypadku, gdy ciąg taki nie wystąpił.

W celu utworzenia algorytmu działającego dla każdej długości od 1 do 32 warto na problem spojrzeć z nieco innej strony. Najpierw należy zaobserwować, że trzy powyższe przypisania mogą wystąpić w dowolnej kolejności. Dla nas bardziej wygodna będzie kolejność odwrotna. Aby zrozumieć ogólną metodę, przyjrzyjmy się algorytmowi dla $n = 10$:

$$x_1 \leftarrow x \,\&\, (x << 5)$$

$$x_2 \leftarrow x_1 \,\&\, (x_1 << 2)$$

$$x_3 \leftarrow x_2 \,\&\, (x_2 << 1)$$

$$x_4 \leftarrow x_3 \,\&\, (x_3 << 1)$$

Pierwsza instrukcja przesuwa x w lewo o wartość $n/2$. Po tej instrukcji nasz problem zostaje zredukowany do wyszukania ciągu pięciu jedynek w x_1. Możemy tego dokonać przesuwając x_1 w lewo o $\lfloor 5/2 \rfloor = 2$ miejsc i wyliczeniu koniunkcji z oryginalną wartością x_1. W wyniku tej operacji musimy wyszukać ciąg jedynek o długości 3 (5 – 2). Kolejne dwa wyrażenia wyszukują ciąg jedynek o długości 3 w x_2. Suma przesunięć jest zawsze równa $n - 1$. Algorytm przedstawiamy na listingu 6.5. Czas jego wykonania wynosi od 3 do 36 instrukcji z pełnego zestawu instrukcji RISC dla n od 1 do 32.

Listing 6.5. *Wyszukiwanie pierwszego ciągu n jedynek za pomocą sekwencji operacji przesunięć i koniunkcji*

```
int ffstrl(unsigned x, int n) {
   int s;

   while (n > 1) {
      s = n >> 1;
      x = x & (x << s);
      n = n - s;
   }
   return nlz(x);
}
```

W przypadku, gdy `n` jest dość duże, całkiem rozsądne wydaje się rozwinięcie tej pętli pięciokrotnie i pominięcie testu `n > 1`. Pięciokrotne wykonanie pętli zawsze wystarczy dla 32-bitowego słowa. Dzięki temu uzyskujemy algorytm niewykorzystujący rozgałęzień i wykonujący się w stałej liczbie 50 instrukcji (można pominąć ostatnie przypisanie zmiennej x). W przypadku jednak niewielkich wartości zmiennej x ostatnie trzy przebiegi wykonują się bez potrzeby a wynik nie pozostaje zmieniony przez dodatkowe kroki, ponieważ zmienna n pozostaje przy wartości 1. Dla wartości $n = 1$ dodatkowe

kroki nie mają wpływu na wartość x ani n. Wersja rozwinięta jest szybsza (pod względem liczby wykonywanych instrukcji) od wersji wykorzystującej rozgałęzienia już dla $n \geq 5$.

Ciąg dokładnie n jedynek można wyszukać za pomocą sześciu dodatkowych instrukcji (w przypadku, gdy dostępna jest instrukcja *and not* wystarczą cztery). Wartość x wyliczona w algorytmie na listingu 6.5 posiada bit o wartości 1 na każdej pozycji, od której w oryginalnym słowie rozpoczyna się ciąg n lub więcej jedynek. Stąd, w ostatecznej wartości x wyliczonej w tym algorytmie indeks ciągu składającego się z dokładnie n jedynek będzie odosobnioną jedynką (otoczoną zerami). Można to wyznaczyć za pomocą następującej formuły:

$$x \,\&\, \neg(x \overset{u}{>>} 1) \,\&\, \neg(x << 1)$$

Algorytm ten można również zmodyfikować w taki sposób, aby znajdował ciągi bitów o określonej długości n występujące w określonym miejscu. Na przykład w celu określenia, czy ciąg bitów występuje na granicach bajtu wystarczy wynik ostatniego kroku w powyższym algorytmie poddać operacji *koniunkcji* z liczbą 0x80808080.

Aby wyszukać ciągi zer, można na przykład dopełnić wartość x na początku lub operacje *koniunkcji* zastąpić operacjami *alternatywy* a wartość x dopełnić przed samym wyliczeniem „nlz". Na przykład poniżej prezentujemy algorytm wyszukujący pierwszy (lewostronny) bajt zerowy (szerszy opis tego problemu można znaleźć w podrozdziale 6.1. *Wyszukiwanie pierwszego bajtu o wartości O* na stronie 91).

$$x \leftarrow x \mid (x << 4)$$

$$x \leftarrow x \mid (x << 2)$$

$$x \leftarrow x \mid (x << 1)$$

$$x < \mathbf{0x7F7F7F7F} \mid x$$

$$p \leftarrow \mathrm{nlz}(\neg x) \overset{u}{>>} 3$$

Powyższy algorytm wykonuje się w 12 instrukcjach z pełnego zestawu RISC. Jak widać nie jest to algorytm tak dobry jak algorytm z listingu 6.2, na stronie 112, który wykonuje się w ośmiu instrukcjach.

Rozdział 7.

Manipulacja bitami i bajtami

7.1. Odwracanie kolejności bitów i bajtów

Przez operacje *odwrócenia kolejności bitów* (ang. *reversing bits*) rozumiemy wykonanie „odbicia lustrzanego", na przykład:

rev(**0x01234567**) = **0xE6A2C480**

Przez operację odwrócenia bajtów rozumiemy podobne odbicie, z tym, że w tym przypadku odwracamy kolejność bajtów w słowie. Odwracanie kolejności bajtów jest operacją konieczną w przypadku konwersji pomiędzy formatami *little-endian*, stosowanymi w procesorach DEC oraz Intel a formatem *big-endian*, stosowanym przez większość pozostałych producentów procesorów.

Odwrócenie kolejności bitów może zostać wykonane w stosunkowo wydajny sposób przez zamianę kolejności sąsiadujących bitów, następnie zamianę kolejności w sąsiadujących parach pól 2-bitowych itd. [Aus1]. Poniżej przedstawiamy przykład realizacji tego mechanizmu. Wszystkie operacje przypisania można wykonać w dowolnej kolejności.

```
x = (x & 0x55555555) << 1 | (x & 0xAAAAAAAA) >> 1;
x = (x & 0x33333333) << 2 | (x & 0xCCCCCCCC) >> 2;
x = (x & 0x0F0F0F0F) << 4 | (x & 0xF0F0F0F0) >> 4;
x = (x & 0x00FF00FF) << 8 | (x & 0xFF00FF00) >> 8;
x = (x & 0x0000FFFF) << 16 | (x & 0xFFFF0000) >> 16;
```

Na większości maszyn jest możliwe dokonanie niewielkiego usprawnienia, polegającego na wykorzystaniu mniejszej liczby różnych stałych oraz na wykonaniu ostatnich dwóch przypisań w bardziej bezpośredni sposób, tak jak zostało przedstawione na listingu 7.1 (30 instrukcji z podstawowego zestawu RISC, bez rozgałęzień).

Listing 7.1. *Odwracanie kolejności bitów*

```
unsigned rev(unsigned x) {
   x = (x & 0x55555555) <<  1 | (x >>  1) & 0x55555555;
   x = (x & 0x33333333) <<  2 | (x >>  2) & 0x33333333;
   x = (x & 0x0F0F0F0F) <<  4 | (x >>  4) & 0x0F0F0F0F;
   x = (x << 24) | ((x & 0xFF00) << 8) |
       ((x >> 8) & 0xFF00) | (x >> 24);
   return x;
}
```

Ostatnie przypisanie zmiennej x w tym kodzie realizuje odwrócenie bajtów w dziewięciu instrukcjach zestawu podstawowego RISC. W przypadku, gdy maszyna udostępnia instrukcje *przesunięć cyklicznych*, można tego dokonać w siedmiu instrukcjach, wykorzystując następującą formułę:

$$x = ((x \,\&\, \mathbf{0x00FF00FF}) \overset{rot}{>>} 8) \mid ((x \overset{rot}{<<} 8) \,\&\, \mathbf{0x00FF00FF}).$$

Na procesorach PowerPC operację odwrócenia bajtów można wykonać w zaledwie trzech instrukcjach [Hay1]: *przesunięcie cykliczne w lewo* o 8, które umieszcza dwa bajty na odpowiednich miejscach, po którym wykonuje się dwa wywołania instrukcji `rlwimi` (ang. *rotate left word immediate then mask insert — przesunięcie cykliczne w lewo a następnie podstawienie wartości z zastosowaniem maski*).

Uogólnienie operacji odwrócenia bitów

W publikacji [GLS1] zasugerowano następujący sposób uogólnienia operacji odwrócenia bitów. Sposób ten nazwano `flip`. Bardzo dobrze nadaje się on do zastosowania jako nowa instrukcja w zestawie instrukcji procesora:

```
if (k & 1) x = (x & 0x55555555) << 1 | (x & 0xAAAAAAAA) >>  1;
if (k & 2) x = (x & 0x33333333) << 2 | (x & 0xCCCCCCCC) >>  2;
if (k & 4) x = (x & 0x0F0F0F0F) << 4 | (x & 0xF0F0F0F0) >>  4;
if (k & 8) x = (x & 0x00FF00FF) << 8 | (x & 0xFF00FF00) >>  8;
if (k & 16) x = (x & 0x0000FFFF) << 16 | (x & 0xFFFF0000) >> 16;
```

Można pominąć ostatnie dwa wywołania instrukcji *and*. Dla $k = 31$ operacja ta dokonuje odwrócenia bitów w słowie. Dla $k = 24$ odwraca bajty w słowie. Dla $k = 7$ odwraca bity w każdym bajcie bez dokonywania zmian położenia bajtów. Dla $k = 16$ dokonuje zamiany półsłów w słowie itd. Ogólnie mówiąc, operacja ta przenosi bit z pozycji m na pozycję $m \oplus k$. Instrukcja ta może zostać zaimplementowana w sprzęcie w podobny sposób, w jaki z reguły są implementowane instrukcje *przesunięcia cyklicznego* (pięć segmentów MUX, z których każdy jest kontrolowany za pomocą jednego bitu rozmiaru przesunięcia k).

Wariacje na temat odwracania bitów

Pozycja 167 w [HAK] przedstawia dość nietypowe wyrażenia wykonujące odwracanie 6, 7 i 8-bitowych liczb całkowitych. Wyrażenia te są zaprojektowane dla 36-bitowych

maszyn, lecz wersja dla wartości 6-bitowych działa również na maszynie 32-bitowej, natomiast wersje dla liczb 7 i 8-bitowych działają na maszynach 64-bitowych.

6 bitów: $\text{remu}((x * \mathbf{0x00082082}) \,\&\, \mathbf{0x01122408}, \mathbf{255})$

7 bitów: $\text{remu}((x * \mathbf{0x40100401}) \,\&\, \mathbf{0x442211008}, \mathbf{255})$

8 bitów: $\text{remu}((x * \mathbf{0x202020202}) \,\&\, \mathbf{0x10884422010}, \mathbf{1023})$

W wyniku powyższych wyrażeń powstaje „czysta" liczba całkowita — wyrównana do prawej, bez zostawiania żadnego niepotrzebnego bardziej znaczącego bitu.

W każdej z tych sytuacji funkcja `remu` może zostać zastąpiona funkcją `rem` lub `mod`, ponieważ jej argumenty są liczbami dodatnimi. Funkcja `rem` (ang. *reminder* — reszta z dzielenia) po prostu sumuje cyfry liczby o podstawie 256 lub 1024, co działa zupełnie tak samo, jak w przypadku metody odrzucania dziewiątek (ang. *casting out nines*). Dzięki temu można ją zastąpić za pomocą kombinacji *mnożenia* i *przesunięć w prawo*. Na przykład 6-bitowa formuła posiada na maszynie 32-bitowej następującą postać alternatywną (mnożenie musi być wykonywane modulo 2^{32}):

$$t \leftarrow (x * \mathbf{0x00082082}) \,\&\, \mathbf{0x01122408}$$

$$(t \overset{u}{*} \mathbf{0x01010101}) >> 24$$

Przedstawione formuły mają dość ograniczone zastosowanie z powodu wykorzystania operacji *reszty z dzielenia* (20 cykli lub więcej) oraz kilku dodatkowych mnożeń i operacji ładowania dużych wartości bezpośrednich. Ostatnia z powyższych formuł wykorzystuje dziesięć instrukcji z podstawowego zestawu RISC, z których dwie są instrukcjami *mnożenia*, co na współczesnych procesorach klasy RISC daje w sumie około 20 cykli. Z drugiej strony wykorzystanie kodu z listingu 7.1 w celu odwracania liczb 6-bitowych wymaga zastosowania około 15 instrukcji oraz około 9 do 15 cykli, w zależności od możliwości procesora w zakresie równoległego wykonania niezależnych instrukcji. Techniki te można jednak zaimplementować za pomocą prostego i zwartego kodu. Poniżej prezentujemy techniki, które mogą być przydatne w maszynach 32-bitowych. Wykorzystują one coś na kształt podwójnego zastosowania pomysłu z [HAK], co posłużyło do rozwinięcia tej techniki do liczb 8 i 9-bitowych na maszynach 32-bitowych.

Poniższa formuła służy do odwracania bitów w 8-bitowej liczbie całkowitej:

$$s \leftarrow (x * \mathbf{0x02020202}) \,\&\, \mathbf{0x84422010}$$

$$t \leftarrow (x * \mathbf{8}) \,\&\, \mathbf{0x00000420}$$

$$\text{remu}(s + t, \mathbf{1023})$$

W tym przypadku funkcji `remu` nie można zastąpić kombinacją *mnożenia* i *przesunięcia*. Wyjaśnienie przyczyny pozostawiamy Czytelnikowi. Dla ułatwienia proponujemy przyjrzeć się układowi bitów w stałych, wykorzystywanych przez formułę.

Oto podobna formuła służąca do odwracania bitów w 8-bitowych liczbach całkowitych. Jest ona ciekawa dlatego, że możemy ją dodatkowo nieco uprościć:

$$s \leftarrow (x * \mathbf{0x00020202}) \,\&\, \mathbf{0x01044010}$$

$$t \leftarrow (x * \mathbf{0x00080808}) \,\&\, \mathbf{0x02088020}$$

$$\mathrm{remu}(s + t, \mathbf{4095})$$

Uproszczenia polegają na tym, że drugi iloczyn jest po prostu *przesunięciem w lewo* pierwszego iloczynu, natomiast ostatnią z zastosowanych stałych można wyliczyć z drugiej za pomocą pojedynczej instrukcji *przesunięcia* a operację *wyliczania reszty z dzielenia* można w tym przypadku zastąpić kombinacją *mnożenia* i *przesunięcia*. Dzięki temu formułą ta upraszcza się do 14 instrukcji z podstawowego zestawu instrukcji RISC, z których dwie to instrukcje *mnożenia*:

$$u \leftarrow x * \mathbf{0x00020202}$$

$$m \leftarrow \mathbf{0x01044010}$$

$$s \leftarrow u \,\&\, m$$

$$t \leftarrow (u << 2) \,\&\, (m << 1)$$

$$(\mathbf{0x01001001} * (s + t)) \overset{u}{>>} 24$$

Formuła służąca do odwracania bitów w liczbach 9-bitowych jest następująca:

$$s \leftarrow (x * \mathbf{0x01001001}) \,\&\, \mathbf{0x84108010}$$

$$t \leftarrow (x * \mathbf{0x00040040}) \,\&\, \mathbf{0x00841080}$$

$$\mathrm{remu}(s + t, \mathbf{1023})$$

Drugie z mnożeń można wyeliminować, ponieważ jego iloczyn jest równy pierwszemu iloczynowi przesuniętemu o sześć miejsc w prawo. Ostatnia stała jest równa drugiej przesuniętej w prawo o osiem pozycji. Wykorzystując te uproszczenia można tę formułę sprowadzić do postaci wykorzystującej 12 instrukcji z podstawowego zestawu RISC, w tym dwa *mnożenia* i jedną *resztę z dzielenia*. Operacja *wyznaczania reszty* musi być bez znaku i nie można jej zastąpić kombinacją *mnożenia* i *przesunięcia*.

Czytelnik może samodzielnie skonstruować podobny kod dla innych operacji manipulujących bitami. W charakterze prostego (i sztucznego) przykładu posłużymy się operacją wydobycia co drugiego bitu z 8-bitowej wartości a następnie kompresji czterech wydobytych bitów z wyrównaniem do prawej. To znaczy mamy zamiar wykonać następującą operację:

```
0000 0000 0000 0000 0000 0000 abcd efgh ==>
0000 0000 0000 0000 0000 0000 0000 bdfh
```

Operację tę można zaimplementować w następujący sposób:

$$t \leftarrow (x * \mathbf{0x01010101}) \,\&\, \mathbf{0x40100401}$$

$$(t * \mathbf{0x08040201}) \overset{u}{>>} \mathbf{27}$$

Na większości maszyn najpraktyczniejszy sposób realizacji tego zadania polega na utworzeniu tablicy przeglądowej wszystkich wartości (na przykład jednobajtowych lub 9-bitowych).

Zwiększanie wartości odwróconej liczby całkowitej

Algorytm *szybkiego przekształcenia Fouriera* (ang. *Fast Fourier Transform — FFT*) wymaga zastosowania liczby całkowitej ***i*** oraz jej odwróconej wersji rev(***i***) w pętli, w której wartość ***i*** jest za każdym razem zwiększana o 1 [PB]. W najprostszym rozwiązaniu w każdej iteracji wyliczalibyśmy nową wartość liczby ***i*** a następnie wyliczalibyśmy rev(***i***). Dla niewielkich liczb całkowitych zastosowanie tablicy przeglądowej jest proste i praktyczne. Dla dużych wartości ***i*** technika taka nie jest praktyczna, a jak już wiemy wyliczenie rev(***i***) wymaga 29 instrukcji.

Jeśli jest wykluczone zastosowanie tablicy przeglądowej, bardziej wydajne byłoby osobne przechowywanie wartości ***i*** oraz rev(***i***), w każdej iteracji zwiększając obydwie te wartości. W tym momencie pojawia się problem zwiększenia liczby, której wartość posiadamy w odwróconej formie. W celu ilustracji prezentujemy zastosowanie tej techniki na maszynie 4-bitowej w postaci kolejnych wartości w notacji szesnastkowej:

```
0, 8, 4, C, 2, A, 6, E, 1, 9, 5, D, 3, B, 7, F.
```

W algorytmie FFT zarówno liczba, i jak jej odwrócona wersja stanowią określoną liczbę bitów o określonej długości, która nigdy nie przekracza 32 i obydwie są wyrównane do prawej w ramach rejestru. Załóżmy, że ***i*** jest 32-bitową liczbą całkowitą. Po zwiększeniu o 1 jej odwróconej wartości *przesunięcie w prawo* o odpowiednią liczbę bitów spowoduje, że liczba wynikowa stanie się właściwą wartością w algorytmie FFT (zarówno ***i***, jak i rev(***i***) są wykorzystywane jako indeksy w tablicy).

Najprostszym sposób zwiększenia wartości odwróconej liczby całkowitej jest wyszukanie lewostronnego zera, zmiana jego wartości na 1 a następnie ustawienie wszystkich bitów na lewo od tego miejsca na 0. Jeden ze sposobów realizacji tego algorytmu prezentuje następujący listing:

```
unsigned x, m;

m = 0x80000000;
x = x ^ m;
if ((int)x >= 0) {
   do {
      m = m >> 1;
      x = x ^ m;
   } while (x < m);
}
```

W przypadku, gdy pierwszy bit od lewej ma wartość 0, powyższy kod wykonuje się w trzech instrukcjach z podstawowego zestawu RISC i w każdej iteracji są wykorzystywane cztery kolejne instrukcje. Wartość `x` rozpoczyna się bitem 0 w połowie przypadków, sekwencją 10 w jeden czwartej przypadków i tak dalej, zatem średnia liczba instrukcji wykorzystywanych przez ten algorytm wynosi w przybliżeniu:

$$
\begin{aligned}
& 3\cdot\frac{1}{2}+7\cdot\frac{1}{4}+11\cdot\frac{1}{8}+15\cdot\frac{1}{16}+\ldots \\
&= 4\cdot\frac{1}{2}+8\cdot\frac{1}{4}+12\cdot\frac{1}{8}+16\cdot\frac{1}{16}+\ldots-1 \\
&= 4\left(\frac{1}{2}+\frac{2}{4}+\frac{3}{8}+\frac{4}{16}+\ldots\right)-1 \\
&= 7.
\end{aligned}
$$

W drugim wierszu powyższych obliczeń dodaliśmy i odjęliśmy 1, pierwsza z tych jedynek została rozpisana jako 1/2 + 1/4 + 1/8 + 1/16 + ... Pod tym względem obliczenia te są podobne do przedstawionych na stronie 107). W najgorszym przypadku powyższy algorytm wymaga wykonania dość dużej liczby instrukcji, bo aż 131.

W przypadku, gdy dostępna jest instrukcja *wyliczająca liczbę zer wiodących*, zwiększenie o 1 wartości odwróconej liczby całkowitej można zrealizować w następujący sposób:

$$
\begin{aligned}
\text{Najpierw wyliczamy} \quad & s \leftarrow \mathrm{nlz}(\neg x) \\
\text{następnie:} \quad & x \leftarrow x \oplus (\mathbf{0x80000000} \overset{s}{>>} s) \\
\text{lub:} \quad & x \leftarrow ((x << s) + \mathbf{0x80000000}) \overset{u}{>>} s
\end{aligned}
$$

Każdy ze sposobów wykorzystuje pięć instrukcji z pełnego zestawu RISC a dodatkowo wymagane jest, aby przesunięcia były wykonywane modulo 64, w celu obsłużenia przypadku, gdy następuje zawinięcie wartości **0xFFFFFFFF** do **0** (dlatego powyższe formuły nie będą działać na maszynach z rodziny Intel x86, ponieważ w tych procesorach przesunięcia są wykonywane modulo 32).

7.2. Tasowanie bitów

Kolejnym ważnym sposobem manipulowania bitami jest operacja „tasowania zupełnego" (ang. *perfect shuffle*), wykorzystywana w kryptografii. Istnieją dwie odmiany tej operacji, znane jako „wewnętrzna" (ang. *inner*) oraz „zewnętrzna" (ang. *outer*). Obydwie w wyniku dają ułożone naprzemiennie bity z dwóch połówek słowa w sposób przypominający dokładne potasowanie dwóch połówek talii złożonej z 32 kart. Różnica pomiędzy tymi dwoma odmianami polega na tym, z której części talii pobieramy pierwszą kartę. W tasowaniu zewnętrznym zewnętrzne bity pozostają na pozycjach zewnętrznych, w odmianie wewnętrznej bit na pozycji 15 przesuwa się na lewą stronę wyniku (pozycję 31). Załóżmy, że nasze słowo składa się z bitów oznaczonych w następujący sposób:

```
abcd efgh ijkl mnop ABCD EFGH IJKL MNOP
```

W wyniku tasowania zewnętrznego uzyskamy następujące słowo:

```
aAbB cCdD eEfF gGhH iIjJ kKlL mMnN oOpP
```

W wyniku tasowania wewnętrznego uzyskamy następujący wynik:

```
AaBb CcDd EeFf GgHh IiJj KkLl MmNn OoPp
```

Załóżmy, że rozmiar słowa W jest potęgą liczby dwa. W tym przypadku operację tasowania zupełnego można wykonać za pomocą instrukcji podstawowego zestawu RISC w $\log_2(W/2)$ kroków. Każdy z kroków składa się z zamiany kolejności drugiej i trzeciej ćwiartki coraz mniejszego fragmentu słowa. Ilustruje to następujący przykład:

```
abcd efgh ijkl mnop ABCD EFGH IJKL MNOP
abcd efgh ABCD EFGH ijkl mnop IJKL MNOP
abcd ABCD efgh EFGH ijkl IJKL mnop MNOP
abAB cdCD efEF ghGH ijIJ klKL mnMN opOP
aAbB cCdD eEfF gGhH iIjJ kKlL mMnN oOpP
```

Oto najprostszy sposób realizacji tego zadania:

```
x = (x & 0x0000FF00) << 8 | (x >> 8) & 0x0000FF00 | x & 0xFF0000FF;
x = (x & 0x00F000F0) << 4 | (x >> 4) & 0x00F000F0 | x & 0xF00FF00F;
x = (x & 0x0C0C0C0C) << 2 | (x >> 2) & 0x0C0C0C0C | x & 0xC3C3C3C3;
x = (x & 0x22222222) << 1 | (x >> 1) & 0x22222222 | x & 0x99999999;
```

Powyższy sposób wymaga zastosowania 42 instrukcji z podstawowego zestawu instrukcji RISC. Liczbę tę można zredukować do 30 lecz kosztem zwiększenia liczby instrukcji w przypadku procesorów o nieograniczonej liczbie jednocześnie wykonywanych, niezależnych instrukcji. W tym celu wykorzystujemy instrukcję *różnicy symetrycznej* w celu wymiany sąsiadujących pól w rejestrze (opisywanej na stronie 57). W poniższym kodzie wszystkie wartości są liczbami bez znaku:

```
t = (x ^ (x >> 8)) & 0x0000FF00;  x = x ^ t ^ (t << 8);
t = (x ^ (x >> 4)) & 0x00F000F0;  x = x ^ t ^ (t << 4);
t = (x ^ (x >> 2)) & 0x0C0C0C0C;  x = x ^ t ^ (t << 2);
t = (x ^ (x >> 1)) & 0x22222222;  x = x ^ t ^ (t << 1);
```

Operacja odwrotna, *zewnętrzne odtasowanie* (ang. *outer unshuffle*), może zostać zrealizowana za pomocą odwrócenia kolejności operacji w powyższym kodzie:

```
t = (x ^ (x >> 1)) & 0x22222222;  x = x ^ t ^ (t << 1);
t = (x ^ (x >> 2)) & 0x0C0C0C0C;  x = x ^ t ^ (t << 2);
t = (x ^ (x >> 4)) & 0x00F000F0;  x = x ^ t ^ (t << 4);
t = (x ^ (x >> 8)) & 0x0000FF00;  x = x ^ t ^ (t << 8);
```

Wykorzystanie dwóch ostatnich kroków z dowolnego z powyższych algorytmów tasowania spowoduje potasowanie każdego z bajtów osobno. Wykorzystanie ostatnich trzech kroków spowoduje potasowanie każdego z półsłów osobno itd. Podobne uwagi dotyczą operacji *odtasowania*, z tą różnicą, że kroki liczymy od początku.

W celu uzyskania *wewnętrznego tasowania zupełnego* (ang. *inner perfect shuffle*) na początku każdego z powyższych sposobów należy dodać następujące wyrażenie, zamieniające stronami połówki słowa:

```
x = (x >> 16) | (x << 16);
```

Można również zastosować *przesunięcie cykliczne* o 16 pozycji. Operację *odtasowania* można zrealizować za pomocą zakończenia procedury tym samym wyrażeniem.

Efektem zmodyfikowania algorytmu w ten sposób, że zamiana miejscami będzie dotyczyła *pierwszej* i *czwartej* ćwiartki kolejno pomniejszanych pól, jest słowo będące odwróceniem wewnętrznego tasowania zupełnego.

Warto wspomnieć o przypadku szczególnym, gdy lewa połówka słowa ma wszystkie bity o wartości 0. Innymi słowy, chcemy przenieść bity prawej połówki do co drugiego bitu, przekształcając słowo o następującej strukturze:

```
0000 0000 0000 0000 ABCD EFGH IJKL MNOP
```

W wyniku uzyskamy następujące słowo:

```
0A0B 0C0D 0E0F 0G0H 0I0J 0K0L 0M0N 0O0P
```

Algorytm zewnętrznego tasowania zupełnego można zmodyfikować w taki sposób, aby to zadanie było realizowane w 22 instrukcjach podstawowego zestawu RISC. Poniższy kod wykonuje to zadanie w 19 instrukcjach bez dodatkowego kosztu, w przypadku wykonywania go na maszynach o nieograniczonych możliwościach jednoczesnego wykonania niezależnych instrukcji (12 cykli w przypadku każdej z metod). Metoda ta nie wymaga, aby zawartość bardziej znaczącej połówki słowa była wstępnie wyczyszczona.

```
x = ((x & 0xFF00) << 8) | (x & 0x00FF);
x = ((x << 4) | x) & 0x0F0F0F0F;
x = ((x << 2) | x) & 0x33333333;
x = ((x << 1) | x) & 0x55555555;
```

Istnieje także możliwość skonstruowania podobnej do metody tasowania połówkowego metody odwrócenia tego przypadku tasowania (będącą szczególnym przypadkiem operacji *kompresji*, omówionej na stronie 137). Metoda ta wymaga od 26 do 29 instrukcji podstawowego zestawu RISC, w zależności od tego, czy wymagane jest wstępne wyczyszczenie bitów na nieparzystych pozycjach. Poniższy kod wymaga jednak od 18 do 21 instrukcji z podstawowego zestawu RISC, natomiast w przypadku maszyny obsługującej równoległe wykonanie niezależnych instrukcji kod ten może zostać wykonany w 12 do 15 cyklach procesora.

```
x = x & 0x55555555;          // jeśli konieczne
x = ((x >> 1) | x) & 0x33333333;
x = ((x >> 2) | x) & 0x0F0F0F0F;
x = ((x >> 4) | x) & 0x00FF00FF;
x = ((x >> 8) | x) & 0x0000FFFF;
```

7.3. Transponowanie macierzy bitów

Transpozycja macierzy A to taka macierz, która powstaje w wyniku przestawienia wierszy macierzy A w miejsce kolumn z zachowaniem ich kolejności. W tym podrozdziale zajmiemy się transpozycją macierzy bitów. Elementy omawianej macierzy są zgrupowane

w bajty. Wiersze i kolumny tej macierzy rozpoczynają się na granicy bajtu. Taka pozornie prosta operacja, jaką jest transpozycja tego typu macierzy jest bardzo kosztowna pod względem liczby wykorzystywanych instrukcji.

Na większości maszyn ładowanie i przechowywanie pojedynczych bitów byłoby bardzo powolne, głównie z powodu kodu, wymaganego w celu wyodrębnienia i (co gorsza) przechowywania pojedynczych bitów. Lepszy sposób polega na podzieleniu macierzy na podmacierze o rozmiarach 8×8 bitów. Każdą z takich macierzy 8×8 bitów ładujemy do rejestrów, wyliczamy transpozycję podmacierzy a następnie macierz wynikową 8×8 zapisujemy w odpowiednim miejscu macierzy wynikowej.

W niniejszym podrozdziale w pierwszej kolejności zajmiemy się problemem wyznaczania transpozycji macierzy 8×8 bitów.

Sposób przechowywania tablicy, to znaczy kolejność *wiersze-kolumny* (ang. *row-major*) lub *kolumny-wiersze* (ang. *column-major*) nie ma znaczenia. Wyznaczenie macierzy transponowanej w każdym z tych przypadków wymaga takich samych operacji. Dla celów dalszych rozważań przyjmijmy stosowanie kolejności wiersze-kolumny, w przypadku której podmacierz 8×8 jest ładowana do ośmiu rejestrów za pomocą ośmiu instrukcji *ładowania wartości z pamięci*. Oznacza to, że adresy kolejnych instrukcji *ładowania bajtów* różnią się o wielokrotność szerokości oryginalnej macierzy liczonej w bajtach. Po wykonaniu transpozycji podmacierz 8×8 zostaje umieszczona w kolumnie macierzy docelowej. Podmacierz ta jest zapisywana za pomocą czterech instrukcji *zapisu bajtu w pamięci*, z adresami poszczególnych bajtów różniącymi się od siebie o wielokrotność szerokości w bajtach tabeli docelowej (która będzie różna od szerokości w bajtach tabeli źródłowej, jeśli ta nie była kwadratowa). Załóżmy zatem, że mamy osiem 8-bitowych wartości, wyrównanych do prawej w rejestrach `a0`, `a1`, ... `a7`. Chcemy wyliczyć osiem 8-bitowych wartości, wyrównanych do prawej w rejestrach `b0`, `b1`, ... `b7`, które będą wykorzystane w instrukcjach *zapisu bajtów w pamięci*. Sytuacje tę ilustrujemy poniżej, każdy z bitów oznaczając inną cyfrą lub literą. Warto zwrócić uwagę na fakt, że główna przekątna przebiega od bitu 7. bajtu 0 do bitu 0. bajtu 7. Czytelnicy przywykli do notacji *big-endian* mogliby oczekiwać, że główna przekątna przebiega od bitu 0. bajtu 0 do bitu 7. bajtu 7.

```
a0 = 0123 4567        b0 = 08go wEMU
a1 = 89ab cdef        b1 = 19hp xFNV
a2 = ghij klmn        b2 = 2aiq yGOW
a3 = opqr stuv  ==>   b3 = 3bjr zHPX
a4 = wxyz ABCD        b4 = 4cks AIQY
a5 = EFGH IJKL        b5 = 5dlt BJRZ
a6 = MNOP QRST        b6 = 6emu CKS$
a7 = UVWX YZ$.        b7 = 7fnv DLT.
```

Najprostszy kod wykonujący to zadanie obrabiałby każdy bit indywidualnie w sposób przedstawiony poniżej. Mnożenia i dzielenia reprezentują, odpowiednio, przesunięcia w prawo lub w lewo.

```
b0 = (a0 & 128)    | (a1 & 128)/2  | (a2 & 128)/4  | (a3 & 128)/8 |
     (a4 & 128)/16 | (a5 & 128)/32 | (a6 & 128)/64 | (a7      )/128;
b1 = (a0 &  64)*2  | (a1 &  64)    | (a2 &  64)/2  | (a3 &  64)/4 |
     (a4 &  64)/8  | (a5 &  64)/16 | (a6 &  64)/32 | (a7 &  64)/64;
b2 = (a0 &  32)*4  | (a1 &  32)*2  | (a2 &  32)    | (a3 &  32)/2 |
     (a4 &  32)/4  | (a5 &  32)/8  | (a6 &  32)/16 | (a7 &  32)/32;
b3 = (a0 &  16)*8  | (a1 &  16)*4  | (a2 &  16)*2  | (a3 &  16)   |
     (a4 &  16)/2  | (a5 &  16)/4  | (a6 &  16)/8  | (a7 &  16)/16;
b4 = (a0 &   8)*16 | (a1 &   8)*8  | (a2 &   8)*4  | (a3 &   8)*2 |
     (a4 &   8)    | (a5 &   8)/2  | (a6 &   8)/4  | (a7 &   8)/8;
b5 = (a0 &   4)*32 | (a1 &   4)*16 | (a2 &   4)*8  | (a3 &   4)*4 |
     (a4 &   4)*2  | (a5 &   4)    | (a6 &   4)/2  | (a7 &   4)/4;
b6 = (a0 &   2)*64 | (a1 &   2)*32 | (a2 &   2)*16 | (a3 &   2)*8 |
     (a4 &   2)*4  | (a5 &   2)*2  | (a6 &   2)    | (a7 &   2)/2;
b7 = (a0      )*128| (a1 &   1)*64 | (a2 &   1)*32 | (a3 &   1)*16|
     (a4 &   1)*8  | (a5 &   1)*4  | (a6 &   1)*2  | (a7 &   1);
```

Powyższy kod na większości maszyn wymaga 174 instrukcji (62 *koniunkcje*, 56 *przesunięć* oraz 56 *alternatyw*). Instrukcje *alternatywy* można oczywiście zastąpić instrukcjami *dodawania*. Na procesorach PowerPC kod ten może zostać wykonany w 63 instrukcjach, co może być dość zaskakujące (siedem instrukcji *przeniesienia* wartości i 56 instrukcji *przesunięcia cyklicznego w lewo* z następującym podstawieniem wartości z zastosowaniem maski). Nie liczymy instrukcji *ładowania* i *zapisywania* wartości bajtów ani kodu niezbędnego do wyliczenia ich adresów.

Nie jest powszechnie znany żaden algorytm, który rozwiązywał by ten problem i który można by uznać za doskonały. Mimo to kolejna z omawianych technik jest dwukrotnie lepsza od powyższej, w każdym razie w przypadku procesorów obsługujących instrukcje podstawowego zestawu RISC.

W pierwszej kolejności należy potraktować macierz 8×8 jako 16 macierzy 2×2 i wykonać transpozycje każdej z tych macierzy 2×2. Następnie całą macierz 8×8 traktujemy jak cztery macierze 2×2, z których każda zawiera macierze 2×2 z poprzedniego kroku i ponownie wykonujemy transpozycję. Na końcu całą macierz traktujemy jako macierz 2×2 składającą się z macierzy z poprzedniego kroku i wykonujemy transpozycję tej macierzy. Odpowiednie przekształcenia będą miały następujący przebieg:

```
0123 4567        082a 4c6e        08go 4cks        08go wEMU
89ab cdef        193b 5d7f        19hp 5dlt        19hp xFNV
ghij klmn        goiq ksmu        2aiq 6emu        2aiq yGOW
opqr stuv  ==>   hpjr ltnv  ==>   3bjr 7fnv  ==>   3bjr zHPX
wxyz ABCD        wEyG AICK        wEMU AIQY        4cks AIQY
EFGH IJKL        xFzH BJDL        xFnV BJRZ        5dlt BJRZ
MNOP QRST        MUOW QYS$        yGOW CKS$        6emu CKS$
UVWX YZ$.        NVPX RZT.        zHPX DLT.        7fnv DLT.
```

Zamiast wykonywać opisane kroki na ośmiu niezależnych bajtach w ośmiu rejestrach, główne usprawnienie wykorzystuje możliwość spakowania wspomnianych bajtów po cztery do jednego rejestru, następnie można wykonać wymianę bitów na powstałych w ten sposób dwóch rejestrach a następnie rozpakować wynik. Kompletna procedura została zaprezentowana na listingu 7.2. Parametr `A` określa adres pierwszego bajtu podmacierzy 8×8 macierzy źródłowej o wymiarach `8m`×`8n` bitów. Podobnie parametr `B` stanowi adres pierwszego bajtu podmacierzy 8×8 macierzy docelowej o wymiarach `8n`×`8m` bitów. Oznacza to, że cała macierz źródłowa ma wymiary `8m`×`n` bajtów, natomiast macierz wyjściowa ma wymiary `8n`×`m` bajtów.

Listing 7.2. *Transponowanie macierzy 8×8 bitów*

```
void transpose8(unsigned char A[8], int m, int n,
                unsigned char B[8]) {
   unsigned x, y, t;

   // ładujemy tablicę i umieszczamy ją w x oraz y

   x = (A[0]<<24)   | (A[m]<<16)   | (A[2*m]<<8) | A[3*m];
   y = (A[4*m]<<24) | (A[5*m]<<16) | (A[6*m]<<8) | A[7*m];

   t = (x ^ (x >> 7)) & 0x00AA00AA;  x = x ^ t ^ (t << 7);
   t = (y ^ (y >> 7)) & 0x00AA00AA;  y = y ^ t ^ (t << 7);

   t = (x ^ (x >>14)) & 0x0000CCCC;  x = x ^ t ^ (t <<14);
   t = (y ^ (y >>14)) & 0x0000CCCC;  y = y ^ t ^ (t <<14);

   t = (x & 0xF0F0F0F0) | ((y >> 4) & 0x0F0F0F0F);
   y = ((x << 4) & 0xF0F0F0F0) | (y & 0x0F0F0F0F);
   x = t;

   B[0]=x>>24;    B[n]=x>>16;    B[2*n]=x>>8;  B[3*n]=x;
   B[4*n]=y>>24;  B[5*n]=y>>16;  B[6*n]=y>>8;  B[7*n]=y;
}
```

Z całą pewnością mało zrozumiały może wydać się następujący wiersz kodu:

```
t = (x ^ (x >> 7)) & 0x00AA00AA;  x = x ^ t ^ (t << 7);
```

Jego zadaniem jest zamiana miejscami w słowie `x` bitów 1. i 8. (licząc od prawej), 3. i 10., 5. i 12. itd., nie naruszając zawartości bitów 0., 2., 4. itd. Zamiana bitów miejscami jest realizowana za pomocą metody wykorzystującej różnicę symetryczną, opisanej na stronie 56. Zawartość słowa `x` przed i po pierwszej zamianie wygląda następująco:

```
0123 4567 89ab cdef ghij klmn opqr stuv
082a 4c6e 193b 5d7f goiq ksmu hpjr ltnv
```

W celu uzyskania realistycznego porównania opisanych metod, „naiwną” metodę ze strony 129 zastosowano w programie podobnym do przedstawionego na listingu 7.2. Obydwie procedury skompilowano za pomocą kompilatora GNU C na maszynie o parametrach bardzo przypominających parametry podstawowego zestawu instrukcji RISC. W wyniku uzyskano kod składający się z 219 instrukcji dla metody „naiwnej” (licząc

instrukcje *ładowania*, *zapisu*, *adresowania* oraz instrukcje przygotowujące oraz kończące procedurę) oraz 101 instrukcji dla kodu z listingu 7.2 (instrukcje wstępne i kończące nie występowały, za wyjątkiem instrukcji powrotu z rozgałęzienia). Adaptacja kodu z listingu 7.2 do 64-bitowej wersji standardowego zestawu RISC (w której x i y byłyby zapisane w tym samym rejestrze) wykona się w 85 instrukcjach.

Algorytm z listingu 7.2 wykonuje przetwarzanie od największego do najmniejszego rozdrobnienia (biorąc pod uwagę wielkość grup bitów zamienianych miejscami). Metodę tę można również zmodyfikować w taki sposób, aby wykonywała przetwarzanie od najmniejszego do największego rozdrobnienia. W tym celu traktujemy macierz 8×8 bitów jako macierz 2×2, składającą się z macierzy 4×4 bity i wykonujemy transpozycję tej macierzy. Następnie każdą z macierzy 4×4 traktujemy jako macierze 2×2 złożone z macierzy 2×2 bity i wykonujemy transpozycje tych macierzy itd. Kod wynikowy takiego algorytmu będzie taki sam, jak na listingu 7.2 za wyjątkiem trzech grup wyrażeń modyfikujących kolejność bitów, które wystąpią w odwróconej kolejności.

Transponowanie macierzy o wymiarach 32×32 bity

Podobna technika do zastosowanej w przypadku macierzy 8×8 może być oczywiście zastosowana dla macierzy o większych rozmiarach. Na przykład w przypadku macierzy 32×32 metoda ta wymaga zastosowania pięciu kroków.

Szczegóły implementacji różnią się jednak w stosunku do kodu przedstawionego na listingu 7.2, ponieważ zakładamy, że cała macierz 32×32 nie mieści się w dostępnej przestrzeni rejestrów. Dlatego należy znaleźć zwarty sposób indeksowania odpowiednich słów macierzy bitowej, za pomocą którego będzie możliwe przeprowadzenie odpowiednich operacji na bitach. Poniższy algorytm działa najlepiej w przypadku techniki od najmniejszego do największego rozdrobnienia.

W pierwszym etapie traktujemy macierz jako cztery macierze 16×16 bitów i dokonujemy ich transpozycji w następujący sposób:

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} \Rightarrow \begin{bmatrix} A & C \\ B & D \end{bmatrix}$$

A oznacza lewą połówkę pierwszych 16 słów macierzy, *B* oznacza prawą połówkę pierwszych 16 słów macierzy itd. Powyższa transformacja może zostać zrealizowana za pomocą następujących zamian:

Prawa połówka słowa 0 z lewą połówką słowa 16,

Prawa połówka słowa 1 z lewą połówką słowa 17,

...

Prawa połówka słowa 15 z lewą połówką słowa 31,

W celu implementacji tego mechanizmu posłużymy się indeksem k o wartościach od 0 do 15. W pętli kontrolowanej wartością k, prawa połówka słowa k zostanie lewą połówką słowa $k + 16$.

W drugiej fazie macierz jest traktowana jako 16 macierzy 8×8 bitów, na której przeprowadzamy następujące przekształcenie:

$$\begin{bmatrix} A & B & C & D \\ E & F & G & H \\ I & J & K & L \\ M & N & O & P \end{bmatrix} \Rightarrow \begin{bmatrix} A & E & C & G \\ B & F & D & H \\ I & M & K & O \\ J & N & L & P \end{bmatrix}$$

Transformację tę można zrealizować za pomocą następujących przekształceń:

Bity 0x00FF00FF słowa 0 zamieniamy z bitami 0xFF00FF00 słowa 8

Bity 0x00FF00FF słowa 1 zamieniamy z bitami 0xFF00FF00 słowa 9, itd.

Oznacza to, że bity 0 – 7 (osiem najmniej znaczących bitów) słowa 0 zamieniamy z bitami 8 – 15 słowa 8 itd. Indeksy pierwszego słowa w tych zamianach to $k = 0, 1, 2, 3, 4, 5, 6, 7, 16, 17, 18, 19, 20, 21, 22, 23$. Sposobem na przejście zmiennej k przez wymienione wartości jest następujące wyrażenie:

$$k' = (k + 9) \,\&\, \neg 8$$

W pętli kontrolowanej wartością zmiennej k bity słowa o indeksie k są zamieniane z bitami słowa o indeksie $k + 8$.

Podobnie trzecia faza wykonuje następujące zamiany:

Bity 0x0F0F0F0F słowa 0 zamieniamy z bitami 0xF0F0F0F0 słowa 4,

Bity 0x0F0F0F0F słowa 1 zamieniamy z bitami 0xF0F0F0F0 słowa 5, itd.

Indeksy pierwszego słowa w tych zamianach to $k = 0, 1, 2, 3, 8, 9, 10, 11, 16, 17, 18, 19, 24, 25, 26, 27$. Sposobem na przejście zmiennej k przez wymienione wartości jest następujące wyrażenie:

$$k' = (k + 5) \,\&\, \neg 4$$

W pętli kontrolowanej wartością zmiennej k, bity słowa o indeksie k są zamieniane z bitami słowa o indeksie $k + 4$.

Wynik powyższych rozważań można zakodować w języku C w dość zwarty sposób, przedstawiony na listingu 7.3.[GLS1]. Zewnętrzna pętla kontroluje pięć etapów przetwarzania, zmienna `j` przyjmuje wartości 16, 8, 4, 2 oraz 1. Pętla ta również kontroluje zmianę maski o wartościach odpowiednio dla każdego przebiegu: 0x0000FFFF, 0x00FF00FF, 0x0F0F0F0F, 0x33333333 oraz 0x55555555 (kod generujący maskę to `m = m ^ (m << j)`). Algorytm ten nie posiada wersji odwracającej operację, jest to jeden z głównych powodów najlepszego funkcjonowania przekształceń od najmniejszego

do największego rozdrobnienia macierzy. Wewnętrzna pętla jest kontrolowana wartością zmiennej `k`, która przyjmuje kolejno wartości wymienione wyżej. Wewnętrzna pętla powoduje wymianę bitów `a[k]` określonych maską `m` z bitami `a[k + j]` przesuniętymi w prawo o `j` i również określonymi maską `m`, co odpowiada bitom `a[k + j]` określonym dopełnieniem maski `m`. Kod wykonujący wspomniane zamiany bitów jest adaptacją techniki wykorzystującej trzy różnice symetryczne przedstawionej w podrozdziale *Wymiana wartości między rejestrami* na stronie 56 w kolumnie (c).

Listing 7.3. *Transpozycja macierzy o wymiarach 32×32 bity*

```
void transpose32(unsigned A[32]) {
   int j, k;
   unsigned m, t;

   m = 0x0000FFFF;
   for (j = 16; j != 0; j = j >> 1, m = m ^ (m << j)) {
      for (k = 0; k < 32; k = (k + j + 1) & ~j) {
         t = (A[k] ^ (A[k+j] >> j)) & m;
         A[k] = A[k] ^ t;
         A[k+j] = A[k+j] ^ (t << j);
      }
   }
}
```

Po skompilowaniu tej funkcji za pomocą kompilatora GNU C na maszynie o właściwościach zbliżonych do podstawowego zestawu instrukcji RISC kod ten zawiera 31 instrukcji, 20 w wewnętrznej pętli i 7 w zewnętrznej. Funkcja ta wykonuje się zatem w $4 + 5(7 + 16 \cdot 20) = 1639$ instrukcjach. Dla porównania: gdyby funkcje tę wywołano z wykorzystaniem 16 wywołań programu z listingu 7.2, wykonującego transpozycję macierzy 8×8, zajęłoby to $16(101 + 5) = 1696$ instrukcji, z założeniem, że te 16 wywołań zostałoby wykonanych jedno po drugim.

Obliczenia te obejmują również pięć instrukcji niezbędnych do wykonania wywołania funkcji (własność zaobserwowana w skompilowanym kodzie). Stąd wynika, że obydwie opisane funkcje są bardzo zbliżone pod względem czasu wykonania.

Z drugiej strony dla maszyn 64-bitowych kod z listingu 7.3 można z łatwością zmodyfikować w taki sposób, że wykonuje transpozycję macierzy 64×64 bity w około $4 + 6(7 + 32 \cdot 20) = 3886$ instrukcjach. Realizacja tego celu za pomocą 64 wywołań transpozycji macierzy 8×8 wymagałoby około $64(85 + 5) = 5760$ instrukcji.

Algorytm ten wykonuje się „w miejscu" (na oryginalnej macierzy), dlatego w przypadku transpozycji większych macierzy wymaga się dodatkowego kopiowania podmacierzy 32×32-bitowych. Algorytm ten można zmusić do dokonywania zapisu w osobnym obszarze pamięci. W tym celu należy wyodrębnić pierwszy lub ostatni krok pętli `for j` i wynik w tym kroku zapisać w osobnym obszarze pamięci[1].

[1] Jeśli zrobimy to w pierwszym kroku, unikniemy nadpisania oryginalnej zawartości macierzy źródłowej — *przyp. tłum.*

Około połowę instrukcji definiujących funkcję na listingu 7.3 stanowią instrukcje kontrolne pętli oraz funkcje odczytujące i zapisujące pięciokrotnie całą macierz. Czy rozsądne byłoby zmniejszenie tego nakładu za pomocą rozwinięcia pętli? Byłoby w przypadku, gdyby programiście zależało na jak największej prędkości wykonania i gdyby zwiększona objętość kodu nie stanowiła problemu. Ważnym też jest, aby mechanizm pobierania instrukcji (ang. *I-fetching*) w procesorze poradził sobie z płynnym wykonywaniem tak długiego bloku nierozgałęzionego kodu. Przede wszystkim jednak mechanizm ten byłby warty rozważenia, jeśli rozgałęzienia i procedury ładowania i zapisu danych z i do pamięci byłyby kosztownymi operacjami pod względem czasu wykonywania. Większość programu będzie stanowiło sześć wierszy wykonujących zamianę bitów, powtórzone 16 razy (co w wyniku da 80 wierszy). Dodatkowo program będzie potrzebował 32 instrukcji *ładowania danych*, pobierających macierz źródłową oraz 32 instrukcje *zapisu danych*, zapisujące macierz wynikową. Wszystko to da w wyniku 544 instrukcje.

Nasz kompilator GNU C nie rozwija pętli w tak dużej liczbie przebiegów (15 w wewnętrznej pętli, 5 w zewnętrznej). Listing 7.4 przedstawia wersję funkcji, w której rozwinięć dokonano ręcznie. Program ten prezentujemy w wersji niepracującej „w miejscu", jednak w razie konieczności będzie on poprawnie działał w takiej wersji. W tym celu funkcję należy wywołać z obydwoma identycznymi argumentami. W programie występuje 80 wierszy wywołujących makro `swap`. Nasz kompilator GNU C kompiluje ten kod na maszynę obsługującą podstawowy zestaw instrukcji RISC z wykorzystaniem 576 instrukcji (bez rozgałęzień, za wyjątkiem powrotu z funkcji), wliczając w to czynności przygotowawcze i kończące procedurę. Wykorzystana maszyna nie obsługuje instrukcji *ładowania* ani *zapisu wielu wartości* (ang. *load multiple* oraz *store multiple*), lecz potrafi zapisać i odczytać wartość dwóch rejestrów na raz, z wykorzystaniem instrukcji *store double* oraz *load double* (zapis oraz odczyt podwójnej wartości).

Listing 7.4. *Transpozycja macierzy o rozmiarze 32x32 bitów w postaci rozwiniętej*

```
#define swap(a0, a1, j, m) t = (a0 ^ (a1 >>j)) & m; \
                           a0 = a0 ^ t; \
                           a1 = a1 ^ (t << j);

void transpose32(unsigned A[32], unsigned B[32]) {
   unsigned m, t;
   unsigned a0, a1, a2, a3, a4, a5, a6, a7,
            a8, a9, a10, a11, a12, a13, a14, a15,
            a16, a17, a18, a19, a20, a21, a22, a23,
            a24, a25, a26, a27, a28, a29, a30, a31;

   a0  = A[ 0];  a1  = A[ 1];  a2  = A[ 2];  a3  = A[ 3];
   a4  = A[ 4];  a5  = A[ 5];  a6  = A[ 6];  a7  = A[ 7];
...
   a28 = A[28];  a29 = A[29];  a30 = A[30];  a31 = A[31];

   m = 0x0000FFFF;
   swap(a0,  a16, 16, m)
   swap(a1,  a17, 16, m)
...
```

```
    swap(a15, a31, 16, m)
    m = 0x00FF00FF;
    swap(a0,  a8,   8, m)
    swap(a1,  a9,   8, m)
...
...
    swap(a28, a29,  1, m)
    swap(a30, a31,  1, m)

    B[ 0] = a0;   B[ 1] = a1;   B[ 2] = a2;   B[ 3] = a3;
    B[ 4] = a4;   B[ 5] = a5;   B[ 6] = a6;   B[ 7] = a7;
...
    B[28] = a28;  B[29] = a29;  B[30] = a30;  B[31] = a31;
}
```

Istnieje sposób na dalsze zwiększenie wydajności, o ile maszyna obsługuje instrukcje *przesunięcia cyklicznego* (w lewo lub w prawo, bez różnicy). Pomysł polega na zastąpienie wszystkich wywołań makra `swap` na listingu 7.4 (z których każde wykorzystuje sześć instrukcji) prostszą formą zamiany, niewykorzystującą przesunięć i wykorzystującą tylko cztery instrukcje (w istniejącym makrze `swap` należy usunąć operacje *przesunięć*).

Najpierw należy przesunąć cyklicznie o 16 pozycji w prawo słowa $A[16..31]$ (to znaczy $A[k]$, gdzie $16 \le k \le 31$). Następnie należy zamienić miejscami prawe połówki słów $A[0]$ z $A[16]$, $A[1]$ z $A[17]$ itd., jak to przedstawiono na listingu 7.4. Następnie należy obrócić o osiem pozycji w prawo słowa $A[0..8]$ oraz $A[24..31]$ i zamienić miejscami bity oznaczone w masce 0x00FF00FF w słowach $A[0]$ z $A[8]$, $A[1]$ z $A[9]$ itd., jak to przedstawiono na listingu 7.4. Po pięciu etapach takich zamian transpozycja nie będzie jeszcze wykonana. Należy na końcu przesunąć cyklicznie $A[1]$ w lewo o jedną pozycję, $A[2]$ o dwie pozycje itd. (31 instrukcji). Nie prezentujemy kompletnego kodu, natomiast przedstawiamy przykład działania dla macierzy 4×4.

```
abcd       abcd       abij       abij       aeim       aeim
efgh  ==>  efgh  ==>  efmn  ==>  nefm  ==>  nbfj  ==>  bfjn
ijkl       klij       klcd       klcd       kocg       cgko
mnop       opmn       opgh       hopg       hlpd       dhlp
```

Fragment programu na listingu 7.4 odpowiedzialny za manipulację bitami wykorzystuje 480 instrukcji (80 zamian po sześć instrukcji każda). Zmodyfikowany program wykorzystujący instrukcję *przesunięcia cyklicznego* wykorzystuje 80 zamian po cztery instrukcje każda, plus 80 instrukcji *przesunięcia cyklicznego* ($16 \cdot 5$) w pięciu etapach plus końcowe 31 instrukcji *przesunięcia cyklicznego*, co w sumie daje 431 instrukcji. Kod przygotowujący oraz kończący funkcję nie ulegnie zmianie, zatem wykorzystanie *przesunięć cyklicznych* pozwala na zaoszczędzenie 49 instrukcji.

Istnieje inny sposób realizacji transpozycji macierzy, wykorzystujący trzy *przekształcenia poprzeczne* (ang. *shearing transformation*) [GLS1]. W przypadku, gdy macierz ma wymiary $n \times n$ wykorzystywane są następujące etapy:

- przesunięcie cykliczne wiersza i w prawo o i bitów;
- przesunięcie cykliczne kolumny j w górę o $(j + 1) \bmod n$ bitów;
- przesunięcie cykliczne wiersza i w prawo o $(i + 1) \bmod n$ bitów;
- pionowe odbicie macierzy względem środka.

W celu ilustracji tej techniki przedstawiamy jej zastosowanie na macierzy 4×4:

```
abcd      abcd      hlpd      dhlp      aeim
efgh ==>  hefg ==>  kocg ==>  cgko ==>  bfjn
ijkl      klij      nbfj      bfjn      cgko
mnop      nopm      aeim      aeim      dhlp
```

Metoda ta nie stanowi konkurencji dla pozostałych z powodu dużej kosztowności kroku 2. Aby wykonać ten krok w rozsądnej liczbie operacji, należałoby wykonać przesunięcie cykliczne o $n/2$ pozycji wszystkich kolumn, które wykonują przesunięcie o $n/2$ lub większą liczbę pozycji (są to kolumny od $n/2 - 1$ do $n - 2$] a następnie wykonać przesunięcie cykliczne odpowiednich kolumn o $n/4$ pozycji w górę itd. Kroki 1. oraz 3. wymagają tylko $n - 1$ instrukcji, natomiast krok 4. nie wymaga żadnych instrukcji, jeśli wyniki są od razu zapisywane do odpowiednich obszarów w pamięci.

W przypadku macierzy 8×8 zapisanej w pojedynczym słowie 64-bitowym w oczywisty sposób (to znaczy górny wiersz macierzy jest zapisany w najbardziej znaczących ośmiu bitach rejestru) operacja *transpozycji* jest równoważna trzem operacjom zewnętrznego tasowania zupełnego i odtasowania [GLS1]. Jest to doskonały sposób realizacji tego zadania, o ile maszyna udostępnia instrukcję *tasowania*, lecz w przypadku maszyny ze standardowym zestawem instrukcji RISC nie jest to dobre rozwiązanie.

7.4. Kompresja lub uogólniona ekstrakcja

Język programowania APL posiada operację *kompresji* (ang. *compress*) zapisywaną B/V, gdzie B jest wektorem bitów, natomiast V jest wektorem o tej samej długości co B, zawierającym dowolne elementy. Wynikiem operacji jest wektor składający się z elementów V, dla których odpowiadające im bity wektora B mają wartość 1. Długość wektora wynikowego jest równa liczbie jedynek w B.

W niniejszym podrozdziale zajmiemy się podobną operacją na bitach w słowie. Po podaniu maski $\boldsymbol{m}$ i słowa $\boldsymbol{x}$, bity w $\boldsymbol{x}$, dla których odpowiednie bity w masce $\boldsymbol{m}$ mają wartość 1 zostają skopiowane do wyniku i przesunięte (skompresowane) do prawej. Załóżmy na przykład, że operujemy na słowie $\boldsymbol{x}$ o następującej strukturze:

```
abcd efgh ijkl mnop qrst uvx yzAB CDEF
```

Maska w naszej operacji będzie następująca:

```
0000 1111 0011 0011 1010 1010 0101 0101
```

W takim przypadku wynikiem operacji `compress` będzie następujące słowo:

```
0000 0000 0000 0000 efgh klop qsuw zBDF
```

Operację tę można również nazwać *uogólnioną ekstrakcją* (ang. *generalized extract*), przez analogię do instrukcji `extract` udostępnianej przez wiele komputerów.

Interesuje nas kod wykonujący te operacje o jak najmniejszym koszcie wykonawczym w najgorszym przypadku. Jako element wyjściowy posłużymy się kodem na listingu 7.5, który postaramy się usprawnić. Kod ten nie wykorzystuje rozgałęzień i wykonuje się w najgorszym wypadku w 260 instrukcjach, wliczając operacje wstępne i końcowe.

Listing 7.5. *Prosta forma operacji compress*

```
unsigned compress(unsigned x, unsigned m) {
   unsigned r, s, b;    // wynik, przesunięcie, maska bitowa

   r = 0;
   s = 0;
   do {
      b = m & 1;
      r = r | ((x & b) << s);
      s = s + b;
      x = x >> 1;
      m = m >> 1;
   } while (m != 0);
   return r;
}
```

Można usprawnić ten kod za pomocą metody prefiksu równoległego (zob. rozdział 5., strona 85) z wykorzystaniem operacji *różnicy symetrycznej* [GLS1]. Operację *prefiksu równoległego* z wykorzystaniem różnicy symetrycznej oznaczmy PP-XOR. Główny pomysł polega tu na wyodrębnieniu bitów argumentu x, które należy przesunąć w prawo o nieparzystą liczbę pozycji i na wykonaniu tego przesunięcia. Operację tę można uprościć przez koniunkcję x z maską w celu usunięcia niepotrzebnych bitów. Bity maski przesuwamy w ten sam sposób. Następnie identyfikujemy te bity w x, które należy przesunąć o liczbę miejsc będącą nieparzystą wielokrotnością dwójki (2, 6, 10 itd.) i przesuwamy te bity w x oraz w masce. Następnie identyfikujemy i przesuwamy bity, które należy przesunąć o nieparzystą wielokrotność liczby 4, następnie powtarzamy tę operację dla nieparzystej wielokrotności 8 a następnie dla bitów, które należy przesunąć o 16 pozycji.

Algorytm ten (którego pierwszą publikację przypisuje się [GLS1]) wydaje się być dość trudny do zrozumienia i taki sposób zrealizowania czegokolwiek nie wydaje się być w ogóle możliwy, zatem operacje wykorzystywane przez ten algorytm omówimy z nieco większą szczegółowością. Załóżmy, że naszymi wartościami wejściowymi są:

```
x = abcd efgh ijkl mnop qrst uvwx yzAB CDEF
m = 1000 1000 1110 0000 0000 1111 0101 0101
    1    1    111
    9    6    333            4444  3 2  1 0
```

Każda litera w `x` symbolizuje jeden bit (o wartości 0 lub 1). Liczba poniżej każdej jedynki w masce oznacza liczbę miejsc, o które należy przesunąć w prawo dany bit słowa `x`. Jest to liczba zer w masce na prawo od tego miejsca. Jak wspomniano wcześniej, wygodnie jest oczyścić `x` z niepotrzebnych bitów, co da w wyniku:

```
x = a000 e000 ijk0 0000 0000 uvwx 0z0B 0D0F
```

Plan polega na określeniu bitów, które przesuwają się o nieparzystą liczbę miejsc w prawo i przesunąć je o jedną pozycję. Przypomnijmy, że operacja PP-XOR w wyniku daje 1 na każdej pozycji, na której liczba jedynek od tego miejsca (włącznie) w prawo jest nieparzysta. My chcemy natomiast zidentyfikować miejsca, od których w prawo liczba zer jest nieparzysta. Możemy to wyliczyć za pomocą zmiennej pomocniczej `mk = ~m << 1`, wyliczając na niej PP-XOR.

Otrzymamy:

```
mk = 1110 1110 0011 1111 1110 0001 0101 0100
mp = 1010 0101 1110 1010 1010 0000 1100 1100
```

Zaobserwujemy, że `mk` identyfikuje bity, które zawierają zera bezpośrednio po swojej prawej stronie, natomiast `mp` sumuje te zera, modulo 2, od prawej strony. W ten sposób `mp` identyfikuje bity w `m`, które posiadają nieparzystą liczbę zer po swojej prawej stronie.

Bity, które chcemy przesunąć o 1 to są te bity, które posiadają nieparzystą liczbę zer po swojej prawej stronie (zidentyfikowane przez `mp`) oraz w oryginalnej masce mają wartość 1. Dlatego bity te zidentyfikujemy za pomocą `mv = mp & m`:

```
mv = 1000 0000 1110 0000 0000 0000 0100 0100
```

Bity te można przesunąć za pomocą następującego wyrażenia:

```
m = (m ^ mv) | (mv >> 1);
```

Natomiast odpowiadające im bity w `x` przesuwamy za pomocą następującego wyrażenia:

```
t = x & mv;
x = (x ^ t) | (t >> 1);
```

Przesuwanie odpowiednich bitów w `m` jest prostsze, ponieważ wszystkie odpowiednie bity mają wartość 1. W tym przypadku operacja *różnicy symetrycznej* wyłącza bity, będące jedynkami w `x`, natomiast *alternatywa* bitowa włącza bity mające wartość 0 w `m` oraz w `x`. Operacja ta może również wykorzystywać dwie operacje *różnicy symetrycznej* lub, odpowiednio, *odejmowania* i *dodawania*. Wyniki, po odpowiednim przesunięciu wybranych bitów, będą wyglądać następująco:

```
m = 0100 1000 0111 0000 0000 1111 0011 0011
x = 0a00 e000 0ijk 0000 0000 uvwx 00xB 00DF
```

W następnej kolejności musimy przygotować maskę dla drugiej iteracji, gdzie nastąpi wykrywanie bitów, które należy przesunąć w prawo o nieparzystą wielokrotność dwójki. Zauważmy, że wartość `mk & ~mp` identyfikuje bity, które mają wartość 0 bezpośrednio z prawej strony w oryginalnej masce `m` oraz które posiadają parzystą liczbę zer po prawej stronie w oryginalnej masce. Własności te łączą się, tworząc własności zmienionej maski `m` (oznacza to, że `mk` identyfikuje *wszystkie* miejsca w nowej masce `m`, które sąsiadują z prawej strony z zerem oraz posiadają parzystą liczbę zer ze swojej prawej strony). Wartość ta, po podsumowaniu przez PP-XOR, zidentyfikuje bity, które należy przesunąć w prawo o nieparzystą wielokrotność dwójki (2, 6, 10 itd.) Procedura nasza będzie polegała na przypisaniu tej własności do `mk` i wykonaniu drugiej iteracji wymienionych kroków. Nowa wartość `mk` wynosi:

```
mk = 0100 1010 0001 0101 0100 0001 0001 0000
```

Kompletna funkcja w C operacji *kompresji* została przedstawiona na listingu 7.6. Wykonuje się w stałej liczbie 127 instrukcji podstawowego zestawu RISC, wliczając czynności przygotowawcze oraz kończące funkcję. Listing 7.7 przedstawia sekwencję wartości przyjmowanych przez różne zmienne w kluczowych punktach wyliczeń, z tymi samymi wartościami, które zostały użyte w powyższej dyskusji. Zauważmy, że produktem ubocznym tej procedury jest skompresowana wersja oryginalnej maski `m`.

Listing 7.6. *Procedura compress z wykorzystaniem prefiksu równoległego*

```
unsigned compress(unsigned x, unsigned m) {
   unsigned mk, mp, mv, t;
   int i;

   x = x & m;           // czyścimy niepotrzebne bity
   mk = ~m << 1;        // będziemy liczyć zera po prawej stronie

   for (i = 0; i < 5; i++) {
      mp = mk ^ (mk << 1);              // prefiks równoległy
      mp = mp ^ (mp << 2);
      mp = mp ^ (mp << 4);
      mp = mp ^ (mp << 8);
      mp = mp ^ (mp << 16);
      mv = mp & m;                      // bity do przesunięcia
      m = m ^ mv | (mv >> (1 << i));    // kompresujemy m
      t = x & mv;
      x = x ^ t | (t >> (1 << i));      // kompresujemy x
      mk = mk & ~mp;
   }
   return x;
}
```

Listing 7.7. *Wykorzystanie prefiksu równoległego przy operacji compress*

```
                    x = abcd efgh ijkl mnop qrst uvwx yzAB CDEF
                    m = 1000 1000 1110 0000 0000 1111 0101 0101
                    x = a000 e000 ijk0 0000 0000 uvwx 0z0B 0D0F
i = 0,             mk = 1110 1110 0011 1111 1110 0001 0101 0100
po wyliczeniu PP   mp = 1010 0101 1110 1010 1010 0000 1100 1100
                    m = 0100 1000 0111 0000 0000 1111 0011 0011
                    x = 0a00 e000 0ijk 0000 0000 uvwx 00xB 00DF
```

```
i = 1,            mk = 0100 1010 0001 0101 0100 0001 0001 0000
po wyliczeniu PP  mp = 1100 0110 0000 1100 1100 0000 1111 0000
                  mv = 0100 0000 0000 0000 0000 0000 0011 0000
                   m = 0001 1000 0111 0000 0000 1111 0000 1111
                   x = 000a e000 0ijk 0000 0000 uvwx 0000 zBDF

i = 2,            mk = 0000 1000 0001 0001 0000 0001 0000 0000
po wyliczeniu PP  mp = 0000 0111 1111 0000 1111 1111 0000 0000
                  mv = 0000 0000 0111 0000 0000 1111 0000 0000
                   m = 0001 1000 0000 0111 0000 0000 1111 1111
                   x = 000a e000 0000 0ijk 0000 0000 uvwx zBDF

i = 3,            mk = 0000 1000 0000 0001 0000 0000 0000 0000
po wyliczeniu PP  mp = 0000 0111 1111 1111 0000 0000 0000 0000
                  mv = 0000 0000 0000 0111 0000 0000 0000 0000
                   m = 0001 1000 0000 0000 0000 0111 1111 1111
                   x = 000a e000 0000 0000 0000 0ijk uvwx zBDF

i = 4,            mk = 0000 1000 0000 0000 0000 0000 0000 0000
po wyliczeniu PP  mp = 1111 1000 0000 0000 0000 0000 0000 0000
                  mv = 0001 1000 0000 0000 0000 0000 0000 0000
                   m = 0000 0000 0000 0000 0001 1111 1111 1111
                   x = 0000 0000 0000 0000 000a eijk uvwx zBDF
```

Na 64-bitowej maszynie obsługującej podstawowy zestaw instrukcji RISC algorytm z listingu 7.6. wymaga 169 instrukcji, natomiast najgorszy przypadek algorytmu z listingu 7.5 wykorzystuje 516 instrukcji.

Liczbę instrukcji wykorzystywanych przez algorytm z listingu 7.6 można jeszcze zredukować, o ile maska `m` jest stała. Takie założenie można przyjąć w dwóch przypadkach:

1. Wykonanie funkcji `compress(x, m)` następuje w pętli, w której wartość `m` jest niezmienna, choć może nie być znana z góry.
2. Wartość `m` jest znana z góry i kod funkcji `compress` jest generowany z wyprzedzeniem, na przykład przez kompilator.

Zauważmy, że wartość przypisywana zmiennej `x` w pętli na listingu 7.6 nie jest wykorzystywana gdziekolwiek w pętli za wyjątkiem przypisań zmiennej `x`. Wartość zmiennej `x` zależy wyłącznie od niej samej i od wartości zmiennej `mv`. Dzięki temu można zmodyfikować pętlę w taki sposób, że wszystkie odwołania do `x` zostaną usunięte, natomiast pięć kolejnych wartości `mv` zostanie zapisanych w zmiennych `mv0`, `mv1`, … `mv4`. W pierwszym opisanym wyżej przypadku funkcja nie wykorzystująca odwołań do `x` może zostać umieszczona poza pętlą, w której występuje wywołanie `compress(x, m)`, natomiast w pętli umieścimy następujące wyrażenia:

```
x = x & m;
t = x & mv0; x = x ^ t | (t >> 1);
t = x & mv1; x = x ^ t | (t >> 2);
t = x & mv2; x = x ^ t | (t >> 4);
t = x & mv3; x = x ^ t | (t >> 8);
t = x & mv4; x = x ^ t | (t >> 16);
```

Dzięki temu w pętli mamy tylko 21 instrukcji (ładowanie stałych można zrealizować poza pętlą), co stanowi spore usprawnienie w porównaniu do 127 instrukcji wykorzystanych w pełnej wersji z listingu 7.6.

W drugim opisanym przypadku, w którym wartość `m` jest znana z góry, można zrobić podobną modyfikację, lecz możliwe są dalsze usprawnienia. Być może jedna z pięciu masek ma wartość 0, w tym przypadku można pominąć jeden z powyższych wierszy przypisań. Na przykład załóżmy, że maska `mv1` jest równa 0, co oznacza, że w `x` nie ma żadnych pozycji, które należy przesunąć o nieparzystą liczbę miejsc, natomiast `mv4` będzie równe 0 w przypadku, gdy żaden bit nie musi być przesunięty o więcej niż 15 pozycji w prawo itd.

Za przykład przyjmijmy następującą maskę:

```
m = 0101 0101 0101 0101 0101 0101 0101 0101
```

Maski kolejnych etapów będą miały następujące wartości:

```
mv0 = 0100 0100 0100 0100 0100 0100 0100 0100
mv1 = 0011 0000 0011 0000 0011 0000 0011 0000
mv2 = 0000 1111 0000 0000 0000 1111 0000 0000
mv3 = 0000 0000 1111 1111 0000 0000 0000 0000
mv4 = 0000 0000 0000 0000 0000 0000 0000 0000
```

Kod może zostać skompilowany z pominięciem zbędnych instrukcji, ponieważ ostatnia maska ma wartość 0, dzięki czemu operacja *kompresji* wykona się w 17 instrukcjach (nie licząc ładowania masek do rejestrów). Wynik ten nie jest tak dobry, jak przedstawiony na stronie 128 (13 instrukcji, nie licząc ładowania masek), która wykorzystuje fakt wyboru naprzemiennych bitów.

Wykorzystanie instrukcji wstawiania i ekstrakcji

W przypadku, gdy procesor udostępnia instrukcje *wstawiania* (ang. *insert*), najlepiej z wartościami bezpośrednimi określającymi pole bitowe w rejestrze wynikowym, liczba wykorzystanych instrukcji może zostać jeszcze zmniejszona. Wykorzystując instrukcję wstawiania możemy również uniknąć blokowania rejestrów przechowujących maski.

Rejestr wynikowy jest ustawiany na wartość 0 a następnie dla każdej ciągłej grupy jedynek w masce `m` zmienna `x` jest przesuwana w prawo, wyrównując do prawej kolejne pole. Instrukcja *insert* służy do wstawienia odpowiednich bitów zmiennej `x` na odpowiednie miejsce w rejestrze wynikowym. Dzięki temu cała operacja wykorzystuje $2n + 1$ instrukcji, gdzie n oznacza liczbę pól (grup sąsiadujących jedynek) w masce. W najgorszym przypadku operacja potrzebuje 33 instrukcje, ponieważ największa możliwa liczba pól wynosi 16 (co stanowi maskę z naprzemiennych jedynek i zer).

Przykład sytuacji, w której metoda wykorzystująca wstawianie wykona się w znacznie mniejszej liczbie instrukcji stanowi maska `m = 0x0010084A`. Wykonanie kompresji z wykorzystaniem tej maski wymaga przesunięcia bitów o 1, 2, 4, 8 oraz 16 pozycji. W przypadku metody wykorzystującej prefiks równoległy algorytm wykona się w 21

instrukcjach, lecz w przypadku zastosowania wstawiania potrzebne jest tylko 11 instrukcji (w masce występuje pięć pól jedynek). Bardziej ekstremalnym przypadkiem jest na przykład `m = 0x8000000`. W tym przypadku należy przesunąć tylko jeden bit o 31 pozycji, co w przypadku metody wykorzystującej prefiks równoległy wymaga 21 instrukcji, natomiast metoda wykorzystująca wstawianie wykorzysta tylko jedną instrukcję (*przesunięcie w prawo* o 31).

W operacji *kompresji* ze znaną maską można również wykorzystać instrukcję *ekstrakcji* (ang. *extract*), dzięki czemu algorytm będzie wymagać $3n - 2$ instrukcji, gdzie n jest liczbą pól jedynek w masce.

Z naszych rozważań wynika jednoznaczny wniosek, że wybór optymalnego kodu realizującego operację *kompresji* ze znaną maską nie jest łatwym zadaniem.

Kompresja do lewej strony

W celu wykonania kompresji bitów do lewej strony można oczywiście wykorzystać dopełnienie argumentu `x` oraz maski, skompresować je do prawej strony a następnie wykonać odbicie wyniku. Inny sposób polega na skompresowaniu do prawej a następnie przesunięcie wyniku w lewo o $\mathrm{pop}(\overline{m})$ pozycji. Sposoby te mogą dawać zadowalające wyniki, jeśli wykorzystywany procesor udostępnia instrukcje wykonujące *odbicie bitowe* (ang. *bit reversal*) lub instrukcję *zliczania populacji*. W przeciwnym wypadku można w prosty sposób zaadaptować algorytm z listingu 7.6. Po prostu zmieniamy kierunek wykonywanych przesunięć, za wyjątkiem dwóch wykonujących `1 << i` (osiem wyrażeń do zmiany).

7.5. Uogólnione permutacje, operacja typu „owce i kozły"

W celu wykonania operacji ogólnych permutacji bitów w słowie lub jakiejkolwiek uporządkowanej strukturze danych, pierwszym problemem, z którym należy się zmierzyć jest sposób reprezentacji permutacji. Tego typu operacji nie można reprezentować w sposób bardzo zwarty. Ponieważ w słowie 32-bitowym istnieje 32! możliwych permutacji, w celu reprezentacji jednej z możliwych permutacji potrzeba co najmniej $\lceil \log_2(32!) \rceil = 118$ bitów, czyli trzy słowa plus 22 bity.

Jeden z interesujących sposobów reprezentacji permutacji jest związany z operacjami *kompresji*, omówionymi w podrozdziałach 7.1 – 7.4 [GLS1]. Rozpoczynamy od bezpośredniej metody określającej pozycje, na które mają przejść poszczególne bity. Na przykład weźmy pod uwagę permutację dokonywaną za pomocą przesunięcia cyklicznego w lewo o cztery pozycje, bit na pozycji 0 przesuwa się na pozycję 4, 1 na pozycję 5, … 31 przesuwa się na pozycję 3. Permutacja ta może zostać zapisana w postaci wektora 32 5-bitowych indeksów:

```
00100
00101
 ...
11111
00000
00001
00010
00011
```

Traktując te wektory jako macierz bitową należy wykonać transpozycję macierzy a następnie odbić ją według przekątnej w taki sposób, że górny wiersz będzie zawierać najmniej znaczące bity wektorów a wynik będzie w schemacie *little-endian*. W ten sposób powyższe wektory możemy przechować w postaci następującej macierzy bitowej:

```
p[0] = 1010 1010 1010 1010 1010 1010 1010 1010
p[1] = 1100 1100 1100 1100 1100 1100 1100 1100
p[2] = 0000 1111 0000 1111 0000 1111 0000 1111
p[3] = 0000 1111 1111 0000 0000 1111 1111 0000
p[4] = 0000 1111 1111 1111 1111 0000 0000 0000
```

Każdy bit słowa `p[0]` zawiera najmniej znaczący bit numeru pozycji, na którą przesuwa się odpowiedni bit słowa `x`. Każdy bit słowa `p[1]` zawiera kolejny bit numeru pozycji itd. Jest to sytuacja podobna do zapisu maski w `mv`, wykorzystywanego w poprzednim podrozdziale, z tą różnicą, że `mv` odnosiło się do nowej maski w algorytmie kompresji, nie do oryginalnej maski.

Operacja *kompresji*, która jest nam teraz potrzebna, musi skompresować do lewej wszystkie bity oznaczone w masce wartością 1, natomiast do prawej skompresować wszystkie bity oznaczone w masce wartością 0[2]. Sposób ten nazywany jest czasem operacją rozdzielania „owiec i kozłów[3]" (ang. *sheep and goats — SAG*) lub *uogólnionym odtasowaniem* (ang. *generalized unshuffle*). Wynik takiej operacji można wyznaczyć za pomocą następującego wyrażenia:

```
SAG(x, m) = compress_left(x, m) | compress(x, ~m)
```

Wykorzystując SAG jako operacje podstawową oraz permutację `p` opisaną wyżej, wynik przekształcenia słowa `x` można wyznaczyć w następujących 15 krokach:

```
x    = SAG(x,    p[0]);
p[1] = SAG(p[1], p[0]);
p[2] = SAG(p[2], p[0]);
p[3] = SAG(p[3], p[0]);
p[4] = SAG(p[4], p[0]);

x    = SAG(x,    p[1]);
p[2] = SAG(p[2], p[1]);
```

[2] W przypadku, gdy wykorzystywany jest format *big-endian* do lewej należy skompresować bity oznaczone w masce wartością 0, natomiast do prawej bity oznaczone wartością 1.

[3] Nazwa pochodzi z jednej z przypowieści w ewangelii według św. Mateusza (Mat 25, 32–33): „I zgromadzą się przed Nim wszystkie narody, a On oddzieli jednych [ludzi] od drugich, jak pasterz oddziela owce od kozłów. Owce postawi po prawej, a kozły po swojej lewej stronie". Z łatwością można dostrzec analogię — rozdzielanie bitów na prawą i lewą stronę rejestru wynikowego — *przyp. red.*

```
p[3] = SAG(p[3], p[1]);
p[4] = SAG(p[4], p[1]);

x    = SAG(x,    p[2]);
p[3] = SAG(p[3], p[2]);
p[4] = SAG(p[4], p[2]);

x    = SAG(x,    p[3]);
p[4] = SAG(p[4], p[3]);

x    = SAG(x,    p[4]);
```

W tych krokach operacja SAG jest wykorzystywana w celu wykonania *stabilnego sortowania* o podstawie 2 (ang. *stable binary radix sort*). W pierwszym kroku wszystkie bity w `x`, dla których `p[0] = 1`, zostają przesunięte do lewej połówki wynikowego słowa, natomiast wszystkie bity, dla których `p[0] = 0`, zostają przeniesione do prawej połówki. Oprócz tych przesunięć kolejność bitów nie ulega zmianie (stąd właśnie to sortowanie nosi miano „stabilnego"). W kolejnych wierszach w podobny sposób sortowane są klucze wykorzystywane w następnym etapie sortowania. Szósty wiersz kodu wykonuje sortowanie `x` na podstawie drugiego mniej znaczącego bitu klucza itd.

Podobnie do omawianej wcześniej kompresji, w sytuacji, gdy permutacja `p` jest wykorzystana na większej liczbie słów `x` możemy uzyskać znaczny przyrost wydajności, jeśli wstępnie wyliczymy parametry niezbędne do poszczególnych etapów sortowania. Macierz permutacji można rozpisać w następujący sposób:

```
p[1] = SAG(p[1], p[0]);
p[2] = SAG(SAG(p[2], p[0]), p[1]);
p[3] = SAG(SAG(SAG(p[3], p[0]), p[1]), p[2]);
p[4] = SAG(SAG(SAG(SAG(p[4], p[0]), p[1]), p[2]), p[3]);
```

Następnie każda z permutacji jest wykonywana w następujący sposób:

```
x = SAG(x, p[0]);
x = SAG(x, p[1]);
x = SAG(x, p[2]);
x = SAG(x, p[3]);
x = SAG(x, p[4]);
```

Bardziej bezpośredni (i zapewne mniej ciekawy) sposób reprezentacji uogólnionych permutacji bitów w słowie polega na zapisie permutacji w postaci sekwencji 32 5-bitowych indeksów.

Indeks o numerze *k* stanowi numer bitu w źródle, z którego pochodzi *k*-ty bit wyniku (jest to lista typu „pochodzi z", w odróżnieniu od metody SAG, w której lista określa sytuację „przechodzi do"). Informacje te można upakować w sześciu słowach 32-bitowych, co wymaga zastosowania sześciu słów do przechowania 32-bitowych indeksów. Instrukcję tę można zaimplementować sprzętowo, na przykład w postaci:

```
bitgather Rt,Rx,Ri,
```

Rejestr `Rt` jest rejestrem wynikowym oraz źródłowym, rejestr `Rx` zawiera bity, które chcemy poddać permutacji, rejestr `Ri` zawiera natomiast sześć 5-bitowych indeksów (z dwoma nieużywanymi bitami). Instrukcja ta wykonuje następującą operację:

$$\boldsymbol{t} \leftarrow (\boldsymbol{t} << 6) \mid \boldsymbol{x}_{i_0}\boldsymbol{x}_{i_1}\boldsymbol{x}_{i_2}\boldsymbol{x}_{i_3}\boldsymbol{x}_{i_4}\boldsymbol{x}_{i_5}$$

Zawartość rejestru wynikowego $\boldsymbol{t}$ zostaje przesunięta w lewo o sześć pozycji a w miejsce powstałe w efekcie tego przesunięcia zostają wstawione wybrane bity słowa $\boldsymbol{x}$. Pobrane bity są określone sześcioma 5-bitowymi indeksami słowa $\boldsymbol{i}$ w kolejności od lewej do prawej. Kolejność bitów w indeksach może być interpretowana w formacie *little-endian* lub *big-endian* a wybór najprawdopodobniej byłby dostosowany do specyfiki konkretnego procesora.

W celu dokonania permutacji słowa należy zastosować sekwencję sześciu takich instrukcji, wszystkie z tymi samymi wartościami `Rt` oraz `Rx` lecz z różnymi rejestrami indeksów. W pierwszym rejestrze indeksowym w sekwencji znaczenie miałyby tylko indeksy i_4 oraz i_5, ponieważ bity wybrane przez pozostałe cztery indeksy zostaną i tak przesunięte poza wynik `Rt`.

Implementacja tej metody z całą pewnością pozwala na powtórzenie wartości indeksów, zatem instrukcja ta może zostać użyta również w innym celu niż permutacja bitów. Można ją zastosować w celu powtórzenia dowolnego bitu dowolną liczbę razy w rejestrze wynikowym. Operacja SAG nie posiada własności pozwalającej na takie uogólnienie.

Implementacja tej instrukcji w postaci szybkiej instrukcji (tzn. wykonującej się w pojedynczym cyklu procesora) nie jest zadaniem niewykonalnym. Układ wyboru bitów składa się tutaj z sześciu multiplekserów MUX 32:1. W przypadku, gdyby zbudować je z pięciu segmentów multiplekserów MUX 2:1 we współczesnej technologii ($6 \cdot 31 = 186$ MUX w sumie), instrukcja ta byłaby szybsza od 32-bitowej instrukcji *dodawania* [MD].

Operacja permutacji bitów ma zastosowanie w kryptografii, natomiast bardzo zbliżona operacja permutacji fragmentów słów (na przykład permutacja bajtów w słowie) ma zastosowanie w grafice komputerowej. Obydwa te zastosowania będą operować na wartościach 64-bitowych lub wręcz na 128-bitowych, natomiast na wartościach 32-bitowych — zdecydowanie rzadziej. Operacje SAG oraz *bitgather* dają się oczywiście zmodyfikować w prosty sposób również dla słów o wspomnianych wielkościach.

W celu wykonania szyfrowania lub odszyfrowania komunikatu za pomocą standardu DES (Data Encryption Standard) należy wykonać szereg przekształceń podobnych do permutacji. W pierwszej kolejności jest wykonywane generowanie klucza i zachodzi to jednorazowo podczas sesji. Operacja ta wymaga zastosowania 17 odwzorowań przypominających permutacje. Pierwsze z nich, nazywane *permuted choice 1* dokonuje odwzorowania 64-bitowej wartości na wartość 56-bitową (wybiera 56 bitów pomijając bit parzystości i wykonuje na nich permutację). Następnie wykonywane jest 16 odwzorowań permutacyjnych z 56 bitów na 48 bitów, z których wszystkie wykorzystują to samo odwzorowanie, zwane *permuted choice 2*.

Po etapie generowania klucza każdy 64-bitowy blok bitów w komunikacie zostaje poddany 34 operacjom permutacyjnym. Pierwszą i ostatnią operację stanowią permutacje 64-bitowej wartości, z których jedna stanowi odwrotność drugiej. Wykonuje się 16 permutacji z powtórzeniami, które odwzorowują wartości 32-bitowe na wartości 48-bitowe, wykorzystując to samo odwzorowanie. Na końcu wykonywane jest 16 32-bitowych permutacji, wykorzystujących tę samą permutację. W całej operacji wykorzystywane jest wiele powtórzeń sześciu różnych odwzorowań. Są one stałe i zostały opisane w [DES].

Algorytm DES jest przestarzały i w roku 1998 organizacja *Electronic Frontier Foundation* udowodniła, że z zastosowaniem specjalnego sprzętu jest możliwe złamanie algorytmu DES. W ramach tymczasowego rozwiązania *National institute of Standards and Technology* (*NIST*) opracował rozwiązanie pod nazwą Triple DES, które polega na trzykrotnym wykonaniu algorytmu DES na każdym z 64-bitowych bloków, za każdym razem wykorzystując inny klucz (co oznacza, że klucz składa się z 192 bitów, wliczając 24 bity parzystości). Z tego powodu algorytm ten wymaga wykonania trzykrotnie większej liczby permutacji niż ma to miejsce w standardowym algorytmie DES zarówno w operacjach szyfrowania, jak i deszyfrowania.

„Prawdziwy" następca algorytmów DES oraz Triple DES, algorytm *AES* (ang. *Advanced Encription Standard*, znany wcześniej pod nazwą algorytmu Rijndael) w ogóle nie wykorzystuje permutacji na poziomie bitów. Jedyną operacją zbliżoną do permutacji jest przesunięcie cykliczne 32-bitowych słów o wielokrotności liczby 8. Inne metody kryptograficzne proponowane lub rzeczywiście wykorzystywane z reguły wykorzystują znacznie mniejszą liczbę permutacji bitów niż ma to miejsce w algorytmie DES.

Porównując dwa sposoby implementacji permutacji, sposób wykorzystujący „instrukcję" *bitgather* ma następujące zalety:

- prostszy proces przygotowania indeksów z danych opisujących permutację;
- prostszy sposób implementacji sprzętowej;
- bardziej uogólnione odwzorowania.

Metoda SAG posiada natomiast następujące zalety:

- permutacja jest wykonywana w pięciu, zamiast sześciu instrukcjach;
- w przypadku implementacji sprzętowej instrukcja SAG wymagałaby tylko dwóch rejestrów (co ma znaczenie w przypadku niektórych implementacji architektury klasy RISC);
- łatwiej rozwinąć ją do obsługi wartości złożonych z dwóch słów (ang. *doubleword*);
- bardziej wydajnie realizuje permutację części słów.

Punkt trzeci z powyższych rozważań szerzej omówiono w pozycji [LSY]. Instrukcja SAG pozwala na realizację uogólnionej permutacji wartości złożonych z dwóch słów za pomocą dwóch wywołań instrukcji SAG, kilku dodatkowych instrukcji z podstawowego zestawu RISC oraz dwóch pełnych permutacji pojedynczych słów. Instrukcja *bitgather*

pozwala na realizację tego celu za pomocą *trzech* permutacji pojedynczych słów plus pięć instrukcji z podstawowego zestawu RISC. Nie uwzględniamy operacji przygotowawczych permutacji, generujących nowe wartości uzależnione wyłącznie od permutacji. Odkrycie tych sposobów pozostawiamy jako ćwiczenie dla Czytelnika.

Zatrzymajmy się jeszcze chwile przy punkcie 4. Chcąc na przykład dokonać permutacji czterech bajtów w słowie za pomocą instrukcji *bitgather*, musimy wykonać sześć instrukcji, dokładnie tak samo, jak ma to miejsce w przypadku uogólnionej operacji permutacji za pomocą instrukcji *bitgather*. Natomiast za pomocą instrukcji SAG można tego dokonać w dwóch instrukcjach, zamiast pięciu wymaganych przez uogólnioną wersję algorytmu permutacji wykorzystującego instrukcję SAG. Przyrost wydajności będzie miał miejsce również w przypadku, gdy rozmiar części słów nie jest potęgą liczby dwa. Wtedy liczba wymaganych kroków wynosi $\lceil \log_2 n \rceil$, gdzie n określa liczbę części słowa, nie licząc części słowa niebiorących udziału w permutacji.

W pozycji [LSY] omówiono dokładniej instrukcje SAG oraz *bitgather* (nazywane tam, odpowiednio, GRP oraz PPERM), jak również inne możliwe instrukcje permutacji oparte na sieciach lub też tablicach przeglądowych.

7.6. Zmiana kolejności oraz transformacje oparte na indeksach

Wiele prostych modyfikacji kolejności bitów w słowie procesora ma związek z jeszcze prostszymi operacjami na współrzędnych, czyli indeksach, bitów [GLS1]. Związki te odnoszą się do zmiany kolejności elementów w każdej jednowymiarowej macierzy, pod warunkiem, że liczba elementów w tablicy stanowi potęgę całkowitą dwójki. W zastosowaniach programistycznych najczęściej elementy macierzy są słowami procesora lub większymi elementami.

W ramach przykładu zewnętrzne tasowanie zupełne elementów w macierzy A o rozmiarze 8 z wynikiem w macierzy B można zapisać za pomocą następujących przesunięć:

$$A_0 \to B_0; \quad A_1 \to B_2; \quad A_2 \to B_4; \quad A_3 \to B_6;$$

$$A_4 \to B_1; \quad A_5 \to B_3; \quad A_6 \to B_5; \quad A_7 \to B_7.$$

Każdy indeks macierzy B odpowiada indeksowi macierzy A przesuniętemu cyklicznie w lewo za pomocą obrotu 3-bitowego. Operacja *zewnętrznego odtasowania zupełnego* (ang. *outer perfect unshuffle*) jest oczywiście realizowana za pomocą odpowiedniej operacji przesunięcia cyklicznego w prawo. Niektóre z podobnych zależności prezentujemy w tabeli 7.1. W opisie n określa liczbę elementów w macierzy, „lsb" określa najmniej znaczący bit, natomiast przesunięcia cykliczne indeksów są realizowane na $\log_2 n$ bitach.

Tabela 7.1. *Zmiana kolejności oraz transformacje oparte na indeksach*

Zmiana kolejności bitów	Transformacja w oparciu o indeksy	
	Indeks w macierzy, numeracja big-endian	**Numeracja little-endian**
Odwrócenie kolejności	Dopełnienie	Dopełnienie
Odwrócenie bitowe lub uogólnione odwrócenie kolejności (strona 122)	Różnica symetryczna ze stałą	Różnica symetryczna ze stałą
Przesunięcie cykliczne w lewo o k pozycji	Odjęcie k (mod n)	Dodanie k (mod n)
Przesunięcie cykliczne w prawo o k pozycji	Dodanie k (mod n)	Odjęcie k (mod n)
Zewnętrzne tasowanie zupełne	Przesunięcie cykliczne w lewo o jedną pozycję	Przesunięcie cykliczne w prawo o jedną pozycję
Zewnętrzne odtasowanie zupełne	Przesunięcie cykliczne w prawo o jedną pozycję	Przesunięcie cykliczne w lewo o jedną pozycję
Wewnętrzne tasowanie zupełne	Przesunięcie cykliczne w lewo o jedną pozycję, następnie dopełnienie „lsb”	Dopełnienie „lsb”, następnie obrót w prawo o jedną pozycję
Wewnętrzne odtasowanie zupełne	Dopełnienie „lsb”, następnie obrót w prawo o jedną pozycję	Przesunięcie cykliczne w lewo o jedną pozycję, następnie dopełnienie „lsb”
Transpozycja macierzy 8×8 bitów przechowywanej w 64-bitowym słowie	Przesunięcie cykliczne (w lewo lub w prawo) o trzy pozycje	Przesunięcie cykliczne (w lewo lub w prawo) o trzy pozycje
Odkodowanie FFT	Odwrócenie kolejności bitów	Odwrócenie kolejności bitów

Rozdział 8.

Mnożenie

8.1. Mnożenie czynników wieloelementowych

Mnożenie tego typu można wykonać za pomocą tradycyjnej metody, znanej ze szkoły podstawowej. Jednakże zamiast budowania tablicy iloczynów częściowych bardziej efektywne będzie sumowanie z dotychczasowym wynikiem każdego kolejnego wiersza po jego wyliczeniu.

W przypadku, gdy pierwszym czynnikiem jest liczba zbudowana z m słów a drugim jest liczba zbudowana z n słów, wynik zajmie $m + n$ słów (lub mniej), niezależnie od tego, czy operację *mnożenia* wyliczamy ze znakiem, czy bez znaku.

W celu zastosowania algorytmu mnożenia znanego ze szkoły podstawowej każde z 32-bitowych słów musimy potraktować jako pojedynczą cyfrę. Zasada ta działa bardzo dobrze w przypadku instrukcji procesora generującej 64-bitowy iloczyn dwóch słów 32-bitowych. Niestety, nawet w przypadku, gdy instrukcja tego typu jest dostępna, najczęściej dostęp do niej z języków programowania wyższego poziomu nie jest możliwy. W rzeczywistości wiele współczesnych procesorów klasy RISC nie obsługuje tej instrukcji po części dlatego, że i tak nie jest ona bezpośrednio dostępna w większości języków programowania i z tego powodu byłaby wykorzystywana bardzo rzadko. Innym powodem pomijania implementacji tej instrukcji jest to, że wynik jej działania musi być zapisywany w dwóch rejestrach procesora.

Nasza procedura została zaprezentowana na listingu 8.1. Wykorzystuje ona półsłowa w charakterze „cyfr". Wynik zostaje zapisany w parametrze `w`, natomiast `u` oraz `v` są czynnikami iloczynu. Każdy z czynników jest tablicą półsłów, z których pierwszy element (`w[0]`, `u[0]`, `v[0]`) jest najmniej znaczącą cyfrą wyniku. Jest to kolejność typu najmniej znaczące wartości najpierw (ang. *little endian*). Parametry `m` oraz `n` określają liczbę słów w `u` oraz `v`.

Listing 8.1. *Mnożenie wieloelementowych liczb całkowitych ze znakiem*

```
void mulmns(unsigned short w[], unsigned short u[],
   unsigned short v[], int m, int n) {

   unsigned int k, t, b;
   int i, j;

   for (i = 0; i < m; i++)
      w[i] = 0;

   for (j = 0; j < n; j++) {
      k = 0;
      for (i = 0; i < m; i++) {
         t = u[i]*v[j] + w[i + j] + k;
         w[i + j] = t;          // (tzn., t & 0xFFFF).
         k = t >> 16;
      }
      w[j + m] = k;
   }

   // w w[] znajduje się teraz iloczyn bez znaku
   // wynik ten należy poprawić za pomocą odjęcia wartości
   // v*2**16m dla u < 0, oraz u*2**16n dla v < 0.

   if ((short)u[m - 1] < 0) {
      b = 0;                    // inicjacja pożyczki
      for (j = 0; j < n; j++) {
         t = w[j + m] - v[j] - b;
         w[j + m] = t;
         b = t >> 31;
      }
   }
   if ((short)v[n - 1] < 0) {
      b = 0;
      for (i = 0; i < m; i++) {
         t = w[i + n] - u[i] - b;
         w[i + n] = t;
         b = t >> 31;
      }
   }
   return;
}
```

Poniżej prezentujemy schemat poszczególnych zmiennych wchodzących do operacji. Pomiędzy wartościami m i n nie ma żadnej określonej relacji i nie ma znaczenia, która z nich jest większa.

$$
\begin{array}{cccccccc}
 & & u_{m-1} & u_{m-2} & \cdots & \cdots & u_1 & u_0 \\
 & & & \times & v_{n-1} & \cdots & v_1 & v_0 \\
\hline
w_{m+n-1} & w_{m+n-2} & & \cdots & \cdots & \cdots & w_1 & w_0
\end{array}
$$

Procedura ta jest skonstruowana zgodnie z algorytmem M z pozycji [Knu2, poz. 4.3.1], lecz jest napisana w języku C i została zmodyfikowana w celu umożliwienia mnożenia ze znakiem. Warto zwrócić uwagę na to, że przypisanie zmiennej `t` w pierwszej części kodu na listingu 8.1 nie powoduje przepełnienia, ponieważ maksymalną wartością, która może zostać przypisana zmiennej `t` jest $(2^{16} - 1)^2 + 2(2^{16} - 1) = 2^{31} - 1$.

Mnożenie wartości zbudowanych z wielu słów najprościej jest wykonać dla argumentów bez znaku. Tak naprawdę gdyby pominąć instrukcje dokonujące korekty (wiersze od trzywierszowego komentarza do instrukcji `return`), kod z listingu 8.1 wykona operację *mnożenia bez znaku*. Istnieją trzy sposoby rozwinięcia wersji wykonującej mnożenie bez znaku do wersji wykonującej mnożenie ze znakiem:

1. Obliczenie wartości bezwzględnej każdego z czynników, wykonanie mnożenia bez znaku a następnie wykonanie dopełnienia wyniku w przypadku, gdy czynniki są przeciwnych znaków.
2. Wykonanie mnożenia bez znaków na poziomie składowych elementów czynników za wyjątkiem najbardziej znaczących półsłów, dla których w zależności od znaków należy wykonać mnożenie liczby ze znakiem przez liczbę bez znaku lub mnożenie dwóch liczb ze znakiem.
3. Wykonanie mnożenia bez znaku a następnie skorygowanie wyniku w odpowiedni sposób.

Pierwsza z metod wymaga przepisania $m + n$ liczby półsłów w celu wyliczenia ich wartości bezwzględnej. W przypadku, gdy jeden z czynników jest liczba dodatnią a drugi ujemną algorytm ten wymaga dodatkowo przepisania do $\max(m, n) + m + n$ półsłów w celu wyliczenia dopełnienia bitowego wyniku. Można oczywiście tego uniknąć, modyfikując parametry wejściowe funkcji (przekazywane przez adres), lecz w niektórych programach może to być niedopuszczalne. Można również zaalokować pamięć tymczasową dla potrzeb przetwarzania, lub zmodyfikować wartości parametrów a na końcu przywrócić ich pierwotne wartości. Żadna z tych możliwości nie wydaje się być dobrym rozwiązaniem.

Druga z metod wymaga zastosowania trzech typów operacji elementarnych (mnożenie dwóch liczb bez znaku, mnożenie dwóch liczb ze znakiem oraz mnożenie liczby ze znakiem przez liczbę bez znaku) i wymaga rozwinięcia w lewo zerami lub jedynkami częściowych iloczynów, co może spowodować dłuższe wykonywanie się każdego z iloczynów częściowych i wydłużenia całkowitego czasu wykonania.

Wybieramy trzecią metodę. Aby sprawdzić jej działanie, załóżmy wartości u oraz v, będące czynnikami mnożenia, są liczbami całkowitymi ze znakiem. Mają one długość, odpowiednio, M oraz N bitów. Instrukcje w pierwszej części listingu 8.1 błędnie interpretują te liczby jako wartości bez znaku $u + 2^M u_{M-1}$, gdzie u_{M-1} jest znakiem bitu liczby u. To znaczy u_{M-1} ma wartość 1 jeżeli u jest liczbą ujemną, w przeciwnym wypadku u_{M-1} ma wartość 0. Podobnie wartość v zostanie błędnie zinterpretowana jako $v + 2^N u_{N-1}$.

Program wyliczy iloczyn tych wartości ze znakiem, to znaczy wykona następujące obliczenia:

$$(u + 2^M u_{M-1})(v + 2^N v_{N-1}) = uv + 2^M u_{M-1} v + 2^N v_{N-1} u + 2^{M+N} u_{M-1} v_{N-1}$$

Aby uzyskać pożądany wynik (uv), musimy odjąć od wyniku bez znaku wartość $2^M u_{M-1} v + 2^N v_{N-1} u$. Nie ma potrzeby odejmowania elementu $2^{M+N} u_{M-1} v_{N-1}$, ponieważ wiemy, iż wynik można zapisać w $M + N$ bitach, nie ma więc potrzeby wyliczania wyniku z bitów na pozycjach bardziej znaczących od $M + N - 1$. Omówione odejmowanie jest wykonywane przez instrukcje poniżej trzywierszowego komentarza na listingu 8.1. Wymagają one przepisania maksymalnie $m + n$ półsłów.

Być może kuszące wyda się zastosowanie w listingu 8.1 tablic słów w miejsce tablic półsłów. Program taki zadziałałby na maszynie typu *little-endian*, lecz nie działałby poprawnie na maszynie typu *big-endian*. Natomiast w przypadku ułożenia tablic w odwrotnej kolejności, dla `u[0]` będącego najbardziej znaczącym słowem czynnika, i odpowiedniej modyfikacji algorytmu, ten program zadziałałby poprawnie na maszynie typu *big-endian*, lecz nie działałby na maszynie *little-endian*.

8.2. Bardziej znacząca połowa 64-bitowego iloczynu

W niniejszym podrozdziale rozpatrzymy problem wyliczania bardziej znaczącego słowa iloczynu dwóch liczb 32-bitowych. Jest to funkcja *mulhs* (ang. *multiply high signed*) oraz *mulhu* (ang. *multiply high unsigned*) z podstawowego zestawu instrukcji RISC.

Dla mnożenia bez znaku algorytm z listingu 8.1 działa poprawnie. Aby to sprawdzić, można przepisać go dla specjalnego przypadku $m = n = 2$, rozwinąć pętle, dokonać odpowiednich uproszczeń i zastąpić parametry funkcji 32 bitowymi liczbami bez znaku.

W przypadku mnożenia ze znakiem nie ma potrzeby wykorzystywania korekty z drugiej części listingu 8.1. Można pominąć tę część algorytmu, jeśli zwróci się uwagę na to, czy wykorzystywane wartości bezwzględne są liczbami ze znakiem czy bez znaku (wykorzystanie wartości ze znakiem spowoduje propagację bitów znaku przy przesunięciach w prawo). Algorytm wynikowy został zaprezentowany na listingu 8.2. Dla wersji bez znaku wystarczy zastąpić wszelkie deklaracje zmiennych typu `int` zmiennymi typu `unsigned`.

Listing 8.2. *Bardziej znaczące słowo wyniku mnożenia liczb 32-bitowych*

```
int mulhs(int u, int v) {
   unsigned u0, v0, w0;
   int u1, v1, w1, w2, t;

   u0 = u & 0xFFFF;  u1 = u >> 16;
   v0 = v & 0xFFFF;  v1 = v >> 16;
```

```
    w0 = u0*v0;
    t  = u1*v0 + (w0 >> 16);
    w1 = t & 0xFFFF;
    w2 = t >> 16;
    w1 = u0*v1 + w1;
    return u1*v1 + w2 + (w1 >> 16);
}
```

Algorytm ten wykorzystuje 16 instrukcji z podstawowego zestawu RISC, z których cztery są operacjami mnożenia.

8.3. Konwersje między bardziej znaczącą połową 64-bitowego iloczynu ze znakiem i bez znaku

Załóżmy, że maszyna jest w stanie wyliczyć bardziej znaczącą połowę 64-bitowego iloczynu dwóch 32-bitowych liczb całkowitych *bez znaku*, lecz chcemy wykonać takie mnożenie odpowiednich liczb *ze znakiem*. Moglibyśmy wykorzystać procedurę z listingu 8.2, lecz wymaga ona zastosowania mnożeń. Mamy jednak możliwość zastosowania procedury zaprezentowanej w [BGN], która jest znacznie wydajniejsza.

Analiza prowadząca do tego algorytmu jest specjalnym przypadkiem analizy wykorzystanej w celu przekształcenia algorytmu M Knutha z procedury bez znaku na obsługującą czynniki ze znakiem. Załóżmy, że x oraz y są liczbami całkowitymi, które chcemy przemnożyć. Maszyna zinterpretuje x jako liczbę *bez znaku* o wartości $x + 2^{32}x_{31}$, gdzie x_{31} jest bitem znaku liczby x (to znaczy najbardziej znaczącym bitem liczby). Podobnie wartość y zostanie zinterpretowana jako $y + 2^{32}y_{31}$.

Pomimo że poszukujemy bardziej znaczących 32 bitów wyniku mnożenia, maszyna dokona następujących obliczeń:

$$(x + 2^{32}x_{31})(y + 2^{32}y_{31}) = xy + 2^{32}(x_{31}y + y_{31}x) + 2^{64}x_{31}y_{31}$$

W celu uzyskania pożądanego wyniku, musimy od wyniku odjąć $2^{32}(x_{31}y + y_{31}x) + 2^{64}x_{31}y_{31}$. Ponieważ wiemy, że wynik można zapisać na 64 bitach, obliczenia możemy wykonać modulo 2^{64}. Oznacza to, że możemy z powodzeniem zignorować ostatni element i bardziej znaczącą połowę wyniku możemy uzyskać za pomocą następujących operacji:

$$\begin{aligned}
&p \leftarrow \text{mulhu}(x, y) && \text{// instrukcja } \textit{multiply high unsigned} \\
&t_1 \leftarrow (x \overset{s}{>>} 31) \,\&\, y && \text{// } t_1 = x_{31}y \\
&t_2 \leftarrow (y \overset{s}{>>} 31) \,\&\, x && \text{// } t_2 = y_{31}x \\
&p \leftarrow p - t_1 - t_2 && \text{// } p = \text{poszukiwany wynik}
\end{aligned} \tag{1}$$

Wersja bez znaku za pomocą wersji ze znakiem

Odwrotna transformacja przebiega bardzo łatwo. Algorytm wynikowy będzie podobny do przedstawionego w formule (1), za wyjątkiem pierwszej instrukcji zastąpionej w tym przypadku instrukcją *mulhs* (ang. *multiply high signed*), natomiast ostatnia operacja powinna zostać zastąpiona operacją $p \leftarrow p + t_1 + t_2$.

8.4. Mnożenie przez stałe

To, że operację *mnożenia* przez stałą można zastąpić sekwencją *przesunięć w lewo* oraz *dodawań*, wydaje się być prawdą oczywistą. Na przykład w celu przemnożenia wartości x przez 13 (dwójkowo 1101) można wykonać następujące operacje:

$$t_1 \leftarrow x << 2$$

$$t_2 \leftarrow x << 3$$

$$r \leftarrow t_1 + t_2 + x$$

Wynik operacji zapisywany jest w r.

W niniejszym podrozdziale *przesunięcia w lewo* są zapisywane w postaci mnożenia przez potęgi dwójki, zatem powyższa procedura może zostać zapisana następujący sposób: $r \leftarrow 8x + 4x + x$, która powinna wykonać się w ośmiu instrukcjach na maszynach klasy RISC oraz większości innych procesorów.

W niniejszym podrozdziale mamy zamiar wykazać, że w tym zagadnieniu mieści się więcej wartości, niż może się początkowo wydawać. Przede wszystkim należy wziąć pod uwagę inne zagadnienia, oprócz liczby wymaganych *przesunięć w lewo* oraz *dodawań*. W celu wyjaśnienia tego twierdzenia, poniżej przedstawiamy procedury mnożenia przez 45 (dwójkowo 101101):

$t \leftarrow 4x$	$t_1 \leftarrow 4x$
$r \leftarrow x + t$	$t_2 \leftarrow 8x$
$t \leftarrow 2t$	$t_3 \leftarrow 32x$
$r \leftarrow r + t$	$r \leftarrow t_1 + x$
$t \leftarrow 4t$	$t_3 \leftarrow t_3 + t_2$
$r \leftarrow r + t$	$r \leftarrow r + t_3$

Procedura w lewej kolumnie wykorzystuje zmienną t, która przechowuje wartość liczby x, przesuniętą w lewo o odpowiednią liczbę miejsc i określoną pozycją jedynki w liczbie, przez którą wykonujemy mnożenie. Każde *przesunięcie* jest wykonywane na wyniku poprzedniego *przesunięcia*. Procedura ta ma kilka zalet:

- wymaga wykorzystania tylko jednego rejestru roboczego oprócz rejestru wejściowego x oraz wyjściowego r;
- za wyjątkiem pierwszych dwóch, wykorzystuje wyłącznie instrukcje dwuadresowe;
- wartości przesunięć są stosunkowo niewielkie.

Procedura ta zachowuje wymienione własności dla dowolnych wartości stałej, przez którą wykonujemy mnożenie.

Procedura w prawej kolumnie najpierw wykonuje wszystkie *przesunięcia* z oryginalną wartością x jako operandem. Zaletą tej wersji jest wyższy poziom równoległości. Na maszynie z odpowiednią liczbą układów umożliwiających równoległe wykonanie niezależnych instrukcji, przedstawiony przykład wykona się w trzech cyklach, podczas gdy procedura w lewej kolumnie na tej samej maszynie wymaga czterech cykli zegara.

Określenie minimalnej liczby operacji niezbędnych w celu wykonania mnożenia przez stałą nie jest problemem trywialnym. Przez „operację" rozumiemy instrukcję z podstawowego zestawu operacji obsługiwanych przez procesor. Dla dalszej części rozważań przyjmijmy, iż zestaw ten składa się z *przesunięcia w lewo*, *dodawania* oraz *negacji*. Zakładamy, że instrukcje występują w formacie trójadresowym. Problem nie byłby wcale prostszy, gdybyśmy ograniczyli się do dodawania (liczbę dodajemy do samej siebie, następnie tę sumę dodajemy do samej siebie itd., co jest sposobem wykonania przesunięcia w lewo) lub gdyby uzupełnić zestaw dopuszczalnych instrukcji o instrukcję dostępną w procesorach HP PA-RISC, wykonującą *przesunięcie* i *dodawanie* (ang. *shift-and-add*). Instrukcja ta wykonuje przesunięcie pierwszego argumentu o jedno, dwa lub trzy miejsca w lewo, następnie dodaje do wyniku wartość drugiego rejestru a wynik zapisuje w trzecim. Dzięki temu w pojedynczej, z założenia szybkiej instrukcji można wykonać mnożenie przez 3, 5 lub 9. Zakładamy również, że potrzebne nam są jedynie 32 mniej znaczące bity wyniku.

Pierwsze usprawnienie mechanizmu rozkładu stałej mogłoby polegać na wykonaniu *odejmowania* zamiast serii dodawań wszędzie tam, gdzie stała składa się z grupy trzech lub większej liczby jedynek występujących obok siebie. Na przykład w celu przemnożenia przez 28 (dwójkowo 11100) możemy wykonać następujące operacje: $32x - 4x$ (trzy instrukcje) zamiast $16x + 8x + 4x$ (pięć instrukcji). Na maszynach wykorzystujących uzupełnienie dwójkowe wynik będzie prawidłowy nawet w przypadku, gdy wynik pośredni $32x$ spowoduje przepełnienie, podczas gdy wynik końcowy przepełnienia nie wywołuje.

W celu wykonania mnożenia przez stałą m za pomocą podstawowego schematu rozłożenia operacji na składowe (z użyciem *przesunięć* i *dodawań*), operacja ta wymaga następującej liczby instrukcji:

$$2\mathrm{pop}(m) - 1 - \delta$$

W powyższym wzorze $\delta = 1$ jeśli m kończy się jedynką (jest liczba nieparzystą), natomiast w przeciwnym wypadku $\delta = 0$. W przypadku, gdy dopuszczamy wykorzystanie *odejmowań*, operacja ta wymaga liczby instrukcji określonej wzorem:

$$4g(m) + 2s(m) - 1 - \delta$$

W powyższym wzorze $g(m)$ określa liczbę grup jedynek o długości 2 lub więcej w m. Natomiast $s(m)$ określa liczbę „samotnych" jedynek w m, δ ma takie samo znaczenie, co w poprzednim wzorze.

Dla grup o wielkości 2 wybór metody nie ma znaczenia.

Kolejne usprawnienie polega na specjalnym traktowaniu grup jedynek, które są odseparowane pojedynczym bitem zerowym. Na przykład rozpatrzmy wartość 55 (dwójkowo 110111). Metoda wykorzystująca grupy jedynek spowoduje rozbicie mnożenia przez tę liczbę na następujące składowe: $(64x - 16x) + (8x - x)$, co wymaga sześciu instrukcji. Rozbicie na następujące operacje: $64x - 8x - x$ wymaga zaledwie czterech instrukcji. Podobnie możemy rozbić stałą 110111011 na $512x - 64x - 4x - x$ (sześć instrukcji).

Powyższe formuły pozwalają na określenie górnego ograniczenia liczby instrukcji wymaganych do obliczenia mnożenia przez stałą m. Ściślejsze ograniczenie można określić na podstawie rozmiaru m w bitach, to znaczy $n = \lfloor \log_2 m \rfloor + 1$.

twierdzenie

Mnożenie zmiennej x przez n-bitową stałą m, $m \geq 1$, można rozłożyć na co najwyżej n instrukcji złożonych z *dodawania, odejmowania* oraz *przesunięcia w lewo* o określoną wartość.

Dowód (przez indukcję względem n). Mnożenie przez 1 można wykonać w liczbie instrukcji równej 0, więc dla $n = 1$ twierdzenie jest spełnione. Dla $n > 1$, jeśli m kończy się bitem zerowym, mnożenie przez m można wykonać za pomocą mnożenia przez liczbę składającą się z lewostronnych $n - 1$ bitów z m (to znaczy przez $m/2$) w $n - 1$ instrukcjach, po których należy wykonać instrukcję *przesunięcia w lewo* o jedną pozycję. Operacja ta wymaga w sumie wykonania n instrukcji.

W przypadku, gdy m kończy się sekwencją bitów 01, mx można wyliczyć przez pomnożenie x przez liczbę składającą się z lewostronnych $n - 2$ bitów stałej m w $n - 2$ instrukcjach, po których należy wykonać *przesunięcie w lewo* o dwa miejsca i *dodanie* wartości x. Operacje te łącznie wymagają zastosowania liczby instrukcji równej n.

W przypadku, gdy m kończy się sekwencją bitów 11, należy wziąć pod uwagę przypadki 0011, 0111, 1011 oraz 1111. Zmiennej t przypiszemy wynik mnożenia x przez lewostronne $n - 4$ bity stałej m. W przypadku, gdy m kończy się sekwencją bitów 0011, $mx = 16t + 2x + x$, które wymaga $(n - 4) + 4 = n$ instrukcji. Jeśli m kończy się sekwencją 0111, $mx = 16t + 8x - x$, co wymaga n instrukcji. Jeśli m kończy się sekwencją 1111, $mx = 16t + 16x - x$, co wymaga n instrukcji. Ostatni przypadek to m kończące się sekwencją 1011.

Można łatwo wykazać, że mx można wyliczyć za pomocą n instrukcji w przypadku, gdy m kończy się sekwencją 001011, 011011 lub 111011. Ostatni przypadek to m kończące się sekwencją 101011.

Powyższe rozumowanie można kontynuować, za każdym razem „ostatni przypadek" zawsze będzie miał postać 101010...10101011. W końcu zostanie osiągnięty rozmiar m i jedynym pozostającym przypadkiem będzie 101010...10101011. Ta n-bitowa liczba składa się z $n/2 + 1$ jedynek. Z poprzedniej obserwacji wynika, że x można pomnożyć przez m za pomocą $2(n/2 + 1) - 2 = n$ instrukcji.

Z tego wynika, że za pomocą opisanej metody każde mnożenie przez stałą można zrealizować na 32-bitowej maszynie za pomocą najwyżej 32 instrukcji. Można z łatwością wykazać, że dla każdej parzystej liczby n, mnożenie przez każdą n-bitową liczbę typu 101010...101011 wymaga n instrukcji, dla nieparzystych liczb n, mnożenie przez każdą n-bitowa liczbę typu 1010101...010110 wymaga również n instrukcji, zatem ograniczenie jest ścisłe.

Opisana metodologia nie jest bardzo trudna do ręcznego rozpisania, lub do zaimplementowania w algorytmie, który można wykorzystać w kompilatorze. Lecz taki algorytm nie zawsze prowadzi do optymalnego kodu, możliwe są dalsze jego usprawnienia. Usprawnienia tego typu mogą wynikać z rozbicia liczby m na części składowe lub wykorzystanie odpowiedniej wartości bezpośredniej w trakcie wyliczania iloczynu mx. Dla przykładu $m = 45$ (101101) z opisanej metody wynika procedura wymagająca sześciu instrukcji. Rozbicie liczby 45 na czynniki 5 i 9 pozwoli na opracowanie procedury wykorzystującej cztery instrukcje:

$$t \leftarrow 4x + x$$

$$r \leftarrow 8t + t$$

Rozbijanie stałej na czynniki można łączyć z metodami dodawania. Na przykład mnożenie x przez stałą $m = 106$ (1101010) wymaga zastosowania siedmiu instrukcji za pomocą opisanej metody, lecz rozpisanie stałej na $7 \cdot 15 + 1$ pozwoli na skonstruowanie bardziej optymalnej procedury (pięć instrukcji).

Problem oszacowania maksymalnej liczby instrukcji za pomocą rozbijania stałej na czynniki jest, zdaniem Autora, nierozstrzygnięty. Dla dużych wartości n liczba ta może być niższa od ograniczenia n oszacowanego powyżej. Na przykład dla m = 0xAAAAAAAB metoda bez rozbijania na czynniki wymaga 32 instrukcji, po zapisaniu tej liczby jako $2 \cdot 5 \cdot 17 \cdot 257 \cdot 65537 + 1$ mnożenie można wykonać w dziesięciu instrukcjach. Dziesięć instrukcji nie jest najprawdopodobniej typowym oszacowaniem dla dużych liczb m. Możliwość rozbicia stałej na czynniki dotyczy prostych układów przeplatających się zer i jedynek.

Powyższe rozważania powinny przybliżyć poziom komplikacji związanych nawet z pozornie prostymi problemami. Knuth [Knu2, podrozdz. 4.6.3] omawia problem ściśle związany z omówionym w niniejszym rozdziale. Polega on na wyliczeniu wyniku a^m za pomocą minimalnej liczby mnożeń. Jest to problem podobny do wyliczania wyniku mnożenia przez m wyłącznie za pomocą instrukcji *dodawania*. Algorytm wyliczający mx przeznaczony do zastosowania w kompilatorach został opisany w artykule [Bern].

Rozdział 9.

Dzielenie całkowitoliczbowe

9.1. Warunki wstępne

W niniejszym oraz kolejnym rozdziale zostaną przedstawione techniki i algorytmy służące do wyliczania wyniku dzielenia liczb całkowitych. W formułach matematycznych dzielenie tradycyjne oznaczamy x / y, dzielenie całkowitoliczbowe ze znakiem oznaczamy $x \div y$ (z obcięciem w kierunku 0), natomiast dzielenie całkowite bez znaku oznaczamy $x \overset{u}{\div} y$. W kodzie w języku C operacja `x / y` oznacza oczywiście dzielenie komputerowe, w którym znak wyniku jest uzależniony od typów operandów — w przypadku, gdy obydwa lub jeden z operandów jest liczbą ze znakiem, wynik również będzie ze znakiem, natomiast w przypadku, gdy obydwa operandy są liczbami całkowitymi bez znaku, wynik dzielenie będzie również liczbą całkowitą bez znaku.

Dzielenie jest procesem skomplikowanym a algorytmy implementujące tę operację z reguły nie są przykładem elegancji. Sposób, w jaki zdefiniowane jest dzielenie całkowitoliczbowe jest zwykle uzależniony od subiektywnego osądu. Najczęściej dzielenie całkowitoliczbowe jest definiowane jako wynik zwykłego dzielenia z obcięciem w kierunku zera. Istnieją również dwa inne rozwiązania. Wszystkie z nich prezentujemy poniżej:

		obcięcie	**moduł**	**podłoga**
`7 ÷ 3`	=	`2 rem 1`	`2 rem 1`	`2 rem 1`
`(-7) ÷ 3`	=	`-2 rem -1`	`-3 rem 2`	`-3 rem 2`
`7 ÷ (-3)`	=	`-2 rem 1`	`-2 rem 1`	`-3 rem 2`
`(-7) ÷ (-3)`	=	`2 rem -1`	`3 rem 2`	`2 rem -1`

Relacja *dzielna* = *iloraz* * *dzielnik* + *reszta* jest zachowana dla wszystkich trzech metod. Dzielenie z zastosowaniem modułu definiuje się w ten sposób, że reszta nie może

być liczbą ujemną[1]. Dzielenie z zastosowaniem funkcji *podłoga* definiuje się w ten sposób, że iloraz jest wynikiem funkcji *podłoga* na wyniku zwykłego dzielenia. W przypadku dzielenia liczb całkowitych bez znaku metody wykorzystujące funkcję *moduł* oraz funkcję *podłoga* dają identyczne wyniki. Istnieje również czwarty sposób, najrzadziej stosowany, który polega na zaokrągleniu ilorazu do najbliższej liczby całkowitej.

Jedną z zalet stosowania funkcji *moduł* i *podłoga* jest fakt, że upraszczają one większość z algorytmów. Na przykład dzielenie przez 2^n można zastąpić *przesunięciem w prawo ze znakiem* o *n* pozycji. Resztę z dzielenia uzyskamy wykonując operację *koniunkcji* logicznej liczb *x* oraz $2^n - 1$. Metody wykorzystujące funkcje *moduł* oraz *podłoga* dają zwykle oczekiwane wyniki. Rozważmy program rysujący wykres funkcji zwracającej wartości całkowite z przedziału od *imin* do *imax*. Wartościami skrajnymi będą, odpowiednio, $(imin \div 10) * 10$ oraz $((imax + 9) \div 10) * 10$, o ile zostaną zastosowane dzielenia typu *moduł* lub *podłoga*. W przypadku, gdy stosowane jest zwykłe dzielenie, należy wyliczyć następujące wyrażenia:

```
if (imin >= 0) gmin = (imin/10)*10;
else           gmin = ((imin - 9)/10)*10;
if (imax >= 0) gmax = ((imax + 9)/10)*10;
else           gmax = (imax/10)*10;
```

Iloraz jest bardziej użyteczny, gdy zostanie wyliczony za pomocą dzielenia typu *podłoga* niż w przypadku wersji wykorzystującej obcięcie, ponadto zakładamy, iż bardziej pożądana będzie nieujemna reszta z dzielenia niż reszta, która potencjalnie może być ujemna.

Wybór pomiędzy dzieleniem wykorzystującym funkcję *moduł* a wykorzystującym funkcję *podłoga* nie jest łatwy, ponieważ różnica pomiędzy nimi występuje wyłącznie wtedy, gdy dzielnik jest liczbą ujemną, co nie jest powszechną sytuacją. Preferencje dotyczące języków wyższego programowania nie na wiele się tu przydadzą, ponieważ prawie wszystkie te języki wykorzystują dzielenie obcinające w przypadku, gdy dzielna i dzielnik są liczbami ze znakiem. Niektóre języki zwracają wyniki zmiennoprzecinkowe lub wymierne. Natomiast w przypadku reszty z dzielenia jest spore zamieszanie. W języku Fortran 90 funkcja `MOD` zwraca resztę z dzielenia obcinającego, natomiast `MODULO` zwraca resztę z dzielenia typu *podłoga* (która to wartość może być ujemna). Podobnie w językach Common Lisp oraz ADA, funkcja REM zwraca resztę z dzielenia obcinającego, `MOD` zwraca resztę z dzielenia typu *podłoga*. W języku PL/I funkcja `MOD` zwraca zawsze nieujemne wartości (jest to reszta z dzielenia typu *moduł*). W Pascalu `A mod B` jest zdefiniowane tylko dla `B > 0`, zatem wartość tego wyrażenia jest nieujemna (co w praktyce oznacza resztę z dzielenia typu *moduł* lub typu *podłoga*).

---

[1] Nie jest to zdefiniowane w tak jednoznaczny sposób. W publikacjach Knutha operator „mod" [Knu1] określa resztę z dzielenia za pomocą funkcji *podłoga*, zatem może mieć wartości ujemne (lub 0) w przypadku, gdy dzielnik jest liczbą ujemną. Niektóre języki programowania stosują operację „mod" w znaczeniu reszty z dzielenia całkowitoliczbowego wykorzystującego zaokrąglenie w kierunku zera. W matematyce jednak operacja „modulus" jest wykorzystywana w celu podnoszenia do potęgi liczb zespolonych (nieujemnych), a w teorii kongruencji (ang. *congruence*) modulus jest z reguły uznawany za wartość dodatnią.

Niestety nie zmienimy świata, choćbyśmy nawet wiedzieli, w jaki sposób chcielibyśmy to zrobić[2], zatem w dalszej części naszych rozważań wszędzie, gdzie występuje zapis typu $x \div y$ stosujemy powszechnie wykorzystywane dzielenie obcinające.

Istnieje pewna interesująca właściwość dzielenia obcinającego:

$$(-n) \div d = n \div (-d) = -(n \div d), \text{ dla } d \neq 0$$

W przypadku stosowania tej zależności należy jednak zachować ostrożność, ponieważ w przypadku, gdy n lub d jest największą możliwą liczbą ujemną, $-n$ lub $-d$ nie można zapisać za pomocą liczby 32-bitowej. Operacja $(-2^{31}) \div (-1)$ powoduje przepełnienie (wyniku nie można wyrazić za pomocą 32-bitowej liczby ze znakiem w notacji uzupełnienia dwójkowego) i na większości komputerów wynik jest niezdefiniowany lub operacja jest zatrzymana.

Dzielenie liczb ze znakiem (obcinające) posiada następujące właściwości w odniesieniu do zwykłego dzielenia:

$$n \div d = \begin{cases} \lfloor n/d \rfloor, & \text{dla} \quad d \neq 0, nd \geq 0, \\ \lceil n/d \rceil, & \text{dla} \quad d \neq 0, nd < 0. \end{cases} \tag{1}$$

W przypadku dzielenia całkowitoliczbowego bez znaku, to znaczy takiego, w którym zarówno dzielna n, jak i dzielnik d są liczbami bez znaku, sytuacja spełnia warunek górnej części formuły (1).

W dalszej dyskusji będziemy wykorzystywać następujące arytmetyczne własności elementarne, których dowodu nie będziemy przeprowadzać. Dalsze ciekawe szczegóły dotyczące funkcji *podłoga* oraz *sufit* można znaleźć w pozycjach [Knu1] oraz [GKP].

twierdzenie

TWIERDZENIE D1

Dla rzeczywistej liczby x oraz całkowitej liczby k zachodzą następujące własności:

$\lfloor x \rfloor = -\lceil -x \rceil$	$\lceil x \rceil = -\lfloor -x \rfloor$
$x - 1 < \lfloor x \rfloor \leq x$	$x \leq \lceil x \rceil < x + 1$
$\lfloor x \rfloor \leq x < \lfloor x \rfloor + 1$	$\lceil x \rceil - 1 < x \leq \lceil x \rceil$
$x \geq k \Leftrightarrow \lfloor x \rfloor \geq k$	$x \leq k \Leftrightarrow \lceil x \rceil \leq k$
$x > k \Rightarrow \lfloor x \rfloor \geq k$	$x < k \Rightarrow \lceil x \rceil \leq k$
$x \leq k \Rightarrow \lfloor x \rfloor \leq k \Rightarrow x < k + 1$	$x \geq k \Rightarrow \lceil x \rceil \geq k \Rightarrow x > k - 1$
$x < k \Leftrightarrow \lfloor x \rfloor < k$	$x > k \Leftrightarrow \lceil x \rceil > k$

[2] Są jednak tacy, którzy próbują. IBM w swoim języku PL.8 wykorzystuje dzielenie typu *moduł*, natomiast maszyna MMIX autorstwa Knutha wykorzystuje w instrukcji dzielenia wersję typu *podłoga* [MMIX].

TWIERDZENIE D2

twierdzenie

Dla liczb całkowitych n oraz d, jeśli $d > 0$ zachodzą następujące własności:

$$\left\lfloor \frac{n}{d} \right\rfloor = \left\lceil \frac{n-d+1}{d} \right\rceil \text{ oraz } \left\lceil \frac{n}{d} \right\rceil = \left\lfloor \frac{n+d-1}{d} \right\rfloor.$$

Jeśli $d < 0$, zachodzą następujące własności:

$$\left\lfloor \frac{n}{d} \right\rfloor = \left\lceil \frac{n-d-1}{d} \right\rceil \text{ oraz } \left\lceil \frac{n}{d} \right\rceil = \left\lfloor \frac{n+d+1}{d} \right\rfloor.$$

TWIERDZENIE D3

twierdzenie

Dla rzeczywistej liczby x oraz całkowitej liczby $d \neq 0$ zachodzą następujące własności:

$$\lfloor \lfloor x \rfloor / d \rfloor = \lfloor x / d \rfloor \text{ oraz } \lceil \lceil x \rceil / d \rceil = \lceil x / d \rceil.$$

WNIOSEK.

Dla rzeczywistych liczb a, b, $b \neq 0$ oraz całkowitej liczby $d \neq 0$ zachodzą następujące własności:

$$\left\lfloor \left\lfloor \frac{a}{b} \right\rfloor / d \right\rfloor = \left\lfloor \frac{a}{bd} \right\rfloor \text{ oraz } \left\lceil \left\lceil \frac{a}{b} \right\rceil / d \right\rceil = \left\lceil \frac{a}{bd} \right\rceil$$

TWIERDZENIE D4

twierdzenie

Dla całkowitych liczb n, d, $d \neq 0$ oraz rzeczywistej liczby x zachodzą następujące własności:

$$\left\lfloor \frac{n}{d} + x \right\rfloor = \left\lfloor \frac{n}{d} \right\rfloor, \text{ jeżeli } 0 \le x < \left|\frac{1}{d}\right| \text{ oraz } \left\lceil \frac{n}{d} + x \right\rceil = \left\lceil \frac{n}{d} \right\rceil \text{ jeżeli } -\left|\frac{1}{d}\right| < x \le 0$$

W poniższych twierdzeniach rem(n, d) oznacza resztę z dzielenia n / d. Dla $d < 0$ rem(n, $-d$) = rem(n, d), tak samo jak ma to miejsce w przypadku dzielenia obcinającego oraz dzielenia z wykorzystaniem funkcji *modul*. Nie wykonuje się rem(n, d) dla $n < 0$. Dlatego w naszych zastosowaniach reszta z dzielenia jest zawsze liczbą nieujemną.

TWIERDZENIE D5

twierdzenie

Dla $n \ge 0$, $d \neq 0$ zachodzą następujące własności:

$$\mathrm{rem}(2n, d) = \begin{cases} 2\mathrm{rem}(n, d) & \text{lub} \\ 2\mathrm{rem}(n, d) - |d| \end{cases}$$

oraz:

$$\mathrm{rem}(2n+1, d) = \begin{cases} 2\mathrm{rem}(n, d) + 1 & \text{lub} \\ 2\mathrm{rem}(n, d) - |d| + 1 \end{cases}$$

Wybierana jest ta wartość, która jest większa lub równa zeru oraz mniejsza od $|d|$.

TWIERDZENIE D6

Dla $n \geq 0$, $d \neq 0$ zachodzi następująca własność:

$$\mathrm{rem}(2n, 2d) = 2\mathrm{rem}(n, d)$$

Twierdzenia D5 oraz D6 można łatwo dowieść za pomocą definicji reszty z dzielenia, to znaczy dowieść, że dla liczby całkowitej q jest spełniona następująca własność:

$$n = qd + \mathrm{rem}(n, d), \text{ dla } 0 \leq \mathrm{rem}(n, d) < |d|$$

założywszy, że $n \geq 0$ oraz $d \neq 0$ (n oraz d mogą nie być liczbami całkowitymi, lecz dla celów naszych dalszych rozważań będziemy te twierdzenia stosować dla liczb całkowitych).

9.2. Dzielenie wartości wieloelementowych

Podobnie w przypadku mnożenia wartości wieloelementowych, również dzielenie wartości zbudowanych z wielu słów może zostać wykonane za pomocą metody znanej wszystkim ze szkoły podstawowej. Szczegóły takiej implementacji są jednak zadziwiająco skomplikowane. Listing 9.1 przedstawia algorytm D. Knutha [Knu2 pdrozdz. 4.3.1], zakodowany w języku C. Algorytm wykorzystuje formę dzielenia $32 \overset{u}{\div} 16 \Rightarrow 32$ (w rzeczywistości iloraz wykorzystywanych operacji *dzielenia* ma co najwyżej 17 bitów).

Listing 9.1. *Dzielenie całkowite bez znaku wartości wieloelementowych*

```
int divmnu(unsigned short q[], unsigned short r[],
     const unsigned short u[], const unsigned short v[],
     int m, int n) {

   const unsigned b = 65536; // podstawa liczby (16 bitów).
   unsigned short *un, *vn;  // znormalizowana postać u, v.
   unsigned qhat;            // oszacowana cyfra ilorazu.
   unsigned rhat;            // reszta.
   unsigned p;               // iloczyn dwóch cyfr
   int s, i, j, t, k;

   if (m < n || n <= 0 || v[n-1] == 0)
      return 1;              // wyjście, gdy parametr jest nieprawidłowy

   if (n == 1) {                        // obsługa przypadku
      k = 0;                            // wykorzystania
      for (j = m - 1; j >= 0; j--) {    // dzielnej składającej się
         q[j] = (k*b + u[j])/v[0];      // z pojedynczej cyfry
         k = (k*b + u[j]) - q[j]*v[0];
      }
      if (r != NULL) r[0] = k;
      return 0;
   }
```

```
// normalizacja odbywa się za pomocą przesunięcia v w lewo,
   // tak aby najbardziej znaczący bit miał wartość 1
   // a następnie przesuwamy u w lewo o tę samą liczbę pozycji.
   // Być może konieczne będzie dopisanie bardziej znaczącej cyfry
   // do dzielnej. Wykonujemy tę operację bezwarunkowo.

   s = nlz(v[n-1]) - 16;         // 0 <= s <= 16.
   vn = (unsigned short *)alloca(2*n);
   for (i = n - 1; i > 0; i--)
      vn[i] = (v[i] << s) | (v[i-1] >> 16-s);
   vn[0] = v[0] << s;

   un = (unsigned short *)alloca(2*(m + 1));
   un[m] = u[m-1] >> 16-s;
   for (i = m - 1; i > 0; i--)
      un[i] = (u[i] << s) | (u[i-1] >> 16-s);
   un[0] = u[0] << s;

   for (j = m - n; j >= 0; j--) {        // pętla główna
      // wyliczamy oszacowanie qhat dla q[j].
      qhat = (un[j+n]*b + un[j+n-1])/vn[n-1];
      rhat = (un[j+n]*b + un[j+n-1]) - qhat*vn[n-1];
again:
      if (qhat >= b || qhat*vn[n-2] > b*rhat + un[j+n-2])
      { qhat = qhat - 1;
        rhat = rhat + vn[n-1];
        if (rhat < b) goto again;
      }

      // mnożenie i odejmowanie
      k = 0;
      for (i = 0; i < n; i++) {
         p = qhat*vn[i];
         t = un[i+j] - k - (p & 0xFFFF);
         un[i+j] = t;
         k = (p >> 16) - (t >> 16);
      }
      t = un[j+n] - k;
      un[j+n] = t;

      q[j] = qhat;               // zapisujemy cyfrę ilorazu
      if (t < 0) {               // jeśli odjęliśmy za dużo
         q[j] = q[j] - 1;        // dodajemy nadmiar z powrotem
         k = 0;
         for (i = 0; i < n; i++) {
            t = un[i+j] + vn[i] + k;
            un[i+j] = t;
            k = t >> 16;
         }
         un[j+n] = un[j+n] + k;
      }
   } // koniec pętli j
   // jeśli procedura wywołująca potrzebuje wartość reszty
   // denormalizujemy ją i przekazujemy
   if (r != NULL) {
      for (i = 0; i < n; i++)
         r[i] = (un[i] >> s) | (un[i+1] << 16-s);
   }
   return 0;
}
```

Algorytm przetwarza parametry wejściowe i wyjściowe w krokach co pół słowa. Oczywiście lepiej byłoby przetwarzać jednocześnie całe słowa, lecz w tym celu byłaby konieczna instrukcja wykonująca *dzielenie* typu $64 \overset{u}{\div} 32 \Rightarrow 32$. Zakładamy, że maszyna nie ma możliwości wykonania takiej instrukcji, bądź dostęp do tej instrukcji jest utrudniony w wykorzystywanym języku wysokiego poziomu. Pomimo iż z reguły zakładamy udostępnianie przez maszynę dzielenia klasy $32 \overset{u}{\div} 32 \Rightarrow 32$, w tym przypadku wystarczy $32 \overset{u}{\div} 16 \Rightarrow 16$.

W tej implementacji algorytmu Knutha podstawą `b` jest 65536. Szczegółowe wyjaśnienia algorytmu można znaleźć w pozycji [Knu2].

Dzielna `u` i dzielnik `v` są zapisane w formacie *little-endian*, to znaczy `u[0]` oraz `v[0]` są najmniej znaczącymi cyframi liczb (choć kod będzie działał poprawnie zarówno na maszynach *little-endian*, jak i *big-endian*). Parametry `m` oraz `n` określają, odpowiednio, liczby półsłów w `u` oraz `v` (Knuth określa `m` jako długość ilorazu). Alokacja miejsca dla ilorazu `q` oraz opcjonalnie dla reszty `r` jest zadaniem procedury wywołującej. Obszar pamięci zaalokowany na potrzeby zapisu ilorazu musi mieć rozmiar co najmniej `m - n + 1` półsłów a dla reszty `n` półsłów. Alternatywnie, można przekazać wartość `NULL` zamiast adresu reszty, w ten sposób sygnalizując funkcji, że reszta nie jest potrzebna.

Algorytm wymaga, aby najbardziej znacząca cyfra dzielnika `v[n - 1]` miała wartość różną od zera. W ten sposób można uprościć czynności normalizacyjne oraz upewnić się, że procedura wywołująca zabezpieczyła odpowiedni obszar pamięci do zapisu ilorazu. Kod sprawdza, czy `v[n - 1]` nie ma wartości 0, jak również sprawdza, czy `n` ≥ 1 oraz `m` ≥ `n`. Jeśli przynajmniej jedno z tych założeń nie jest spełnione, funkcja kończy się błędem (zwraca wartość 1).

Po wykonaniu wspomnianych sprawdzeń kod wykonuje dzielenie dla prostego przypadku, gdy dzielnik ma długość jednej cyfry. Przypadek ten nie został wyodrębniony w celu przyśpieszenia, lecz dlatego, że w pozostałej części algorytmu zakłada się, że dzielnik ma długość 2 cyfrlub więcej.

Gdy dzielnik ma długość 2 słów lub więcej, algorytm normalizuje dzielnik przesuwając w lewo, tak aby najbardziej znaczący bit miał wartość 1. Dzielna jest przesuwana w lewo o tę samą liczbę pozycji, zatem iloraz nie ulega zmianie z powodu tych przesunięć. Jak to zostało objaśnione przez Knutha, kroki te są niezbędne w celu oszacowania każdej z cyfr ilorazu z dużą dokładnością. Rozmiar przesunięcia jest określany za pomocą funkcji nlz(x).

W krokach normalizacyjnych jest alokowane nowe miejsce dla znormalizowanej dzielnej i dzielnika. Przyczyną takiej operacji jest przede wszystkim zabezpieczenie argumentów wejściowych przed modyfikacją, ponieważ z punktu widzenia procedury wywołującej niepożądane jest modyfikowanie tych wartości, lub jest wręcz niemożliwe, ponieważ są odczytywane z pamięci przeznaczonej tylko do odczytu. Ponadto może zaistnieć potrzeba ustawienia bardziej znaczącej cyfry w dzielnej. Funkcja `alloca` języka C jest idealna w celu alokacji przestrzeni pamięci dla dzielnej i dzielnika. Funkcja ta jest z reguły zaimplementowana z dużą wydajnością, wykorzystując zazwyczaj zaledwie dwie

lub trzy instrukcje wbudowanego kodu (ang. *in-line*) i nie są potrzebne żadne instrukcje zwalniające tę zaalokowaną pamięć. Obszar jest alokowany na stosie programu w taki sposób, że po powrocie z funkcji jest zwalniany automatycznie.

W pętli głównej cyfry ilorazu są wyliczane po jednej na każdą iterację pętli a dzielna jest odpowiednio redukowana aż do momentu, gdy stanie się resztą z dzielenia. Oszacowanie `qhat` każdej z cyfr ilorazu, po korekcie w pętli opatrzonej etykietą `again` będzie dokładną cyfrą, lub za dużą o 1.

Kolejne kroki mnożą `qhat` przez dzielnik i odejmują iloczyn od aktualnej reszty, tak samo, jak robimy to podczas obliczeń dzielenia metodą pisemną. Jeśli reszta jest liczbą ujemną, konieczne jest zmniejszenie cyfry ilorazu o 1 i ponowne obliczenie iloczynu oraz odjęcie go od reszty lub po prostu wprowadzenie korekty wartości reszty przez dodanie do niej wartości dzielnika. Potrzeba takiej korekty występuje co najwyżej raz, ponieważ cyfra ilorazu była dokładna lub za duża o 1.

Na koniec reszta jest zwracana procedurze wywołującej (jeśli adres obszaru przeznaczonego na nią nie jest równy `NULL`). Resztę z dzielenia należy przed zwróceniem przesunąć w prawo o liczbę pozycji równą `s`, czyli o przesunięcie normalizujące.

Kroki dodające wartość, o jaką przeszacowano iloraz są wykonywane raczej rzadko. Aby to zaobserwować, należy przyjrzeć się obliczeniom każdej oszacowanej cyfry ilorazu `qhat`, które obejmują dzielenie najbardziej znaczących dwóch cyfr aktualnej reszty przez najbardziej znaczącą cyfrę dzielnika. Kroki w pętli `again` korygujące wartość `qhat` do poprawnej wartości polegają na przeprowadzeniu dzielenia *trzech* najbardziej znaczących cyfr aktualnej reszty przez *dwie* najbardziej znaczące cyfry dzielnika (pomijamy dowód, w celu doświadczalnego sprawdzenia można wykonać odpowiednie przykłady, przyjmując podstawę `b` = 10). Zwróćmy uwagę na to, że dzielnik jest większy lub równy `b/2` (z powodu normalizacji) a dzielna jest mniejsza lub równa `b` * dzielnik (ponieważ reszta jest zawsze mniejsza od dzielnika).

Jak dokładne jest oszacowanie cyfry ilorazu na podstawie tyko trzech cyfr dzielnej i dwóch cyfr dzielnika? Normalizacja została wykonana, zatem oszacowanie to jest stosunkowo dokładne. Aby pojąć intuicyjnie tę własność (nie będzie to formalny dowód), przyjmijmy, że w ten sam sposób chcemy oszacować wynik u/v na liczbach o podstawie dziesiętnej. Można pokazać, że oszacowanie jest bardzo zbliżone lub dokładne. Najgorszy przypadek występuje wtedy, gdy obcięcie dzielnika do dwóch cyfr zmniejsza dzielnik o tyle o ile to możliwe pod względem błędu względnego, natomiast obcięcie dzielnej do trzech cyfr zwiększa błąd o najmniejszą możliwą wartość (czyli 0) a dzielna jest największą możliwą liczbą. Sytuacja ta występuje w przypadku 49900...0/5099...9, dla oszacowania 499/50 = 9,98. Prawdziwy wynik wynosi około 499/51 ≈ 9,7843. Różnica wynosi 0,1957, co pokazuje, że oszacowana cyfra ilorazu i prawdziwa cyfra ilorazu (które stanowią wynik operacji *podłoga* na tych obliczeniach) będą różnić się najwyżej o 1, a sytuacja ta wystąpi w około 20% przypadków (założywszy, że cyfry ilorazu będą miały rozkład jednostajny). To z kolei oznacza, że kroki korygujące (polegające na dodaniu dzielnika do reszty) będą wykonywane w około 20% przypadków.

Kontynuując tę luźną analizę dla ogólnej podstawy b otrzymamy różnicę oszacowanej i prawdziwej cyfry ilorazu, wynoszącą co najwyżej $2/b$. Dla $b = 65536$ ponownie stwierdzimy, że różnica pomiędzy oszacowaną i prawdziwą cyfrą ilorazu może wynieść co najwyżej 1 a prawdopodobieństwo wystąpienia takiego błędu wynosi $2/65536 \approx 0{,}00003$. Stąd można stwierdzić, że operacje korygujące wynik będzie trzeba wykonywać w 0,003% przypadków.

Przykładem sytuacji wymagającej dokonania korekty wyniku jest dziesiątkowe dzielenie 4500/501. Podobnym przykład dla podstawy 65536 może być 0x7FFF8000 0000 0000/ 0x8000 0000 0001.

Nie będziemy oszacowywać czasu wykonania całego programu, lecz zauważmy, że dla dużych liczb m i n czas wykonania jest zdeterminowany przez pętlę *mnożącą* i *odejmującą*. Dobry kompilator skompiluje tę pętlę do około 16 instrukcji podstawowego zestawu RISC, z których jedna to *mnożenie*. Pętla `for j` wykonuje się n razy, natomiast pętla wykonująca mnożenia i dodawania wykona się $m - n + 1$ razy, co daje czas wykonania tej części programu wynoszący $(15 + mul)n(m - n + 1)$ cykli, gdzie mul oznacza czas potrzebny na wykonanie mnożenia dwóch zmiennych 16-bitowych. Program również wymaga wykonania $m - n + 1$ instrukcji *dzielenia* oraz jedną instrukcję *wyznaczającą liczbę zer wiodących*.

Dzielenie wartości wieloelementowych ze znakiem

Nie będziemy przedstawiać algorytmu specjalnie dla dzielenia wartości wieloelementowych ze znakiem, lecz wskażemy kilka miejsc w algorytmie dla liczb bez znaku, których modyfikacja pozwoli na zaadaptowanie go do naszych potrzeb. W tym celu trzeba wykonać poniższe operacje.

1. Jeśli dzielna jest liczbą ujemną, należy zmienić jej znak i podobnie uczynić z dzielnikiem.
2. Przekształcić dzielną i dzielnik do postaci bez znaku.
3. Zastosować algorytm dla wartości wieloelementowych bez znaku.
4. Przekształcić iloraz i resztę z dzielenia do postaci ze znakiem.
5. Zanegować iloraz, jeśli dzielna i dzielnik były liczbami przeciwnych znaków.
6. Zanegować resztę, jeśli dzielna miała znak ujemny.

Powyższe kroki czasem wymagają dodania lub usunięcia najbardziej znaczącej cyfry. Na przykład załóżmy, że liczby zapisujemy w systemie o podstawie 256 (jeden bajt na cyfrę) i że w reprezentacji ze znakiem najbardziej znaczący bit sekwencji cyfr jest bitem znaku. Jest to sytuacja bardzo przypominająca reprezentację liczby w uzupełnieniu dwójkowym. W tym przypadku dzielnik równy 255 o reprezentacji ze znakiem równej 0x00FF musi zostać znormalizowany w drugim kroku do 0xFF. Podobnie iloraz w kroku 3. zaczynający się od bitu o wartości 1 musi zostać przekształcony do postaci zawierającej na początku bajt o wartości 0, aby został prawidłowo zinterpretowany jako liczba ze znakiem.

9.3. Krótkie dzielenie bez znaku za pomocą dzielenia ze znakiem

Przez *krótkie dzielenie* (ang. *short division*) rozumiemy dzielenie pojedynczego słowa przez drugie (na przykład $32 \div 32 \Rightarrow 32$). Jest to forma dzielenia realizowana w języku C oraz innych językach przez operator /, gdy operandami są liczby całkowite. Język C obsługuje dzielenie całkowite ze znakiem i bez znaku, lecz niektóre komputery udostępniają wyłącznie instrukcje *dzielenia ze znakiem*. W jaki sposób można zaimplementować dzielenie bez znaku na takiej maszynie? Nie istnieje żaden złoty środek, lecz przedstawimy kilka możliwych rozwiązań.

Wykorzystanie długiego dzielenia ze znakiem

Nawet w przypadku, gdy maszyna udostępnia operację *długiego dzielenia ze znakiem* (ang. *signed long division*) $64 \div 32 \Rightarrow 32$, zrealizowanie za jej pomocą krótkiego dzielenia bez znaku nie jest tak proste jak mogłoby się wydawać. W kompilatorze XLC dla komputera IBM RS/6000 operację tę zaimplementowano w następujący sposób:

$$
\begin{aligned}
&\text{if } n \overset{u}{<} d \text{ then } q \leftarrow 0\\
&\text{else if } d = 1 \text{ then } q \leftarrow n\\
&\text{else if } d \le 1 \text{ then } q \leftarrow 1\\
&\text{else } q \leftarrow (0 \,\|\, n) \div d
\end{aligned}
$$

Trzeci wiersz stanowi w rzeczywistości test sprawdzający, czy $d \overset{u}{\ge} 2^{31}$. Jeśli d jest algebraicznie mniejsze lub równe 1, to ponieważ nie jest równe 1 (z drugiego wiersza), musi być algebraicznie mniejsze lub równe 0. Nie przejmujemy się przypadkiem, gdy $d = 0$. Zatem jeśli test pokazany w trzecim wierszu da wynik prawdziwy, liczba d ma ustawiony bit znaku, to znaczy $d \overset{u}{\ge} 2^{31}$. Z pierwszego wiersza wiemy, że $n \overset{u}{\ge} d$ i ponieważ n nie może być większe od $2^{32} - 1$, $n \overset{u}{\div} d = 1$.

Czwarty wiersz ma na celu zbudowanie liczby całkowitej podwójnej długości, składającej się z 32 bitów o wartości 0, po których następują 32 bity wartości n, a następnie podzielenie tej wartości przez d. Sprawdzenie, czy $d = 1$ (drugi wiersz) jest niezbędne w celu upewnienia się, że dana operacja dzielenia nie spowoduje przepełnienia (stałoby się tak, gdyby $n \overset{u}{\ge} 2^{31}$ i w tej sytuacji iloraz byłby niezdefiniowany).

Po połączeniu porównań z drugiego i trzeciego wiersza[3] powyższy kod może zostać wykonany w 11 instrukcjach, z których trzy to instrukcje *rozgałęzień*. Jeśli istnieje potrzeba wykonania *dzielenia* dla $d = 0$, to aby wywołać przerwanie przepełnienia, możemy zmodyfikować trzeci wiersz do postaci „if $d < 0$, then $q \leftarrow 1$". W ten sposób cały algorytm wykorzysta 12 instrukcji na maszynie RS/6000.

[3] Na maszynie RS/6000 instrukcja porównania zwraca zestaw bitów stanu, określających sytuację *mniejszy*, *większy* lub *równy*.

Modyfikacja powyższego kodu w taki sposób, aby w najbardziej prawdopodobnych przypadkach ($2 \overset{u}{\le} d \overset{u}{<} 2^{31}$) nie wykonywać tak wielu testów (rozpoczynając od $d \le 1$...) jest stosunkowo prosta, lecz odbywa się kosztem niewielkiego zwiększenia rozmiaru kodu.

Wykorzystanie krótkiego dzielenia ze znakiem

Jeśli nie jest dostępna instrukcja *długiego dzielenia ze znakiem*, lecz jest dostępna instrukcja *krótkiego dzielenia ze znakiem*, wtedy operację $n \overset{u}{\div} d$ można zaimplementować redukując problem do przypadku n, $d < 2^{31}$. Jeśli $d \overset{u}{\ge} 2^{31}$, $n \overset{u}{\div} d$ może mieć wartość **0** lub **1**, zatem ten przypadek możemy pominąć łatwością obsłużyć. Dzięki temu możemy zredukować dzielną wykorzystując fakt, iż wyrażenie $((n \overset{u}{\div} 2) \div d) \times 2$ przybliża $n \overset{u}{\div} d$ z błędem wynoszącym 0 lub 1. Stąd możemy wyprowadzić następującą procedurę:

1. if $d < 0$ then if $n \overset{u}{<} d$ then $q \leftarrow 0$
2. else $q \leftarrow 1$
3. else do
4. $q \leftarrow ((n \overset{u}{\div} 2) \div d) \times 2$
5. $r \leftarrow n - qd$
6. if $r \overset{u}{\ge} d$ then $q \leftarrow q + 1$
7. end

Sprawdzenie $d < 0$ w wierszu pierwszym tak naprawdę służy do sprawdzenia, czy $d \overset{u}{\ge} 2^{31}$. Jeśli $d \overset{u}{\ge} 2^{31}$, wtedy największym możliwym ilorazem będzie $(2^{32} - 1) \div 2^{31} = 1$, zatem dwa pierwsze wiersze wyliczają prawidłowy iloraz.

Wiersz 4. reprezentuje operacje *przesunięcia w prawo* o 1 bez znaku, *dzielenie* i *przesunięcie* w lewo o 1. Oczywiście $n \overset{u}{\div} 2 \overset{u}{<} 2^{31}$ i w tym miejscu również zachodzi własność $d \overset{u}{<} 2^{31}$, zatem wartości te możemy wykorzystać jako argumenty instrukcji *dzielenia całkowitego ze znakiem* (w przypadku, gdy $d = 0$, zostanie zgłoszone przepełnienie).

Oszacowanie wyliczane w 4. wierszu wynosi:

$$q = \lfloor \lfloor n/2 \rfloor / d \rfloor \cdot 2 = \lfloor n/(2d) \rfloor \cdot 2 = \frac{n - \mathrm{rem}(n,2d)}{d}.$$

Wykorzystaliśmy tutaj rozwinięcie twierdzenia D3. Wiersz 5. wylicza resztę z dzielenia odpowiednią dla oszacowanego ilorazu. Reszta ta wynosi:

$$r = n - \frac{n - \mathrm{rem}(n,2d)}{d} d = \mathrm{rem}(n,2d).$$

Stąd $0 < r < 2d$. Jeśli $r < d$, to q jest prawidłowym ilorazem. Jeżeli $r \ge d$, to prawidłowy iloraz uzyskamy dodając 1 do q (program musi zastosować operację *porównania bez znaku*, ponieważ istnieje możliwość, iż $r > 2^{31}$).

Przenosząc *ładowanie wartości bezpośredniej* **0** do q przed porównanie $n \overset{u}{<} d$ i przez zakodowanie przypisania $q \leftarrow 1$ w wierszu 2. jako rozgałęzienie do przypisania $q \leftarrow q + 1$ w wierszu 6., powyższy algorytm można wykonać na większości procesorów w 14 instrukcjach, z których 4 są rozgałęzieniami. Rozbudowanie algorytmu o generowanie reszty z dzielenia jest również oczywiste: do wiersza pierwszego dopisujemy $r \leftarrow n$, do wiersza drugiego dopisujemy $r \leftarrow n - d$ a w wierszu 6. na końcu dopisujemy $r \leftarrow r - d$ (lub poświęcając jedną instrukcję *mnożenia* po prostu wyliczamy resztę z wyrażenia $r \leftarrow n - qd$ na końcu całej sekwencji).

Alternatywnie w miejsce pierwszego i drugiego wiersza można wstawić:

$$\text{if } n \overset{u}{<} d \text{ then } q \leftarrow 0$$

$$\text{else if } d < 0 \text{ then } q \leftarrow 1$$

Dzięki temu algorytm będzie bardziej zwarty, mieszcząc się w 13 instrukcjach, z których trzy są rozgałęzieniami. Wadą tego podejścia jest większa liczba instrukcji w najpowszechniejszych przypadkach (niewielkie liczby przy $n > d$).

Z wykorzystaniem wyrażeń predykatów program ten można zapisać następująco:

1. if $d < 0$ then $q \leftarrow (n \overset{u}{\geq} d)$
2. else do
3. $\quad q \leftarrow ((n \overset{u}{\div} 2) \div d) \times 2$
4. $\quad r \leftarrow n - qd$
5. $\quad q \leftarrow q + (r \overset{u}{\geq} d)$
6. end

Algorytm ten pozwala na uniknięcie dwóch rozgałęzień, pod warunkiem, że procesor udostępnia wyliczenie predykatów bez wykorzystania rozgałęzień. Na procesorach Compaq Alpha predykaty te można wyliczyć za pomocą pojedynczej instrukcji (`CMPULE`), na procesorach MIPS potrzeba dwóch instrukcji (`SLTU` oraz `XORI`). Na większości komputerów każdy z tych predykatów można zrealizować w czterech instrukcjach (w trzech w przypadku, gdy dostępne są instrukcje z pełnego zestawu RISC). W tym celu stosujemy wyrażenie wyliczające predykat $x \overset{u}{\leq} y$, przestawione na stronie 39 oraz nieco upraszczamy całą procedurę, ponieważ w pierwszym wierszu wiadomo, że $d_{31} = 1$, natomiast w wierszu 5. wiadomo, że $d_{31} = 0$. Zatem możemy przeprowadzić następujące modyfikacje przedstawionej procedury:

$$n \overset{u}{\geq} d = (n \,\&\, \neg(n - d)) \overset{u}{>>} 31 \quad \text{w wierszu 1 oraz}$$

$$r \overset{u}{\geq} d = (r \mid \neg(r - d)) \overset{u}{>>} 31 \quad \text{w wierszu 5}$$

Kod niewykorzystujący rozgałęzień można uzyskać zmieniając wartość dzielnej na **0** w przypadku, gdy $d \overset{u}{\geq} 2^{31}$. Następnie dzielnik możemy wykorzystać w instrukcji *dzielenia ze znakiem*, ponieważ nawet w przypadku, gdyby błędnie został zinterpretowany

jako liczba ujemna, wynikiem będzie 0, co różni się od prawidłowego wyniku o 1. Nadal będziemy obsługiwać przypadki dużych wartości dzielnych za pomocą przesunięcia w prawo o jedną pozycję przed operacją *dzielenia* a następnie przesunięcia ilorazu w lewo o jedną pozycję. W wyniku tych zmian uzyskamy następujący kod, wykonujący się w dziesięciu instrukcjach:

1. $t \leftarrow d \overset{s}{>>} 31$
2. $n' \leftarrow n \,\&\, \neg t$
3. $q \leftarrow ((n' \overset{u}{\div} 2) \div d) \times 2$
4. $r \leftarrow n - qd$
5. $q \leftarrow q + (r \overset{u}{\geq} d)$

9.4. Długie dzielenie bez znaku

Przez „długie dzielenie" rozumiemy dzielenie wartości podwójnego słowa przez pojedyncze słowo. Na maszynie 32-bitowej jest to dzielenie typu $64 \div 32 \Rightarrow 32$. W przypadku wystąpienia przepełnienia (wliczając w to dzielenie przez 0) wynik takiego dzielenia jest nieokreślony.

Niektóre maszyny 32-bitowe udostępniają instrukcję *długiego dzielenia bez znaku*. Pełne możliwości tej instrukcji mają jednak niewielkie zastosowanie, ponieważ w większości języków programowania jest dostępne wyłącznie dzielenie typu $32 \overset{u}{\div} 32 \Rightarrow 32$. Z tego powodu projektant procesora mógłby zdecydować, aby udostępnić wyłącznie dzielenie typu $32 \overset{u}{\div} 32$ i chciałby zapewne znać przybliżony koszt czasowy wykonania procedury implementującej brakującą operację. W dalszej części przedstawiamy dwa algorytmy implementujące tę brakującą funkcję.

Algorytmy sprzętowego przesuwania i odejmowania

W pierwszej próbie spróbujmy zasymulować działanie rozwiązania sprzętowego. Powszechnie wykorzystywane są dwa algorytmy, zwane dzieleniem *przywracającym* oraz *nieprzywracającym* (ang. *restoring* oraz *nonrestoring*) [H&P, poz. A-2; EL]. Są to w gruncie rzeczy algorytmy *przesuwania* i *odejmowania* (ang. *shift and subtract*). W wersji przywracającej, którą wykorzystujemy poniżej, krok przywracający wartość polega na dodaniu dzielnika w przypadku, gdy odejmowanie daje wynik ujemny. Wartości x, y i z są przechowywane w 32-bitowych rejestrach. Początkowo dzielna o podwójnej długości ma wartość $x \,||\, y$, natomiast dzielnik jest ma wartość z. Potrzebny nam będzie również jednobitowy rejestr c, przechowujący informację o przepełnieniu odejmowania.

do $i \leftarrow 1$ to 32

$c \| x \| y \leftarrow 2(x \| y)$ // przesunięcie w lewo o jeden

$c \| x \leftarrow (c \| x) - (\mathbf{0b0} \| z)$ // odjęcie (33 bity)

$y_0 \leftarrow \neg c$ // ustawienie bitu ilorazu

if c then $c \| x \leftarrow (c \| x) + (\mathbf{0b0} \| z)$ // odtworzenie

end

Po zakończeniu działania procedury iloraz znajdzie się w rejestrze y, natomiast reszta z dzielenia w rejestrze x.

Algorytm ten w przypadku przepełnienia nie zwróci przydatnej wartości. W przypadku dzielenia $x \| y$ przez 0 uzyskana wartość ilorazu będzie uzupełnieniem jedynkowym liczby x, natomiast reszta z dzielenia będzie wynosić y. W szczególności $0 \overset{u}{\div} 0 \Rightarrow 2^{32}-1$, reszta 0. Inne przypadki przepełnienia są trudne do scharakteryzowania.

Dla dzielników różnych od zera użyteczne może okazać się skorygowanie ilorazu modulo 2^{32} oraz zwrócenie poprawnej reszty. Jedynym jednak sposobem realizacji tego wydaje się reprezentacja liczby $c \| x \| y$, która będzie miała ponad 97 bitów długości i wykonanie pętli 64 razy. Jest to dzielenie typu $64 \overset{u}{\div} 32 \Rightarrow 64$. Odejmowania nadal pozostaną operacjami 33-bitowymi, lecz wymagania dotyczące sprzętu oraz czas wykonania powodują, że modyfikacja taka wydaje się być mało praktyczna.

Algorytm ten jest trudny do zaimplementowania czysto programowo, ponieważ większość maszyn nie udostępnia 33-bitowych rejestrów, w których można zapisać wynik wyrażenia $c \| x$. Listing 9.2 prezentuje jednak algorytm wykorzystujący operacje *przesunięcia* i *odejmowania*, będący zbliżonym odpowiednikiem sprzętowej implementacji tej operacji.

Listing 9.2. *Długie dzielenie bez znaku za pomocą algorytmu przesunięć i odejmowań*

```
unsigned divluh(unsigned x, unsigned y, unsigned z) {
   // dzielenie wartości (x || y) przez z
   int i;
   unsigned t;

   for (i = 1; i <= 32; i++) {
      t = (int)x >> 31;          // same jedynki, jeżeli x(31) = 1.
      x = (x << 1) | (y >> 31); // przesunięcie w lewo wartości x || y
      y = y << 1;                // o jeden bit
      if ((x | t) >= z) {
         x = x - z;
         y = y + 1;
      }
   }
   return y;                     // x zawiera resztę
}
```

Zmienna t jest wykorzystywana przez sprzęt w celu kontrolowania wyniku porównania. Chcemy wykonać 33-bitowe porównanie po przesunięciu wyniku wyrażenia x || y. Jeśli pierwszy bit w x jest równy 1 (przed przesunięciem), wtedy na pewno wartość 33-bitowa jest większa od dzielnika (32 bity). W tym przypadku x | t zawiera same jedynki, zatem porównanie daje prawidłowy wynik (**prawda**). Z drugiej strony, jeśli pierwszy bit jest równy 0, wtedy wystarczające jest porównanie 32-bitowe.

Kod algorytmu z listingu 9.2 wykonuje się w od 321 do 385 instrukcjach z podstawowego zestawu RISC, w zależności od tego, jak często porównania dają wynik prawdziwy. Jeśli maszyna udostępnia instrukcję *przesunięcie dwusłowne w lewo* (ang. *shift left double*), przesunięcie można wykonać w pojedynczej instrukcji, w miejsce prezentowanych czterech. Zredukuje to kod do przedziału od 225 do 289 instrukcji (przewidujemy dwie instrukcje na każdą iterację w ramach instrukcji kontrolnych pętli).

Algorytm na listingu 9.2 może zostać zastosowany do realizacji dzielenia typu $32 \overset{u}{\div} 32 \Rightarrow 32$, definiując x = 0. Jedyne uproszczenie, które pojawia się wraz z tą modyfikacją polega na tym, że można pominąć zmienną t, której zawsze będzie równa 0.

Poniżej przestawiamy nieprzywracające sprzętowe dzielenie całkowite bez znaku. Podstawowy pomysł polega na tym, że po odjęciu dzielnika od wartości 33-bitowej, oznaczonej $c \| x$ nie ma potrzeby ponownego dodawania do wyniku wartości z, jeśli wynik jest ujemny. Zamiast tego wystarczy w następnej iteracji zamiast odejmowania dodać odpowiednią wartość. Dzieje się tak, ponieważ dodanie wartości z (w celu skorygowania błędu odjęcia z w poprzedniej iteracji), przesunięcie w lewo i odjęcie z jest równoznaczne z dodaniem z $(2(u + z) - z = 2u + z)$. Zaleta dla rozwiązania sprzętowego polega na tym, że w każdej iteracji występuje tylko jedna operacja *dodawania* lub *odejmowania* a jednostka sumująca jest najwolniejszym układem w tej pętli[4]. Na końcu należy jeszcze dokonać korekty reszty z dzielenia, jeśli ma znak ujemny. Nie ma natomiast potrzeby korygowania ilorazu.

Dzielna jest wartością o rozmiarze dwusłowa $x \| y$, dzielnikiem jest z. Po zakończeniu procedury iloraz jest zapisany w rejestrze y, natomiast reszta z dzielenia w x.

```
c = 0
do i ← 1 to 32
  if c = 0 then do
    c || x || y ← 2(x || y)          // przesunięcie w lewo o jedną pozycję
    c || x ← (c || x) – (0b0 || z)   // odejmujemy dzielnik
end
```

[4] W rzeczywistości algorytm dzielenia przywracającego może pominąć krok przywrócenia wartości po prostu umieszczając wynik odejmowania w dodatkowym rejestrze i zapisując tę wartość do x wyłącznie w przypadku, gdy wynik odejmowania (33-bitowy) jest nieujemny. Jednak w niektórych implementacjach sposób ten wymaga zastosowania dodatkowego rejestru i prawdopodobnie więcej czasu.

```
    else do
      c || x || y ← 2(x || y)          // przesunięcie w lewo o jedną pozycję
      c || x ← (c || x) + (0b0 || z)   // dodajemy dzielnik
    end
    y0 ← ¬c                            // ustawiamy bit ilorazu
  end
  if c = 1 then x ← x + z              // modyfikujemy resztę, jeśli jest ujemna
```

Algorytm powyższy nie daje się w prosty sposób zaadoptować do wersji wykorzystującej 32-bity.

Minikomputer 801 (wczesna maszyna klasy RISC wyprodukowana przez IBM) posiadał instrukcję kroku dzielenia (ang. *divide step*), która wykonywała pojedynczy przebieg powyższej pętli. Wykorzystywała bit stanu przeniesienia procesora do przechowania wartości c oraz MQ (rejestr 32-bitowy) do przechowywania y. Do implementacji tej metody niezbędne są jednostki sumujące oraz odejmujące, obsługujące wartości 33-bitowe. Instrukcja kroku dzielenia w 801 była nieco bardziej skomplikowana od prezentowanego algorytmu, ponieważ wykonywała dzielenie ze znakiem i posiadała kontrolę przepełnienia. Wykorzystując tę instrukcję można napisać procedurę dzielenia, wykorzystującą 32 kolejne wywołania instrukcji kroku dzielenia wraz z niezbędnymi korektami w celu przywrócenia odpowiedniego znaku dzielnikowi i reszcie z dzielenia.

Wykorzystanie krótkiego dzielenia

Algorytm wykonujący dzielenie typu $64 \overset{u}{\div} 32 \Rightarrow 32$ można uzyskać z algorytmu dzielenia wieloelementowego, przedstawionego na listingu 9.1 na stronie 165, specjalizując go w ten sposób, żeby obsługiwał przypadek $m = 4$, $n = 2$. Niezbędne jest dokonanie też kilku dodatkowych zmian. Parametry muszą być pełnymi słowami przekazanymi przez wartość a nie macierzami półsłów. Sytuacja przepełnienia ma inne znaczenie, występuje wtedy, gdy iloraz nie mieści się w pojedynczym słowie. Możliwe jest zastosowanie kilku uproszczeń tej procedury. Można wykazać, że oszacowanie `qhat` jest zawsze dokładne, jeśli dzielnik składa się wyłącznie z dwóch cyfr o długości półsłowa. Oznacza to, że można pominąć kroki przywracające wartość niepotrzebnie odjętą. Dodatkowe drobne uproszczenia są możliwe po rozwinięciu pętli z listingu 9.1.

Wynik tych modyfikacji przedstawiamy na listingu 9.3. Dzielna składa się z `u1` oraz `u0`, gdzie `u1` zawiera bardziej znaczące słowo. Dzielnik jest przekazywany w parametrze `v`. Iloraz jest zwracany w wyniku funkcji. Jeśli procedura wywołująca przekaże parametr `r` będący wskaźnikiem (nie `NULL`), funkcja zwróci również resztę z dzielenia, zapisując ją w słowie wskazywanym przez `r`.

Listing 9.3. *Dzielenie wartości typu long, bez znaku (ang. divide long unsigned) za pomocą dzielenia pełnych słów*

```
unsigned divlu(unsigned u1, unsigned u0, unsigned v,
               unsigned *r) {
   const unsigned b = 65536; // podstawa liczby (16-bitowa).
   unsigned un1, un0,        // znormalizowane cyfry dzielnej
            vn1, vn0,        // cyfry dzielnika
            q1, q0,          // cyfry ilorazu
            un32, un21, un10,// pary cyfr dzielnej
            rhat;            // reszta
   int s;                    // rozmiar przesunięcia dla normalizacji

   if (u1 >= v) {            // gdy nastąpiło przepełnienie ustawiamy resztę
      if (r != NULL)         // na wartość niemożliwą do uzyskania,
         *r = 0xFFFFFFFF;    // oraz iloraz ustawiamy na największą
      return 0xFFFFFFFF;}    // możliwą wartość

   s = nlz(v);               // 0 <= s <= 31.
   v = v << s;               // normalizujemy dzielną
   vn1 = v >> 16;            // rozbijamy dzielną
   vn0 = v & 0xFFFF;         // na dwie cyfry 16-bitowe

   un32 = (u1 << s) | (u0 >> 32 - s) & (-s >> 31);
   un10 = u0 << s;           // dzielną przesuwamy w lewo

   un1 = un10 >> 16;         // prawą połowę dzielnej rozbijamy
   un0 = un10 & 0xFFFF;      // na dwie cyfry

   q1 = un32/vn1;            // wyliczamy pierwszą cyfrę
   rhat = un32 - q1*vn1;     // ilorazu, q1.
again1:
   if (q1 >= b || q1*vn0 > b*rhat + un1) {
     q1 = q1 - 1;
     rhat = rhat + vn1;
     if (rhat < b) goto again1;}

   un21 = un32*b + un1 - q1*v;  // mnożenie i odejmowanie

   q0 = un21/vn1;            // wyliczamy drugą cyfrę
   rhat = un21 - q0*vn1;     // ilorazu, q0
again2:
   if (q0 >= b || q0*vn0 > b*rhat + un0) {
     q0 = q0 - 1;
     rhat = rhat + vn1;
     if (rhat < b) goto again2;}

   if (r != NULL)            // jeśli potrzebna jest reszta
      *r = (un21*b + un0 - q0*v) >> s;     // zwracamy ją
   return q1*b + q0;
}
```

Jako sygnalizację przepełnienia, program zwraca resztę z dzielenia równą maksymalnej liczbie całkowitej bez znaku. Jest to reszta niemożliwa do uzyskania w wyniku prawidłowej operacji *dzielenia*, ponieważ prawidłowa reszta musi być mniejsza od dzielnika.

W przypadku przepełnienia program zwraca również iloraz równy największej liczbie całkowitej bez znaku, która w niektórych przypadkach może być wystarczającym sygnałem sytuacji przepełnienia, o ile nie wykorzystuje się reszty z dzielenia.

Dziwne wyrażenie `(-s << 31)` w przypisaniu zmiennej `u32` występuje w celu umożliwienia działania programu dla `s = 0` na maszynach udostępniających przesunięcia modulo 32 (np. Intel x86).

Eksperymenty z liczbami losowymi o rozkładzie jednostajnym pozwalają na przyjęcie założenia, że pętle `again` wykonują się w 38% wykonań funkcji. Oznacza to, że średnio potrzeba 52 instrukcji, jeśli nie wykorzystuje się reszty z dzielenia. Jedną z tych instrukcji jest *wyliczenie liczby zer wiodących*, dwie są instrukcjami *dzielenia*. Wykorzystywanych jest również średnio 6,5 mnożeń (nie licząc mnożeń przez `b`, które są realizowane przez przesunięcia). Jeśli potrzebna jest reszta z dzielenia, dodatkowo potrzeba sześciu instrukcji (licząc zapisanie w pamięci wartości `r`), z których jedną jest mnożenie.

Została nam jeszcze wersja ze znakiem funkcji `divlu`. Prawdopodobnie trudno byłoby zmodyfikować kod z listingu 9.3 krok po kroku, tak aby uzyskać jego wersję ze znakiem. Algorytm ten można jednak wykorzystać w celu implementacji dzielenia ze znakiem, gdyby wyliczyć wartości bezwzględne argumentów, wykonać `divlu` a następnie odpowiednio zmodyfikować wynik, aby uzyskać odpowiedni znak (jeśli znaki dzielnej i dzielnika są przeciwne). Nie ma problemów z wartościami ekstremalnymi, jak na przykład największa liczba ujemna, ponieważ wartość bezwzględna z każdej liczby całkowitej ze znakiem da się prawidłowo zinterpretować w postaci liczby bez znaku. Odpowiedni algorytm przedstawiamy na listingu 9.4.

Listing 9.4. *Dzielenie wartości long ze znakiem (ang. divide long signed) za pomocą funkcji dzielenia typu long bez znaku*

```
int divls(int u1, unsigned u0, int v, int *r) {
   int q, uneg, vneg, diff, borrow;

   uneg = u1 >> 31;          // -1 if u < 0.
   if (uneg) {               // wyliczamy wartośc bezwzględną
      u0 = -u0;              // dzielnej u.
      borrow = (u0 != 0);
      u1 = -u1 - borrow;}

   vneg = v >> 31;           // -1 if v < 0.
   v = (v ^ vneg) - vneg;    // wartośc bezwzględna v.

   if ((unsigned)u1 >= (unsigned)v) goto overflow;

   q = divlu(u1, u0, v, (unsigned *)r);

   diff = uneg ^ vneg;        // negujemy wartość v
   q = (q ^ diff) - diff;    // jeśli znaki u i v są przeciwne
   if (uneg && r != NULL)
      *r = -*r;
```

```
    if ((diff ^ q) < 0 && q != 0) {  // w przypadku przepełninia
    overflow:                        // ustawiamy resztę
          if (r != NULL)             // na niemożliwą wartość,
             *r = 0x80000000;        // a iloraz ustawiamy
          q = 0x80000000;}           // na największą możliwą wartość
       return q;
    }
```

Trudno jest wymyślić naprawdę dobry kod wykrywający przepełnienie w uogólnionym przypadku. Algorytm zaprezentowany na listingu 9.4 wykorzystuje wstępny warunek identyczny do wykorzystywanego w procedurze długiego dzielenia bez znaku, które polega na sprawdzeniu, czy $|u/v| < 2^{32}$. Po tej operacji koniecznie należy sprawdzić, czy iloraz posiada odpowiedni znak lub jest równy 0.

Rozdział 10.

Dzielenie liczb całkowitych przez stałe

W przypadku wielu komputerów dzielenie jest operacją bardzo kosztowną pod względem czasu wykonania i należy jej unikać, kiedy to tylko jest możliwe. Wartość równoważna 20-krotności czasu wymaganego do wykonania operacji *dodawania* nie jest niczym niezwykłym, a ponadto czas ten zwykle stanowi tę samą wysoką wartość, nawet gdy operandy są małe. W niniejszym rozdziale przedstawiono pewne metody umożliwiające uniknięcie konieczności użycia instrukcji *dzielenia* w przypadku, gdy dzielnik jest stałą.

10.1. Dzielenie ze znakiem przez znaną potęgę liczby 2

Bez wątpienia wiele osób popełnia błąd polegający na przyjęciu założenia, że operacja *shift right signed* (*przesunięcie w prawo ze znakiem*) o k pozycji daje w efekcie podzielenie liczby przez wartość 2^k, przy użyciu zwykłej obcinającej formy dzielenia [GLS2]. Problem ten okazuje się nieco bardziej skomplikowany. Przedstawiony poniżej kod pozwala obliczyć wartość wyrażenia $q = n \div 2^k$ dla $1 \le k \le 31$ [Hop].

```
shrsi t,n,k-1      Utworzenie liczby całkowitej
shri  t,t,32-k     2**k - 1, jeżeli n < 0; w przeciwnym razie 0.
add   t,n,t        Dodanie jej do n
shrsi q,t,k        i przesunięcie w prawo (ze znakiem)
```

Kod nie zawiera instrukcji *skoków*. Ponadto w często spotykanym przypadku dzielenia przez 2 ($k = 1$) jego wielkość redukuje się do trzech wierszy. Jednakże jego wykonanie jest uzależnione od zdolności danej maszyny do przesuwania o dużą liczbę pozycji w krótkim czasie. Przypadek $k = 31$ nie ma zbyt dużego znaczenia praktycznego,

gdyż maszyna nie jest w stanie zapewnić reprezentacji liczby 2^{31}. Mimo to kod pozwala w tym przypadku na otrzymanie poprawnego wyniku (którym jest $q = -1$, jeżeli $n = -2^{31}$ oraz $q = 0$ dla wszystkich innych wartości n).

W celu podzielenia przez wartość -2^k powyższy kod można uzupełnić instrukcją *negacji*. Wydaje się, że nie istnieje żaden lepszy sposób wykonania tej operacji.

Prostszy kod dzielenia przez 2^k ma następującą postać.

```
        bge   n,label       Skok, jeżeli n >= 0
        addi  n,n,2**k-1    Dodanie 2**k - 1 do n
label shrsi n,n,k           oraz przesunięcie w prawo (ze znakiem)
```

Ta wersja jest preferowana w przypadku maszyn o wolnych operacjach *przesunięcia* i szybkich *skokach*.

Procesor PowerPC posiada rzadko spotykany mechanizm służący do przyspieszenia operacji *dzielenia* przez potęgi liczby 2 [GGS]. Instrukcje *przesunięcia w prawo ze znakiem* ustawiają bit przeniesienia wówczas, gdy przesuwana liczba jest ujemna i jedna lub większa liczba jedynek zostaje wysunięta poza rejestr. Procesor ten udostępnia również instrukcję służącą do dodawania bitu przeniesienia do rejestru, noszącą nazwę `addze`. Pozwala to na wykonanie dzielenia przez dowolną (dodatnią) potęgę liczby 2 w dwóch instrukcjach:

```
shrsi q,n,k
addze q,q
```

Pojedyncze przesunięcie za pomocą instrukcji `shrsi` o k pozycji powoduje wykonanie swego rodzaju dzielenia ze znakiem przez wartość 2^k, które jest zgodne zarówno z dzieleniem z modułem, jak i z dzieleniem z funkcją *podłoga*. To sugeruje, że w przypadku komputerów i języków wysokiego poziomu jedna z tych metod może być bardziej odpowiednia niż dzielenie z obcinaniem. To znaczy, że dzielenie z modułem i dzielenie z funkcją *podloga* odpowiadają instrukcji `shrsi` lepiej niż dzielenie z obcinaniem, co umożliwia kompilatorowi przetłumaczenie wyrażenia $n / 2$ na instrukcję `shrsi`. Ponadto instrukcja `shrsi` z występującą po niej instrukcją `neg` (*negacji*) powoduje wykonanie dzielenia z modułem przez -2^k, co stanowi wskazówkę, że prawdopodobnie dzielenie z modułem jest najlepszym wyborem (jednak jest to w głównej mierze kwestia estetyki — ma niewielkie znaczenie praktyczne, ponieważ dzielenie przez stałą ujemną bez wątpienia stanowi bardzo rzadko wykonywaną operację).

10.2. Reszta ze znakiem z dzielenia przez znaną potęgę liczby 2

Jeżeli potrzebny jest zarówno iloraz, jak i reszta z dzielenia $\boldsymbol{n \div 2^k}$, najprostszym rozwiązaniem jest obliczenie reszty ze wzoru $\boldsymbol{r = q * 2^k - n}$. Wymaga to wykonania tylko dwóch dodatkowych instrukcji po obliczeniu ilorazu $\boldsymbol{q}$:

```
shli  r,q,k
sub   r,r,n
```

Obliczenie samej reszty wydaje się wymagać czterech lub pięciu instrukcji. Jednym ze sposobów jest wykorzystanie sekwencji czterech instrukcji, przedstawionej w poprzednim podrozdziale dla dzielenia ze znakiem przez 2^k, wraz z przedstawioną powyżej sekwencją dwóch instrukcji. W wyniku otrzymuje się występujące obok siebie dwie instrukcje *przesunięcia*, które można zastąpić operacją iloczynu logicznego *and*, uzyskując rozwiązanie składające się z pięciu instrukcji (lub czterech dla przypadku $k = 1$):

```
shrsi t,n,k-1       Utworzenie liczby całkowitej
shri  t,t,32-k      2**k - 1, jeżeli n < 0, w przeciwnym razie 0.
add   t,n,t         Dodanie jej do n,
andi  t,t,-2**k     wyzerowanie k bitów od prawej strony
sub   r,n,t         i odjęcie jej od n.
```

Inna metoda bazuje na następującej zależności

$$\mathrm{rem}(n,2^k) = \begin{cases} n \,\&\, (2^k - 1), & n \geq 0, \\ -((-n) \,\&\, (2^k - 1)), & n < 0. \end{cases}$$

W celu jej wykorzystania w pierwszej kolejności należy obliczyć $t \leftarrow n \overset{s}{>>} 31$, a następnie

$$\boldsymbol{r} \leftarrow ((\mathrm{abs}(\boldsymbol{n}) \,\&\, (\mathbf{2}^k - \mathbf{1})) \oplus \boldsymbol{t}) - \boldsymbol{t}$$

(pięć instrukcji) lub, dla $k = 1$, z uwagi na fakt, że $(-\boldsymbol{n})$ **& 1 = n & 1**:

$$\boldsymbol{r} \leftarrow ((\boldsymbol{n} \,\&\, 1) \oplus \boldsymbol{t}) - \boldsymbol{t}$$

(cztery instrukcje). Metoda ta nie sprawdza się dla przypadku $k > 1$, jeżeli maszyna nie oferuje instrukcji *absolute value* (*wartość bezwzględna*) — obliczenie reszty wymagałoby wówczas siedmiu instrukcji.

Jeszcze inna metoda bazuje na zależności

$$\mathrm{rem}(n, 2^k) = \begin{cases} n \,\&\, (2^k - 1), & n \geq 0, \\ ((n + 2^k - 1) \,\&\, (2^k - 1)) - (2^k - 1), & n < 0. \end{cases}$$

Prowadzi to do

$$\boldsymbol{t} \leftarrow (\boldsymbol{n} \overset{s}{>>} k - 1) \overset{u}{>>} 32 - k$$
$$\boldsymbol{r} \leftarrow ((\boldsymbol{n} + \boldsymbol{t}) \,\&\, (\mathbf{2}^k - \mathbf{1})) - \boldsymbol{t}$$

(pięć instrukcji dla $k > 1$, cztery dla $k = 1$).

Wszystkie omówione metody działają poprawnie dla $1 \leq k \leq 31$.

Przypadkowo okazuje się, że jeżeli instrukcja *przesunięcia w prawo ze znakiem* nie jest dostępna, wartość, która wynosi $2^k - 1$ dla $n < 0$ oraz 0 dla $n \geq 0$, można obliczyć na podstawie zależności

$$t_1 \leftarrow n \overset{u}{>>} 31$$
$$r \leftarrow (t_1 << k) - t_1 .$$

Powoduje to dodanie tylko jednej instrukcji.

10.3. Dzielenie i reszta ze znakiem w przypadku innych wartości niż potęgi liczby 2

Podstawowa sztuczka polega na wykonaniu mnożenia przez swego rodzaju odwrotność dzielnika d, w przybliżeniu $2^{32} / d$, a następnie na wyodrębnieniu 32 bitów od lewej strony iloczynu. Szczegóły implementacji tej metody są jednak bardziej skomplikowane, zwłaszcza w przypadku niektórych dzielników, takich jak 7.

Najpierw zostanie rozpatrzonych kilka konkretnych przykładów. Ilustrują one postać kodu generowanego przez metodę ogólną. Rejestry zostają określone w sposób następujący:

`n` — wejściowa liczba całkowita (licznik, ang. *numerator*)

`M` — rejestr z załadowaną „liczbą magiczną"

`t` — rejestr tymczasowy

`q` — rejestr zawierający iloraz (ang. *quotient*)

`r` — rejestr zawierający resztę (ang. *remainder*)

Dzielenie przez 3

```
li    M,0x55555556  Załadowanie liczby magicznej, (2**32+2)/3
mulhs q,M,n         q = floor(M*n/2**32)
shri  t,n,31        Dodanie 1 do q, jeżeli
add   q,q,t         n jest ujemne.

muli  t,q,3         Obliczenie reszty ze wzoru
sub   r,n,t         r = n - q*3.
```

Dowód. Operacja *multiply high signed* (`mulhs`, *mnożenie ze znakiem dla bardziej znaczącego części wyniku*) nie może spowodować przepełnienia, gdyż iloczyn dwóch 32-bitowych liczb całkowitych zawsze można reprezentować za pomocą 64 bitów a instrukcja `mulhs` zwraca 32 bardziej znaczące bity 64-bitowego iloczynu. Jest to równoważne

podzieleniu 64-bitowego iloczynu przez 2^{32} i zastosowaniu wobec otrzymanego wyniku funkcji *podłoga*. W tym przypadku nie ma znaczenia, czy iloczyn jest dodatni, czy ujemny. Stąd dla $n \geq 0$ powyższy kod oblicza wartość wyrażenia:

$$q = \left\lfloor \frac{2^{32}+2}{3} \frac{n}{2^{32}} \right\rfloor = \left\lfloor \frac{n}{3} + \frac{2n}{3 \cdot 2^{32}} \right\rfloor.$$

Zatem $n < 2^{31}$, gdyż wartość $2^{31} - 1$ to największa możliwa do reprezentowania liczba dodatnia. Stąd składnik „błędu" $2n / (3 \cdot 2^{32})$ jest mniejszy od 1/3 (i jest nieujemny), co na podstawie twierdzenia D4 (strona 164) pozwala stwierdzić, że $q = \lfloor n / 3 \rfloor$. Stanowi to poszukiwany rezultat (równanie (1) na stronie 163).

W przypadku $n < 0$ należy dodać 1 do ilorazu. Stąd też kod oblicza wartość wyrażenia:

$$q = \left\lfloor \frac{2^{32}+2}{3} \frac{n}{2^{32}} \right\rfloor + 1 = \left\lfloor \frac{2^{32}n + 2n + 3 \cdot 2^{32}}{3 \cdot 2^{32}} \right\rfloor = \left\lceil \frac{2^{32}n + 2n + 1}{3 \cdot 2^{32}} \right\rceil,$$

gdzie skorzystano z twierdzenia D2. Zatem

$$q = \left\lceil \frac{n}{3} + \frac{2n+1}{3 \cdot 2^{32}} \right\rceil.$$

Dla $-2^{31} \leq n \leq -1$

$$-\frac{1}{3} + \frac{1}{3 \cdot 2^{32}} \leq \frac{2n+1}{3 \cdot 2^{32}} \leq -\frac{1}{3 \cdot 2^{32}}.$$

Składnik błędu jest nieujemny i większy od $-1/3$, tak więc na podstawie twierdzenia D4 można stwierdzić, że $q = \lceil n / 3 \rceil$, co stanowi poszukiwany wynik (równanie (1) na stronie 163).

Można zatem uznać iloraz za poprawny. Poprawność reszty w prosty sposób wynika z faktu, że po pierwsze, musi ona spełniać zależność

$$n = qd + r,$$

po drugie, mnożenie przez 3 nie może spowodować przepełnienia, ponieważ $-2^{31} / 3 \leq q \leq (2^{31} - 1) / 3$, zaś po trzecie, operacja *odejmowania* nie może spowodować przepełnienia, ponieważ wynik musi należeć do przedziału od -2 do $+2$.

Operacja *multiply immediate* (*mnożenie przez wartość natychmiastową*) może zostać wykonana przy użyciu dwóch instrukcji *dodawania* lub instrukcji *przesunięcia* oraz *dodawania*, o ile któreś z tych rozwiązań może dać poprawę wydajności działania.

W przypadku wielu współczesnych procesorów RISC iloraz można obliczać w sposób przedstawiony powyżej w dziewięciu lub dziesięciu cyklach, gdy instrukcja *divide* wymaga około 20 cykli.

Dzielenie przez 5

W przypadku dzielenia przez 5 korzystne byłoby wykorzystanie tego samego kodu, co w przypadku dzielenia przez 3. Różnica polegałaby na tym, że mnożnikiem byłaby wartość $(2^{32} + 4) / 5$. Niestety, wówczas składnik błędu staje się zbyt duży; wynik jest błędny o 1 w przypadku około 1/5 wszystkich wartości $n \geq 2^{30}$ (co do wartości bezwzględnej). Jednakże istnieje możliwość użycia mnożnika $(2^{33} + 3) / 5$ i dodania instrukcji *przesunięcia w prawo ze znakiem*. Odpowiedni kod ma postać:

```
li    M,0x66666667  Załadowanie liczby magicznej, (2**33+3)/5
mulhs q,M,n         q = floor(M*n/2**32)
shrsi q,q,1
shri  t,n,31        Dodanie 1 do q, jeżeli
add   q,q,t         n jest ujemne.

muli  t,q,5         Obliczenie reszty ze wzoru
sub   r,n,t         r = n - q*5.
```

Dowód. Instrukcja `mulhs` zwraca 32 bity od lewej strony 64-bitowego iloczynu, a następnie jest wykonywane przesunięcie w prawo o jedną pozycję, z zachowaniem znaku (czyli „arytmetycznie"). Jest to równoważne podzieleniu iloczynu przez 2^{33} a następnie zastosowaniu względem wyniku funkcji *podłoga*. Stąd dla $n \geq 0$ powyższy kod oblicza wartość wyrażenia

$$q = \left\lfloor \frac{2^{33}+3}{5} \frac{n}{2^{33}} \right\rfloor = \left\lfloor \frac{n}{5} + \frac{3n}{5 \cdot 2^{33}} \right\rfloor.$$

Dla $0 \leq n \leq 2^{31}$ składnik błędu $3n / (5 \cdot 2^{33})$ jest nieujemny i mniejszy od 1/5, tak więc na podstawie twierdzenia D4 można stwierdzić, że $q = \lfloor n / 5 \rfloor$.

Dla $n < 0$ powyższy kod oblicza wartość wyrażenia

$$q = \left\lfloor \frac{2^{33}+3}{5} \frac{n}{2^{33}} \right\rfloor + 1 = \left\lceil \frac{n}{5} + \frac{3n+1}{5 \cdot 2^{33}} \right\rceil.$$

Składnik błędu jest nieujemny i jest większy od $-1/5$, tak więc $q = \lceil n / 5 \rceil$.

Poprawność obliczenia wartości reszty można uzasadnić podobnie jak w przypadku dzielenia przez 3.

Operacja *mnożenia* przez wartość natychmiastową może zostać wykonana przy użyciu instrukcji *przesunięcia w lewo* o dwie pozycje oraz instrukcji *dodawania*.

Dzielenie przez 7

Dzielenie przez 7 niesie ze sobą nowy problem. Mnożniki $(2^{32} + 3) / 7$ oraz $(2^{33} + 6) / 7$ powodują, że składniki błędu przyjmują zbyt dużą wartość. Mnożnik $(2^{34} + 5) / 7$ byłby odpowiedni, jednak ma on zbyt dużą wartość, aby móc go reprezentować za pomocą

32-bitowego słowa ze znakiem. Istnieje możliwość wykonania mnożenia przez tę liczbę poprzez zastąpienie jej mnożeniem przez $(2^{34} + 5) / 7 - 2^{32}$ (liczba ujemna), a następnie wprowadzenie do iloczynu poprawki za pomocą instrukcji *add*. Odpowiedni kod ma następującą postać:

```
li    M,0x92492493  Liczba magiczna, (2**34+5)/7 - 2**32.
mulhs q,M,n         q = floor(M*n/2**32)
add   q,q,n         q = floor(M*n/2**32) + n.
shrsi q,q,1         q = floor(q/4).
shri  t,n,31        Dodanie 1 do q, jeżeli
add   q,q,t         n jest ujemne.

muli  t,q,7         Obliczenie reszty ze wzoru
sub   r,n,t         r = n - q*7.
```

Dowód. Istotne znaczenie ma zauważenie, że występująca powyżej instrukcja `add q,q,n` nie może spowodować przepełnienia. Wynika to z faktu, że q i n mają przeciwne znaki w wyniku wykonania mnożenia przez liczbę ujemną. Zatem takie dodawanie w ramach „arytmetyki komputerowej" stanowi dokładny odpowiednik dodawania liczb rzeczywistych. Stąd dla $n \geq 0$ powyższy kod oblicza wartość wyrażenia

$$q = \left\lfloor \left(\left\lfloor \left(\frac{2^{34}+5}{7} - 2^{32} \right) \frac{n}{2^{32}} \right\rfloor + n \right) / 4 \right\rfloor = \left\lfloor \left\lfloor \frac{2^{34}n + 5n - 7 \cdot 2^{32}n + 7 \cdot 2^{32}n}{7 \cdot 2^{32}} \right\rfloor / 4 \right\rfloor = \left\lfloor \frac{n}{7} + \frac{5n}{7 \cdot 2^{34}} \right\rfloor,$$

gdzie skorzystano z wniosku z twierdzenia D3.

Dla $0 \leq n \leq 2^{31}$ składnik błędu $5n / (7 \cdot 2^{34})$ jest nieujemny i mniejszy od 1/7, tak więc $q = \lfloor n / 7 \rfloor$.

Dla $n < 0$ powyższy kod oblicza wartość wyrażenia

$$q = \left\lfloor \left(\left\lfloor \left(\frac{2^{34}+5}{7} - 2^{32} \right) \frac{n}{2^{32}} \right\rfloor + n \right) / 4 \right\rfloor + 1 = \left\lceil \frac{n}{7} + \frac{5n+1}{7 \cdot 2^{34}} \right\rceil.$$

Składnik błędu jest nieujemny i większy od –1/7, tak więc $q = \lceil n / 7 \rceil$.

Operacja *mnożenia* przez wartość natychmiastową może zostać wykonana przy użyciu instrukcji *przesunięcia w lewo* o trzy pozycje oraz instrukcji *odejmowania*.

10.4. Dzielenie ze znakiem przez dzielniki ≥ 2

W tym momencie Czytelnik być może zastanawia się, czy inne dzielniki niosą ze sobą dodatkowe problemy. W niniejszym podrozdziale zostanie wykazane, że tak nie jest. Przedstawione trzy przykłady ilustrują jedyne problematyczne przypadki, które mogą się pojawić (dla $d \geq 2$).

Niektóre z prezentowanych dowodów są nieco skomplikowane, należy więc zachować ostrożność, gdyż wykonywane działania opierają się na ogólnym rozmiarze słowa wynoszącym W.

Przyjmując jako rozmiar słowa $W \geq 3$ oraz określając dzielnik d, gdzie $2 \leq d \leq 2^{W-1}$, chcemy znaleźć najmniejszą liczbę całkowitą m oraz liczbę całkowitą p, takie że

$$\left\lfloor \frac{mn}{2^p} \right\rfloor = \left\lfloor \frac{n}{d} \right\rfloor \qquad \text{dla } 0 \leq n < 2^{W-1} \text{ oraz} \tag{1a}$$

$$\left\lfloor \frac{mn}{2^p} \right\rfloor + 1 = \left\lceil \frac{n}{d} \right\rceil \qquad \text{dla } -2^{W-1} \leq n \leq -1, \tag{1b}$$

gdzie $0 \leq m < 2^W$ oraz $p \geq W$.

Powodem, dla którego chcemy, aby liczba m była najmniejszą z możliwych jest to, że mniejszy mnożnik może oznaczać konieczność wykonania przesunięcia o mniejszą liczbę pozycji (być może zero) lub może dać kod przypominający bardziej przykład dzielenia przez 5 niż dzielenia przez 7. Musi być spełnione $m \leq 2^{W-1}$, tak aby kod nie zawierał więcej instrukcji niż dla przypadku dzielenia przez 7 (to znaczy, aby było możliwe obsłużenie mnożnika z zakresu od 2^{W-1} do $2^W - 1$ za pomocą instrukcji add, dodanej w przykładzie dzielenia przez 7, zamiast borykania się z większymi mnożnikami). Musi również zachodzić $p \geq W$, ponieważ generowany kod wydziela lewą połowę iloczynu mn, co jest równoważne przesunięciu w prawo o W pozycji. Stąd całkowita liczba pozycji, o jakie należy dokonać przesunięcia w prawo jest równa W lub więcej.

Istnieje różnica między mnożnikiem m a liczbą magiczną M. Liczba magiczna jest wartością używaną w instrukcji *mnożenia*. Określa ją zależność

$$M = \begin{cases} m, & \text{jeżeli } 0 \leq m < 2^{W-1} \\ m - 2^W, & \text{jeżeli } 2^{W-1} \leq m < 2^W. \end{cases}$$

Z uwagi na fakt, że zależność (1b) musi być spełniona dla $n = -d$, zachodzi związek $\lfloor -mn / 2^p \rfloor + 1 = -1$, z czego wynika

$$\frac{md}{2^p} > 1. \tag{2}$$

Niech n_c będzie największą (dodatnią) wartością n, taką że $\mathrm{rem}(n_c, d) = d - 1$. n_c istnieje, gdyż jedną z możliwych wartości jest $n_c = d - 1$. Można ją obliczyć na podstawie zależności $n_c = \lfloor 2^{W-1} / d \rfloor d - 1 = 2^{W-1} - \mathrm{rem}(2^{W-1}, d) - 1$. n_c jest jedną z największych d dopuszczalnych wartości n, tak więc

$$2^{W-1} - d \leq n_c \leq 2^{W-1} - 1 \tag{3a}$$

oraz oczywiście

$$n_c \geq d - 1. \tag{3b}$$

Z uwagi na fakt, że musi być spełniona zależność (1a) dla $n = n_c$ wiadomo, że

$$\left\lfloor \frac{mn_c}{2^p} \right\rfloor = \left\lfloor \frac{n_c}{d} \right\rfloor = \frac{n_c - (d-1)}{d}$$

lub

$$\frac{mn_c}{2^p} < \frac{n_c + 1}{d}.$$

Połączenie tych zależności z nierównością (2) daje

$$\frac{2^p}{d} < m < \frac{2^p}{d}\frac{n_c + 1}{n_c} \tag{4}$$

Ponieważ wartość m ma być najmniejszą liczbą całkowitą spełniającą zależność (4), jest ona najmniejszą liczbą całkowitą większą od $2^p / d$; to znaczy

$$m = \frac{2^p + d - \mathrm{rem}(2^p, d)}{d}. \tag{5}$$

W połączeniu z prawą stroną zależności (4), upraszczając wyrażenie, otrzymujemy

$$2^p > n_c(d - \mathrm{rem}(2^p, d)). \tag{6}$$

Algorytm

Algorytm znajdowania liczby magicznej M oraz liczby pozycji przesunięcia s na podstawie wartości d opiera się na obliczeniu w pierwszej kolejności n_c, a następnie rozwiązaniu nierówności (6) względem p poprzez wypróbowanie kolejno coraz wyższych wartości. Jeżeli $p < W$, należy ustalić $p = W$ (przedstawione poniżej twierdzenie pokazuje, że taka wartość p również spełnia nierówność (6)). Kiedy zostaje znalezione najmniejsze $p \geq W$, spełniające nierówność (6), m można wyliczyć z równania (5). Jest to najmniejsza możliwa wartość m, gdyż ustalono najmniejszą dopuszczalną wartość p, a na podstawie nierówności (4) łatwo sprawdzić, że zmniejszanie wartości p powoduje zmniejszanie wartości m. W końcu wiadomo, że $s = p - W$, zaś M stanowi po prostu przewartościowanie m na liczbę całkowitą ze znakiem (w ten sposób interpretuje ją instrukcja `mulhs`).

Wymuszenie warunku, aby p było co najmniej tak duże jak W, uzasadnia przedstawione poniżej twierdzenie.

TWIERDZENIE DC1

twierdzenie Jeżeli nierówność (6) jest spełniona dla pewnej wartości p, wówczas jest spełniona dla wszystkich większych wartości p.

Dowód. Załóżmy, że nierówność (6) jest spełniona dla $p = p_0$. Pomnożenie obu stron nierówności (6) przez 2 daje

$$2^{p_0+1} > n_c(2d - 2\mathrm{rem}(2^{p_0}, d)).$$

Z twierdzenia D5 wynika, że $\mathrm{rem}(2^{p_0+1}, d) \geq 2\mathrm{rem}(2^{p_0}, d) - d$. Połączenie tych zależności daje

$$2^{p_0+1} > n_c(2d - (\mathrm{rem}(2^{p_0+1}, d) + d)), \text{czyli}$$
$$2^{p_0+1} > n_c(d - \mathrm{rem}(2^{p_0+1}, d)).$$

Zatem nierówność (6) jest spełniona dla $p = p_0 + 1$, a stąd także dla wszystkich wyższych wartości.

Nierówność (6) można by rozwiązać za pomocą przeszukiwania binarnego, choć preferowaną metodą jest raczej proste przeszukiwanie liniowe (rozpoczynając od $p = W$), ponieważ d jest zazwyczaj małe, a małe wartości d pociągają za sobą małe wartości p.

Dowód wykonalności algorytmu

Należy wykazać, że nierówność (6) zawsze ma rozwiązanie oraz że $0 \leq m \leq 2^W$ (nie ma potrzeby wykazywania, że $p \geq W$, ponieważ warunek ten jest wymuszony).

Wykażemy, że nierówność (6) zawsze ma rozwiązanie poprzez otrzymanie górnego ograniczenia dla p. Niejako przy okazji zostanie również określone dolne ograniczenie przy założeniu, że p nie musi być co najmniej równe W. W celu otrzymania tych ograniczeń dla p należy zauważyć, że dla dowolnej dodatniej liczby całkowitej x istnieje potęga 2 większa od x i mniejsza lub równa $2x$. Zatem, na podstawie nierówności (6) można zapisać

$$n_c(d - \mathrm{rem}(2^p, d)) < 2^p \leq 2n_c(d - \mathrm{rem}(2^p, d)).$$

Z uwagi na fakt, że $0 \leq \mathrm{rem}(2^p, d) \leq d - 1$

$$n_c + 1 \leq 2^p \leq 2n_c d. \tag{7}$$

Z nierówności (3a) i (3b) wynika, że $n_c \geq \max(2^{W-1} - d, d + 1)$. Krzywe $f_1(d) = 2^{W-1} - d$ oraz $f_2(d) = d - 1$ przecinają się w punkcie $d = (2^{W-1} + 1) / 2$. Stąd $n_c \geq (2^{W-1} - 1) / 2$. n_c jest liczbą całkowitą, zatem zachodzi związek $n_c \geq 2^{W-2}$. $d \leq 2^{W-1} - 1$, więc nierówność (7) przyjmuje postać

$$2^{W-2} + 1 \leq 2^p \leq 2(2^{W-1} - 1)^2$$

lub

$$W - 1 \leq p \leq 2W - 2. \tag{8}$$

Dolne ograniczenie $p = W - 1$ może wystąpić (np. dla $W = 32$, $d = 3$), ale w tym przypadku określamy $p = W$.

Jeżeli p nie musi być równe W, wówczas na podstawie nierówności (4) i (7) można zapisać

$$\frac{n_c + 1}{d} < m < \frac{2n_c d}{d} \frac{n_c + 1}{n_c}.$$

Wykorzystanie nierówności (3b) pozwala zapisać

$$\frac{d-1+1}{d} < m < 2(n_c+1).$$

Ponieważ $n_c \le 2^{W-1} - 1$ (w nierówności 3a), to

$$2 < m < 2^W - 1.$$

Jeżeli p musi być równe W, wówczas na podstawie nierówności (4) można zapisać

$$\frac{2^W}{d} < m < \frac{2^W}{d}\frac{n_c+1}{n_c}.$$

Skoro $2 \le d \le 2^{W-1} - 1$ oraz $n_c \ge 2^{W-2}$, to

$$\frac{2^W}{2^{W-1}-1} < m < \frac{2^W}{2}\frac{2^W-2+1}{2^{W-2}} \text{ lub}$$

$$3 < m < 2^{W-1}+1.$$

Stąd w obu przypadkach m należy do przedziału ograniczonego schematem kodu przedstawionego w przykładzie dotyczącym dzielenia przez 7.

Dowód poprawności wyniku

Należy wykazać, że jeżeli p i m zostaną obliczone na podstawie zależności (6) i (5), to będą spełnione równania (1a) oraz (1b).

Łatwo zauważyć, że równanie (5) oraz nierówność (6) implikują zależność (4) — w przypadku, gdy p musi być równe W nierówność (6) wciąż jest zachowana, co pokazuje twierdzenie DC1. Poniżej zostanie oddzielnie rozpatrzonych pięć przedziałów wartości n:

$$0 \le n \le n_c$$
$$n_c + 1 \le n \le n_c + d - 1,$$
$$-n_c \le n \le -1,$$
$$-n_c - d + 1 \le n \le -n_c - 1 \quad \text{oraz}$$
$$n = -n_c - d.$$

Na podstawie nierówności (4), ponieważ m jest liczbą całkowitą, otrzymujemy

$$\frac{2^p}{d} < m < \frac{2^p(n_c+1)-1}{dn_c}.$$

Pomnożenie przez $n / 2^p$, przy założeniu $n \ge 0$, pozwala zamienić to wyrażenie na

$$\frac{n}{d} \le \frac{mn}{2^p} \le \frac{2^p n(n_c+1)-n}{2^p dn_c}, \text{ tak więc}$$

$$\left\lfloor \frac{n}{d} \right\rfloor \le \left\lfloor \frac{mn}{2^p} \right\rfloor \le \left\lfloor \frac{n}{d} + \frac{(2^p - 1) - n}{2^p dn_c} \right\rfloor.$$

Dla $0 \le n \le n_c$, $0 \le (2^p - 1)n / (2^p dn_c) < 1/d$. Tak więc, na mocy twierdzenia D4,

$$\left\lfloor \frac{n}{d} + \frac{(2^p - 1) - n}{2^p dn_c} \right\rfloor = \left\lfloor \frac{n}{d} \right\rfloor.$$

Zatem równanie (1a) jest w tym przypadku ($0 \le n \le n_c$) spełnione.

Dla $n > n_c$, n jest ograniczone do zakresu

$$n_c + 1 \le n \le n_c + d - 1, \tag{9}$$

ponieważ warunek $n \ge n_c + d$ jest sprzeczny z wyborem n_c jako największej wartości n, takiej że $\mathrm{rem}(n_c, d) = d - 1$ (na podstawie nierówności (3a) można również wykazać, że $n \ge n_c + d$ implikuje $n \ge 2^{W-1}$). Z nierówności (4), dla $n \ge 0$, wynika

$$\frac{n}{d} < \frac{mn}{2^p} < \frac{n}{d}\frac{n_c + 1}{n_c}.$$

Za pomocą elementarnych przekształceń algebraicznych wyrażenie to można zapisać w postaci

$$\frac{n}{d} < \frac{mn}{2^p} < \frac{n_c + 1}{d} + \frac{(n - n_c)(n_c + 1)}{dn_c}. \tag{10}$$

Z warunku (9) wynika, że $1 \le n - n_c \le d - 1$, tak więc

$$0 < \frac{(n - n_c)(n_c + 1)}{dn_c} \le \frac{d - 1}{d}\frac{n_c + 1}{n_c}.$$

$n_c \ge d - 1$ (na podstawie nierówności (3b)), zaś $(n_c + 1) / n_c$ osiąga maksimum wówczas, gdy n_c osiąga minimum, więc

$$0 < \frac{(n - n_c)(n_c + 1)}{dn_c} \le \frac{d - 1}{d}\frac{d - 1 + 1}{d - 1} = 1.$$

W nierówności (10) składnik $(n_c + 1) / d$ jest liczbą całkowitą. Składnik $(n - n_c)(n_c + 1) / dn_c$ jest mniejszy lub równy 1. Zatem nierówność (10) można zapisać jako

$$\left\lfloor \frac{n}{d} \right\rfloor \le \left\lfloor \frac{mn}{2^p} \right\rfloor \le \frac{n_c + 1}{d}.$$

Dla wszystkich n należących do przedziału (9), $\lfloor n / d \rfloor = (n_c + 1) / d$. Stąd w tym przypadku ($n_c + 1 \le n \le n_c + d - 1$) równanie (1a) jest spełnione.

Dla $n < 0$, z uwagi na fakt, że m jest liczbą całkowitą, z nierówności (4) otrzymujemy

$$\frac{2^p+1}{d} \le m < \frac{2^p}{d}\frac{n_c+1}{n_c}.$$

Po pomnożeniu przez $n / 2^p$, dla $n < 0$, otrzymujemy

$$\frac{n}{d}\frac{n_c+1}{n_c} < \frac{mn}{2^p} \le \frac{n}{d}\frac{2^p+1}{2^p},$$

czyli

$$\left\lfloor \frac{n}{d}\frac{n_c+1}{n_c} \right\rfloor + 1 \le \left\lfloor \frac{mn}{2^p} \right\rfloor + 1 \le \left\lfloor \frac{n}{d}\frac{2^p+1}{2^p} \right\rfloor + 1.$$

Skorzystanie z twierdzenia D2 daje

$$\left\lfloor \frac{n(n_c+1) - dn_c + 1}{dn_c} \right\rfloor + 1 \le \left\lfloor \frac{mn}{2^p} \right\rfloor + 1 \le \left\lceil \frac{n(2^p+1) - 2^p d + 1}{2^p d} \right\rceil + 1,$$

$$\left\lceil \frac{n(n_c+1)+1}{dn_c} \right\rceil \le \left\lfloor \frac{mn}{2^p} \right\rfloor + 1 \le \left\lceil \frac{n(2^p+1)+1}{2^p d} \right\rceil.$$

Ponieważ $n + 1 \le 0$, prawa nierówność może zostać osłabiona, a stąd

$$\left\lceil \frac{n}{d} + \frac{n+1}{dn_c} \right\rceil \le \left\lfloor \frac{mn}{2^p} \right\rfloor + 1 \le \left\lceil \frac{n}{d} \right\rceil. \tag{11}$$

Dla $-n_c \le n \le -1$,

$$\frac{-n_c+1}{dn_c} \le \frac{n+1}{dn_c} \le 0,$$

czyli

$$\frac{-1}{d} < \frac{n+1}{dn_c} \le 0.$$

Stąd, na mocy twierdzenia D4,

$$\left\lceil \frac{n}{d} + \frac{n+1}{dn_c} \right\rceil = \left\lceil \frac{n}{d} \right\rceil,$$

tak więc równanie (1b) jest spełnione w tym przypadku ($-n_c \le n \le -1$).

Dla $n < -n_c$, n jest ograniczone do przedziału

$$-n_c - d \le n \le -n_c - 1. \tag{12}$$

(Na podstawie nierówności (3a) $n < -n_c - d$ implikuje $n < -2^{W-1}$, co jest niemożliwe). Elementarne przekształcenia algebraiczne przeprowadzone na lewym wyrażeniu nierówności (11) dają

$$\left\lceil \frac{-n_c - 1}{d} + \frac{(n+n_c)(n_c+1)+1}{dn_c} \right\rceil \le \left\lfloor \frac{mn}{2^p} \right\rfloor + 1 \le \left\lceil \frac{n}{d} \right\rceil. \tag{13}$$

Dla $-n_c - d \le n \le -n_c - 1$,

$$\frac{(-d+1)(n_c+1)}{dn_c} + \frac{1}{dn_c} \le \frac{(n+n_c)(n_c+1)+1}{dn_c} \le \frac{-(n_c+1)+1}{dn_c} = -\frac{1}{d}.$$

Proporcja $(n_c + 1) / n_c$ osiąga maksimum, kiedy n_c jest minimalne; to znaczy, $n_c = d - 1$. Stąd

$$\frac{(-d+1)(d-1+1)}{d(d-1)} + \frac{1}{dn_c} \le \frac{(n+n_c)(n_c+1)+1}{dn_c} < 0, \text{ czyli}$$

$$-1 \le \frac{(n+n_c)(n_c+1)+1}{dn_c} < 0.$$

Ponieważ $(-n_c - 1) / d$ jest liczbą całkowitą oraz dodawana do niej wartość należy do przedziału od -1 do 0, z nierówności (13) wynika

$$\frac{-n_c - 1}{d} \le \left\lfloor \frac{mn}{2^p} \right\rfloor + 1 \le \left\lceil \frac{n}{d} \right\rceil.$$

Dla n należącego do przedziału $-n_c - d + 1 \le n \le -n_c - 1$

$$\left\lceil \frac{n}{d} \right\rceil = \frac{-n_c - 1}{d}.$$

Stąd $\lfloor mn / 2^p \rfloor + 1 = \lceil n / d \rceil$ — oznacza to, że równanie (1b) jest spełnione.

Ostatni przypadek, $n = -n_c - d$, może wystąpić tylko w przypadku określonych wartości d. Z nierówności (3a) wynika, że $-n_c - d \le -2^{W-1}$, tak więc, jeżeli n przyjmie taką wartość, musi być spełniony warunek $n = -n_c - d = -2^{W-1}$, a stąd $n_c = 2^{W-1} - d$. Dlatego $\mathrm{rem}(2^{W-1}, d) = \mathrm{rem}(n_c + d, d) = d - 1$ (to znaczy, że d dzieli $2^{W-1} + 1$).

W takim przypadku ($n = -n_c - d$) nierówność (6) posiada rozwiązanie $p = W - 1$ (najmniejsza możliwa wartość p), ponieważ dla $p = W - 1$ zachodzi

$$\begin{aligned} n_c(d - \mathrm{rem}(2^p, d)) &= (2^{W-1} - d)(d - \mathrm{rem}(2^{W-1}, d)) \\ &= (2^{W-1} - d)(d - (d-1)) = 2^{W-1} - d < 2^{W-1} = 2^p. \end{aligned}$$

Następnie na podstawie równania (5) otrzymujemy

$$m = \frac{2^{W-1} + d - \mathrm{rem}(2^{W-1}, d)}{d} = \frac{2^{W-1} + d - (d-1)}{d} = \frac{2^{W-1} + 1}{d}.$$

Stąd

$$\left\lfloor \frac{mn}{2^p} \right\rfloor + 1 = \left\lfloor \frac{2^{W-1} + 1}{d} \frac{-2^{W-1}}{2^{W-1}} \right\rfloor + 1 = \left\lfloor \frac{-2^{W-1} - 1}{d} \right\rfloor + 1$$
$$= \left\lceil \frac{-2^{W-1} - d}{d} \right\rceil + 1 = \left\lceil \frac{-2^{W-1}}{d} \right\rceil = \left\lceil \frac{n}{d} \right\rceil,$$

tak więc równanie (1b) jest spełnione.

To kończy dowód, że jeżeli m i p są obliczane na podstawie związków (5) i (6), to równania (1a) i (1b) są spełnione dla wszystkich dopuszczalnych wartości n.

10.5. Dzielenie ze znakiem przez dzielniki ≤ –2

Całkowitoliczbowe dzielenie ze znakiem spełnia zależność $n \div (-d) = -(n \div d)$, zatem poprawnym podejściem jest napisanie kodu dla operacji $n \div |d|$, a następnie zanegowanie ilorazu (takie postępowanie nie daje poprawnego wyniku dla $d = -2^{W-1}$, ale dla tej oraz innych ujemnych potęg liczby 2 można użyć kodu z podrozdziału 10.1 *Dzielenie ze znakiem przez znane potęgi liczby 2* ze strony 181, uzupełniając go o końcową instrukcję *negacji*). Nie wystarczy zanegowanie dzielnej ze względu na możliwość wystąpienia sytuacji, że będzie to maksymalna wartość ujemna.

Istnieje jednak możliwość uniknięcia wykonywania instrukcji *negacji*. W tym celu należy obliczyć

$$q = \left\lfloor \frac{mn}{2^p} \right\rfloor, \quad \text{kiedy } n \le 0,$$
$$q = \left\lfloor \frac{mn}{2^p} \right\rfloor + 1, \quad \text{kiedy } n > 0.$$

Jednakże dodawanie 1, w przypadku, gdy $n > 0$, jest niewygodne (ponieważ nie można po prostu użyć bitu znaku wartości n), tak więc w kodzie dodawanie będzie wykonywane wówczas, gdy $q < 0$. Jest to rozwiązanie równoważne, gdyż mnożnik m jest ujemny (co zostanie poniżej pokazane).

Odpowiedni kod przedstawiony poniżej odpowiada przypadkowi $W = 32$, $d = -7$.

```
li    M,0x6DB6DB6D  Liczba magiczna, -(2**34+5)/7 + 2**32.
mulhs q,M,n         q = floor(M*n/2**32).
sub   q,q,n         q = floor(M*n/2**32) - n.
shrsi q,q,2         q = floor(q/4).
shri  t,q,31        Dodanie 1 do q, jeżeli
add   q,q,t         q jest ujemne (n jest dodatnie).
```

```
muli  t,q,-7        Obliczenie reszty ze wzoru
sub   r,n,t         r = n - q*(-7).
```

Kod ten jest taki sam, jak w przypadku dzielenia przez +7, z tą różnicą, że wykorzystuje negację mnożnika dla +7, instrukcję sub zamiast add po wykonaniu mnożenia oraz w instrukcji shri przesunięcia o 31 pozycji jest używane q zamiast n, co omówiono powyżej (w przypadku $d = +7$ można użyć wartości q także tutaj, jednak oznaczałoby to zmniejszenie poziomu zrównoleglenia kodu). Operacja *odejmowania* nie powoduje przepełnienia, ponieważ operandy mają ten sam znak. Taki schemat postępowania jednak nie zawsze funkcjonuje poprawnie! Choć powyższy kod dla przypadku $W = 32$, $d = -7$ jest prawidłowy, to analogiczna zamiana kodu „dzielenia przez 3" na kod umożliwiający dzielenie przez –3 nie zapewnia poprawnych wyników dla $W = 32$ i $n = -2^{31}$.

Poniżej kwestia ta zostanie zbadana bliżej.

Posiadając rozmiar słowa $W \geq 3$ oraz dzielnik d, taki że $-2^{W-1} \leq d \leq -2$, chcemy znaleźć najmniejszą (co do wartości bezwzględnej) liczbę całkowitą m oraz liczbę całkowitą p, takie że

$$\left\lfloor \frac{mn}{2^p} \right\rfloor = \left\lfloor \frac{n}{d} \right\rfloor \qquad \text{dla } -2^{W-1} \leq n < 0 \text{ oraz} \tag{14a}$$

$$\left\lfloor \frac{mn}{2^p} \right\rfloor + 1 = \left\lceil \frac{n}{d} \right\rceil \qquad \text{dla } 1 \leq n \leq 2^{W-1}, \tag{14b}$$

gdzie $-2^W \leq m < 0$ oraz $p \geq W$.

Postępując podobnie jak w przypadku dzielenia przez dodatni dzielnik, niech n_c będzie największą ujemną wartością n, taką że $n_c = kd + 1$ dla pewnej liczby całkowitej k. Wartość n_c istnieje, ponieważ jedną z możliwości jest $n_c = d + 1$. Wartość tę można wyliczyć z zależności $n_c = \lfloor(-2^{W-1} - 1) / d\rfloor d + 1 = -2^{W-1} + \text{rem}(2^{W-1} + 1, d)$. n_c jest jedną z najmniejszych $|d|$ dopuszczalnych wartości n, tak więc

$$-2^{W-1} \leq n_c \leq -2^{W-1} - d - 1 \tag{15a}$$

oraz oczywiście

$$n_c \leq d + 1. \tag{15b}$$

Równanie (14b) musi być spełnione dla $n = -d$, a równanie (14a) — dla $n = n_c$, Zatem otrzymujemy, analogicznie do nierówności (4),

$$\frac{2^p}{d} \frac{n_c - 1}{n_c} < m < \frac{2^p}{d}. \tag{16}$$

Wartość m ma być największą liczbą całkowitą spełniającą zależność (16) a więc jest ona największą liczbą całkowitą mniejszą od $2^p / d$; to znaczy

$$m = \frac{2^p - d - \text{rem}(2^p, d)}{d}. \tag{17}$$

W połączeniu z lewą stroną zależności (16), upraszczając wyrażenie, otrzymujemy

$$2^p > n_c(d + \mathrm{rem}(2^p, d)). \tag{18}$$

Dowód wykonalności algorytmu sugerowanego przez zależności (17) i (18) oraz poprawności otrzymanego wyniku jest podobny do analogicznego dowodu, przedstawionego dla dodatniego dzielnika i nie zostanie tu powtórzony. Pojawia się jednak trudność w przypadku próby udowodnienia, że $-2^W \le m \le 0$. W tym celu należy oddzielnie rozpatrzyć przypadki, w których d jest negacją potęgi liczby 2 lub pewną inną liczbą. Dla $d = -2^k$ łatwo wykazać, że $n_c = -2^{W-1} + 1$, $p = W + k - 1$ oraz $m = -2^{W-1} - 1$ (należy do przedziału). Dla d niebędącego w postaci -2^k, w prosty sposób można zmodyfikować wcześniej przedstawiony dowód.

Dla jakich dzielników m(–d) ≠ –m(d)?

Symbol $m(d)$ oznacza mnożnik odpowiadający dzielnikowi d. Jeżeli $m(-d) \ne -m(d)$, kod dzielenia przez ujemny dzielnik można utworzyć poprzez obliczenie mnożnika dla $|d|$, zanegowanie go a następnie utworzenie kodu podobnego do przedstawionego powyżej przypadku „dzielenia przez –7".

Porównując wyrażenie (18) z (6) oraz wyrażenie (17) z (5) można zauważyć, że jeżeli wartość n_c dla $-d$ jest negacją analogicznej wartości dla d, to $m(-d) = -m(d)$. Stąd warunek $m(-d) \ne -m(d)$ jest spełniony tylko wówczas, gdy wartość n_c obliczona dla ujemnego dzielnika jest maksymalną liczbą ujemną, -2^{W-1}. Takie dzielniki są negacjami czynników wartości $2^{W-1} + 1$. Liczby takie pojawiają się dość rzadko, co ilustrują przedstawione poniżej przykłady rozkładu na czynniki:

$$2^{15} + 1 = 3 \cdot 11 \cdot 331$$

$$2^{15} + 1 = 3 \cdot 715\,827\,883$$

$$2^{63} + 1 = 3^3 \cdot 19 \cdot 43 \cdot 5419 \cdot 77\,158\,673\,929$$

Dla *wszystkich* tych czynników $m(-d) \ne -m(d)$. Zarys dowodu można określić następująco: dla $d > 0$ zachodzi równość $n_c = 2^{W-1} - d$. Z uwagi na fakt, że $\mathrm{rem}(2^{W-1}, d) = d - 1$ nierówność (6) jest spełniona dla $p = W - 1$, a stąd także dla $p = W$. Jednak dla $d < 0$ zachodzi równość $n_c = -2^{W-1} - d$ oraz $\mathrm{rem}(2^{W-1}, d) = |d| - 1$. Tak więc nierówność (18) nie jest spełniona dla $p = W - 1$ ani dla $p = W$, a stąd $p > W$.

10.6. Zawieranie obsługi w kompilatorze

Aby kompilator mógł zamienić dzielenie przez stałą na mnożenie, musi obliczyć liczbę magiczną M oraz liczbę pozycji do przesunięcia s przy danym dzielniku d. Narzucającym się sposobem wykonania odpowiednich obliczeń jest wyznaczanie wartości

wyrażenia (6) lub (18) dla $p = W, W + 1, \ldots$ do momentu, aż zostanie ono spełnione. Wówczas m można obliczyć na podstawie wzoru (5) lub (17). M jest po prostu przewartościowaniem m jako liczby całkowitej ze znakiem, zaś $s = p - W$.

Opisany poniżej schemat zapewnia obsługę przypadku, gdy d jest liczbą dodatnią i ujemną za cenę dodania niewielkiej porcji kodu i pozwala uniknąć arytmetyki na podwójnych słowach.

Gwoli przypomnienia, n_c definiujemy jako

$$n_c = \begin{cases} 2^{W-1} + \mathrm{rem}(2^{W-1}, d) - 1, & \text{dla } d > 0, \\ -2^{W-1} + \mathrm{rem}(2^{W-1} + 1, d), & \text{dla } d < 0. \end{cases}$$

Zatem $|n_c|$ można wyliczyć z zależności

$$t = 2^{W-1} + \begin{cases} 0, \text{ dla } d > 0, \\ 1, \text{ dla } d < 0. \end{cases}$$

$$|n_c| = t - 1 - \mathrm{rem}(t, |d|).$$

Resztę należy obliczyć korzystając z dzielenia bez znaku z uwagi na wartość bezwzględną argumentów. Powyżej użyto zapisu rem(t, $|d|$) zamiast równoważnego rem(t, d) w celu podkreślenia faktu, że program musi w tym miejscu operować na dwóch dodatnich (i bez znaku) argumentach.

Na podstawie nierówności (6) i (18) p można wyliczyć z zależności

$$2^p > |n_c|(|d| - \mathrm{rem}(2^p, |d|)), \tag{19}$$

a następnie $|m|$ można wyliczyć z zależności (porównaj równania (5) i (17)):

$$|m| = \frac{2^p + |d| - \mathrm{rem}(2^p, |d|)}{|d|}. \tag{20}$$

Bezpośrednie wyznaczenie wartości rem(2^p, $|d|$) w nierówności (19) wymaga „dzielenia długiego" (ang. *long division* — podzielenia $2W$-bitowej dzielnej przez W-bitowy dzielnik w celu otrzymania W-bitowego ilorazu i reszty) i w rzeczywistości musi to być dzielenie długie *bez znaku*. Istnieje sposób na rozwiązanie nierówności (19), a co więcej na wykonanie wszystkich obliczeń, bez przeprowadzania dzielenia długiego, który można z łatwością zaimplementować w zwykłym języku wysokiego poziomu przy wykorzystaniu tylko W-bitowej arytmetyki. Wymagane jest jednak wykonanie dzielenia bez znaku oraz porównań bez znaku.

Wartość rem(2^p, $|d|$) można obliczać przyrostowo, inicjalizując dwie zmienne q i r wartościami ilorazu i reszty z dzielenia 2^p przez $|d|$ dla $p = 2^{W-1}$, a następnie aktualizować wartość q wraz ze wzrostem wartości r i p.

W miarę postępu wyszukiwania, to znaczy gdy wartość p jest zwiększana o 1, wartości q i r są aktualizowane zgodnie z następującymi zależnościami (patrz twierdzenie D5(a))

```
q = 2*q;
r = 2*r;
if (r >= abs(d)) {
   q = q + 1;
   r = r - abs(d);}
```

Lewa część nierówności (4) oraz prawa część nierówności (16) wraz z udowodnionymi ograniczeniami dla wartości m implikują związek $q = \lfloor 2^p / |d| \rfloor < 2^W$, tak więc q można reprezentować jako W-bitową liczbę całkowitą bez znaku. Ponadto $0 \le r \le |d|$, co oznacza, że r można reprezentować jako W-bitową liczbę całkowitą ze znakiem lub bez znaku (uwaga: wynik pośredni $2r$ może przekroczyć wartość $2^{W-1} - 1$, tak więc wartość r powinna być liczbą bez znaku, podobnie jak wykonywane powyżej porównywanie).

Następnie obliczamy $\delta = |d| - r$. Oba wyrazy odejmowania mogą być reprezentowane jako W-bitowe liczby całkowite bez znaku, a jego wynik spełnia zależność $1 \le \delta \le |d|$, więc nie pojawią się żadne problemy.

W celu uniknięcia mnożenia długiego związanego z nierównością (19), należy ją przepisać w postaci

$$\frac{2^p}{|d|} > \delta.$$

Wielkość $2^p / |n_c|$ można reprezentować jako W-bitową liczbę całkowitą bez znaku (podobnie jak w przypadku zależności (7), na podstawie nierówności (19) można wykazać, że $2^p \le 2|n_c| \cdot |d|$ oraz, dla $d = -2^{W-1}$, $n_c = -2^{W-1} + 1$ i $p = 2W - 2$, tak więc $2^p / |n_c| = 2^{2W-2} / (2^{W-1} - 1) < 2^W$ dla $W \ge 3$). Ponadto można ją łatwo obliczać przyrostowo (wraz ze wzrostem p) w ten sam sposób, co w przypadku wielkości $\mathrm{rem}(2^p, |d|)$. Porównanie powinno być dokonane bez znaku ze względu na możliwość wystąpienia przypadku $2^p / |n_c| \ge 2^{W-1}$ (może on wystąpić dla dużych d).

W celu obliczenia m nie trzeba wyznaczać wartości równania (20) bezpośrednio (co wiązałoby się z koniecznością wykonania dzielenia długiego). Można zauważyć, że

$$\frac{2^p + |d| - \mathrm{rem}(2^p, |d|)}{|d|} = \left\lfloor \frac{2^p}{|d|} \right\rfloor + 1 = q + 1.$$

Wyznaczenie wartości testu domknięcia pętli $2^p / |n_c| > \delta$ jest nieco utrudnione. Wielkość $2^p / |n_c|$ jest dostępna jedynie w postaci ilorazu q_1 i reszty r_1. Wartość $2^p / |n_c|$ może ale nie musi być liczbą całkowitą (jest nią tylko dla $d = 2^{W-2} + 1$ oraz kilku ujemnych wartości d). Sprawdzenie $2^p / |n_c| \le \delta$ można zakodować jako

$$\boldsymbol{q_1 < \delta \mid (q_1 = \delta \;\&\; r_1 = 0)}.$$

Pełna procedura obliczania `M` oraz `s` na podstawie `d` została przedstawiona na listingu 10.1, zawierającym program w języku C, napisany dla $W = 32$. W kilku miejscach może wystąpić przepełnienie, jednak w razie jego zignorowania otrzymuje się poprawny wynik.

Listing 10.1. *Obliczanie liczby magicznej dla dzielenia ze znakiem*

```
struct ms {int M;          // Liczba magiczna
           int s;};        // oraz liczba pozycji przesunięcia.

struct ms magic(int d) {   // Musi zachodzić 2 <= d <= 2**31-1
                           // lub   -2**31 <= d <= -2.
   int p;
   unsigned ad, anc, delta, q1, r1, q2, r2, t;
   const unsigned two31 = 0x80000000;     // 2**31.
   struct ms mag;

   ad = abs(d);
   t = two31 + ((unsigned)d >> 31);
   anc = t - 1 - t%ad;     // Wartość bezwzględna z nc.
   p = 31;                 // Inicjalizacja p.
   q1 = two31/anc;         // Inicjalizacja q1 = 2**p/|nc|.
   r1 = two31 - q1*anc;    // Inicjalizacja r1 = rem(2**p, |nc|).
   q2 = two31/ad;          // Inicjalizacja q2 = 2**p/|d|.
   r2 = two31 - q2*ad;     // Inicjalizacja r2 = rem(2**p, |d|).
   do {
      p = p + 1;
      q1 = 2*q1;           // Modyfikacja q1 = 2**p/|nc|.
      r1 = 2*r1;           // Modyfikacja r1 = rem(2**p, |nc|.
      if (r1 >= anc) {     // (Musi to być porównanie
         q1 = q1 + 1;      // bez znaku).
         r1 = r1 - anc;}
      q2 = 2*q2;           // Modyfikacja q2 = 2**p/|d|.
      r2 = 2*r2;           // Modyfikacja r2 = rem(2**p, |d|.
      if (r2 >= ad) {      // (Musi to być porównanie
         q2 = q2 + 1;      // bez znaku).
         r2 = r2 - ad;}
      delta = ad - r2;
   } while (q1 < delta || (q1 == delta && r1 == 0));

   mag.M = q2 + 1;
   if (d < 0) mag.M = -mag.M; // Zwracana liczba magiczna oraz
   mag.s = p - 32;            // liczba pozycji przesunięcia.
   return mag;
}
```

Aby korzystać z wyników tego programu, kompilator powinien wygenerować instrukcje `li` oraz `mulhs` oraz instrukcję `add`, jeżeli `d` > 0 i `M` < 0 lub operację `sub`, jeżeli `d`< 0 i `M` > 0, a także instrukcję `shrsi`, jeżeli `s` > 0. Wówczas muszą również zostać wygenerowane instrukcje `shri` oraz końcowa instrukcja `add`.

Dla $W = 32$ można uniknąć konieczności operowania na ujemnym dzielniku po prostu dzięki zwróceniu wcześniej obliczonego wyniku dla $d = 3$ i $d = 715\ 827\ 883$ i korzystając z zależności $m(-d) = -m(d)$ dla innych ujemnych dzielników. Jednak taki program nie byłby znacząco krótszy od przedstawionego na listingu 10.1 a nawet mógłby okazać się dłuższy.

10.7. Inne zagadnienia

TWIERDZENIE DC2

twierdzenie Najmniejszy mnożnik m jest nieparzysty, jeżeli p nie musi być równe W.

Dowód. Załóżmy, że równania (1a) oraz (1b) są spełnione dla najmniejszej (niewymuszonej) liczby całkowitej p, a m jest parzyste. Wówczas, oczywiście, m można by podzielić przez 2, zaś od p odjąć 1 i równania (1a) oraz (1b) wciąż byłyby spełnione. Jest to sprzeczne z założeniem, że p ma wartość minimalną.

Unikatowość

Liczba magiczna dla danego dzielnika jest czasem unikatowa (np. dla $W = 32$, $d = 7$) ale często jest inaczej. Przykładowo dla $W = 32$, $d = 6$ istnieją cztery liczby magiczne:

$$\begin{array}{llll} M = & 715\,827\,833 & ((2^{32}+2)/6), & s = 0 \\ M = & 1\,431\,655\,766 & ((2^{32}+2)/3), & s = 1 \\ M = & -1\,431\,655\,765 & ((2^{33}+1)/3 - 2^{32}), & s = 2 \\ M = & -1\,431\,655\,764 & ((2^{33}+4)/3 - 2^{32}), & s = 2. \end{array}$$

Jednakże istnieje następująca właściwość unikatowości:

TWIERDZENIE DC3

twierdzenie Dla danego dzielnika d istnieje tylko jeden mnożnik m, charakteryzujący się minimalną wartością p, jeżeli p nie musi być równe W.

Dowód. Najpierw rozpatrzmy przypadek $d > 0$. Różnica między górnym a dolnym ograniczeniem nierówności (4) wynosi $2^p / dn_c$. Wcześniej udowodniliśmy własność (7), że jeżeli p ma wartość minimalną, to $2^p / dn_c \le 2$. Stąd mogą istnieć co najwyżej dwie wartości m spełniające nierówność (4). Niech m będzie mniejszą z tych wartości, określoną wzorem (5). Wówczas $m + 1$ jest drugą z tych wartości.

Niech p_0 będzie najmniejszą wartością p, dla której $m + 1$ spełnia prawą stronę nierówności (4) (p_0 nie musi być równe W). Wtedy

$$\frac{2^{p_0} + d - \mathrm{rem}(2^{p_0}, d)}{d} + 1 < \frac{2^{p_0}}{d}\frac{n_c + 1}{n_c}.$$

Po uproszczeniu otrzymujemy

$$2^{p_0} > n_c(2d - \mathrm{rem}(2^{p_0}, d)).$$

Podzielenie przez 2 daje

$$2^{p_0 - 1} > n_c\left(d - \frac{1}{2}\mathrm{rem}(2^{p_0}, d)\right).$$

Ze względu na to, że $\mathrm{rem}(2^{p_0}, d) \le 2\mathrm{rem}(2^{p_0-1}, d)$ (na mocy twierdzenia D5 ze strony 164),

$$2^{p_0-1} > n_c(d - \mathrm{rem}(2^{p_0-1}, d)),$$

co stoi w sprzeczności z założeniem, że p_0 ma minimalną wartość.

Dowód dla $d < 0$ jest podobny i nie zostanie tu przedstawiony.

Dzielniki w najlepszych programach

Program dla $d = 3$, $W = 32$ jest wyjątkowo krótki, ponieważ po instrukcji `mulhs` nie występuje instrukcja `add` ani `shrsi`. Pojawia się pytanie, jakie inne dzielniki posiada ten program.

Bierzemy pod uwagę tylko dodatnie dzielniki. Chcemy znaleźć liczby całkowite m i p, które spełniają równania (1a) i (1b) i dla których $p = W$ oraz $0 \le m \le 2^{W-1}$. Wszystkie rozwiązania nierówności (4) dla $p = W$ można wyrazić jako

$$m = \frac{2^W + kd - \mathrm{rem}(2^W, d)}{d}, \text{ dla } k = 1, 2, 3, \ldots$$

Połączenie tej zależności z prawą częścią nierówności (4) i uproszczenie pozwala otrzymać

$$\mathrm{rem}(2^W, d) > kd - \frac{2^W}{n_c}. \tag{21}$$

Najsłabsze ograniczenie wobec $\mathrm{rem}(2^W, d)$ występuje dla $k = 1$ oraz n_c mającego swoją minimalną wartość, wynoszącą 2^{W-2}. Stąd musi zachodzić

$$\mathrm{rem}(2^W, d) > d - 4;$$

oznacza to, że d dzieli $2^W + 1$, $2^W + 2$ i $2^W + 3$.

Poniżej zostanie sprawdzone, które z tych czynników faktycznie posiadają optymalne programy.

Jeżeli d dzieli $2^W + 1$, to $\mathrm{rem}(2^W, d) = d - 1$. Wówczas rozwiązaniem nierówności (6) jest $p = W$, ponieważ przyjmuje ona postać

$$2^W > n_c(d - (d-1)) = n_c,$$

która bez wątpienia jest spełniona, gdyż $n_c < 2^{W-1}$. Następnie, obliczając m, otrzymujemy

$$m = \frac{2^W + d - (d-1)}{d} = \frac{2^W + 1}{d},$$

która to wartość jest mniejsza od 2^{W-1} dla $d \geq 3$ ($d \neq 2$, ponieważ d dzieli $2^W + 1$). Stąd wszystkie czynniki wyrażenia $2^W + 1$ posiadają optymalne programy.

Podobnie, jeżeli d dzieli $2^W + 2$, to rem(2^W, d) = $d - 2$. Ponownie rozwiązaniem nierówności (6) jest $p = W$, ponieważ przyjmuje ona postać

$$2^W > n_c(d-(d-2)) = 2n_c,$$

która oczywiście jest spełniona. Obliczając wartość m, otrzymujemy

$$m = \frac{2^W + d - (d-2)}{d} = \frac{2^W + 2}{d},$$

która to wartość przekracza $2^{W-1} - 1$ dla $d = 2$, ale jest mniejsza lub równa $2^{W-1} - 1$ dla $W \geq 3$, $d \geq 3$ (przypadek $W = 3$, $d = 3$ nie występuje, gdyż 3 nie jest czynnikiem rozkładu wyrażenia $2^3 + 2 = 10$). Stąd wszystkie czynniki wyrażenia $2^W + 2$, z wyjątkiem 2 oraz dopełnienia algebraicznego 2, posiadają optymalne programy (dopełnieniem algebraicznym 2 jest $(2^W + 2) / 2$, a wartości tej nie da się reprezentować jako W-bitowej liczby całkowitej ze znakiem).

Jeżeli d dzieli $2^W + 3$, poniższa argumentacja pokazuje, że d nie posiada optymalnego programu. rem(2^W, d) = $d - 3$, zatem nierówność (21) implikuje, że musi zachodzić

$$n_c < \frac{2^W}{kd - d + 3}$$

dla pewnego $k = 1, 2, 3, \ldots$. Najsłabsze ograniczenie występuje dla $k = 1$, tak więc musi być spełnione $n_c < 2^W / 3$.

Z nierówności (3a) wynika, że $n_c \geq 2^{W-1} - d$, czyli $d \geq 2^{W-1} - n_c$. Zatem musi być spełniona zależność

$$d < 2^{W-1} - \frac{2^W}{3} = \frac{2^W}{6}.$$

Ponadto, z uwagi na fakt, że liczby 2, 3 i 4 nie dzielą wyrażenia $2^W + 3$, najmniejszym możliwym czynnikiem tego wyrażenia jest 5. Stąd największym możliwym czynnikiem jest $(2^W + 3) / 5$. W konsekwencji, jeżeli d dzieli $2^W + 3$ oraz d posiada optymalny program, musi być spełniona zależność

$$\frac{2^W}{6} < d \leq \frac{2^W + 3}{5}.$$

Zapisanie odwrotności względem wyrazu $2^W + 3$ pokazuje, że dopełnienie algebraiczne d, $(2^W + 3) / d$, posiada ograniczenia

$$5 \leq \frac{2^W + 3}{d} < \frac{(2^W + 3) \cdot 6}{2^W} = 6 + \frac{18}{2^W}.$$

Dla $W \geq 5$ to implikuje, że jedynymi możliwymi dopełnieniami algebraicznymi są wartości 5 i 6. Dla $W < 5$ łatwo sprawdzić, że nie istnieją żadne czynniki rozkładu dla $2^W + 3$. 6 nie może być czynnikiem $2^W + 3$, zatem jedyną możliwością jest 5. Dlatego jedynym możliwym czynnikiem rozkładu wyrażenia $2^W + 3$, który może posiadać optymalny program jest $(2^W + 3) / 5$.

Dla $d = (2^W + 3) / 5$,

$$n_c = \left\lfloor \frac{2^{W-1}}{(2^W + 3)/5} \right\rfloor \left(\frac{2^W + 3}{5} \right) - 1.$$

Dla $W \geq 4$,

$$2 \leq \frac{2^{W-1}}{(2^W + 3)/5} < 2{,}5.$$

Zatem

$$n_c = 2\left(\frac{2^W + 3}{5} \right) - 1.$$

Jest to wartość wyższa od $(2^W / 3)$, a stąd $d = (2^W + 3) / 5$ nie posiada optymalnego programu. Z uwagi na fakt, że dla $W < 4$ nie istnieją czynniki rozkładu $2^W + 3$ można wywnioskować, że żadne czynniki $2^W + 3$ nie posiadają optymalnych programów.

Podsumowując, wszystkie czynniki wyrażeń $2^W + 1$ oraz $2^W + 2$, oprócz 2 i $(2^W + 2) / 2$, posiadają optymalne programy i nie można tego powiedzieć o żadnej innej liczbie. Ponadto przedstawiony powyżej dowód pokazuje, że algorytm *magic* (listing 10.1 ze strony 200) zawsze daje optymalny program, kiedy ten istnieje.

Warto rozważyć przypadki szczególne $W = 16$, 32 i 64. Odpowiednie sposoby rozkładu na czynniki przedstawiono poniżej.

$$2^{16} + 1 = 65537 \text{ (liczba pierwsza)} \qquad 2^{32} + 1 = 641 \cdot 6\,700\,417$$

$$2^{16} + 2 = 2 \cdot 3^2 \cdot 11 \cdot 331 \qquad 2^{32} + 2 = 2 \cdot 3 \cdot 715\,827\,883$$

$$2^{64} + 1 = 247\,177 \cdot 67\,280\,421\,310\,721$$

$$2^{64} + 2 = 2 \cdot 3^3 \cdot 19 \cdot 43 \cdot 5419 \cdot 77\,158\,673\,929$$

Zatem posiadamy wyniki dla $W = 16$ — istnieje 20 dzielników, które posiadają optymalne programy. Te z nich, które są mniejsze od 100 to: 3, 6, 9, 11, 18, 22, 33, 66 i 99.

Dla $W = 32$ istnieje sześć takich dzielników: 3, 6, 641, 6 700 417, 715 827 883 oraz 1 431 655 766.

Dla $W = 64$ istnieje 126 takich dzielników. Mniejsze od 100 to 3, 6, 9, 18, 19, 27, 38, 43, 54, 57 oraz 86.

10.8. Dzielenie bez znaku

Dzielenie bez znaku przez potęgi 2 implementuje się, oczywiście, za pomocą pojedynczej instrukcji *shift right logical* (*logicznego przesunięcia w prawo*), zaś resztę określa się za pomocą instrukcji *and immediate* (*iloczynu logicznego z wartością natychmiastową*).

Może się wydawać, że obsłużenie innych dzielników będzie proste dzięki użyciu po prostu wyników dzielenia ze znakiem dla $d > 0$ i pominięciu dwóch instrukcji, które dodają 1, kiedy iloraz jest ujemny. Jak się jednak okaże, w przypadku dzielenia bez znaku pewne szczegóły postępowania są w rzeczywistości bardziej skomplikowane.

Dzielenie bez znaku przez liczbę 3

Przypadek liczb niebędących potęgami liczby 2 rozpoczniemy od dzielenia bez znaku przez 3 na maszynie 32-bitowej. Ponieważ dzielna n może tym razem mieć wartość do $2^{32} - 1$, mnożnik $(2^{32} + 2) / 3$ jest nieodpowiedni, ponieważ składnik błędu $2n / (3 * 2^{32})$ (patrz przedstawiony wcześniej przykład dzielenia przez 3) może przekroczyć wartość 1/3. Jednakże mnożnik $(2^{33} + 1) / 3$ jest prawidłowy. Odpowiedni kod ma postać

```
li    M,0xAAAAAAAB  Załadowanie liczby magicznej (2**33+1)/3
mulhu q,M,n         q = floor(M*n/2**32).
shri  q,q,1

muli  t,q,3         Obliczenie reszty ze wzoru
sub   r,n,t         r = n - q*3.
```

Potrzebna jest instrukcja, która dawałaby w wyniku 32 bardziej znaczące bity 64-bitowego iloczynu bez znaku, którą powyżej reprezentuje mnemonik `mulhu`.

Aby sprawdzić, że powyższy kod jest poprawny, należy zauważyć, że oblicza on

$$q = \left\lfloor \frac{2^{33}+1}{3} \frac{n}{2^{33}} \right\rfloor = \left\lfloor \frac{n}{3} + \frac{2n}{3 \cdot 2^{33}} \right\rfloor.$$

Dla $0 \le n < 2^{32}$ zachodzi $0 \le n / (3 \cdot 2^{33}) < 1/3$, tak więc, na mocy twierdzenia D4, $q = \lfloor n / 3 \rfloor$.

W przypadku obliczania reszty instrukcja *multiply immediate* może spowodować przepełnienie, jeżeli operandy potraktuje się jako liczby całkowite ze znakiem, ale nie wystąpi ono, jeżeli potraktuje się je, wraz z wynikiem, jako wartości bez znaku. Ponadto instrukcja *odejmowania* nie spowoduje przepełnienia, ponieważ wynik należy do przedziału od 0 do 2, tak więc reszta jest poprawna.

Dzielenie bez znaku przez liczbę 7

W przypadku dzielenia bez znaku przez liczbę 7 na maszynie 32-bitowej mnożniki $(2^{32} + 3) / 7$, $(2^{33} + 6) / 7$ oraz $(2^{34} + 5) / 7$ wszystkie są nieodpowiednie, ponieważ dają zbyt duży składnik błędu. Mnożnik $(2^{35} + 3) / 7$ byłby odpowiedni, ale ma on zbyt dużą wartość, żeby reprezentować go jako 32-bitowe słowo bez znaku. Mnożenie przez tę dużą wartość można wykonać mnożąc przez $(2^{35} + 3) / 7 - 2^{32}$ a następnie korygując iloczyn za pomocą instrukcji dodawania add. Odpowiedni kod ma postać

```
li    M,0x24924925  Załadowanie liczby magicznej (2**35+3)/7 - 2**32
mulhu q,M,n         q = floor(M*n/2**32).
add   q,q,n         Może spowodować przepełnienie (ustawić flagę przepełnienia).
shrxi q,q,3         Przesunięcie w prawo z bitem przepełnienia.

muli  t,q,7         Obliczenie reszty ze wzoru
sub   r,n,t         r = n - q*7.
```

W tym momencie pojawia się pewien problem. Instrukcja add może spowodować przepełnienie. W celu uwzględnienia tej ewentualności wprowadzamy nową instrukcję *shift right extended immediate* (shrxi, *rozszerzone przesunięcia w prawo o wartość natychmiastową*), która traktuje bit przepełnienia z instrukcji add oraz 32 bity rejestru q jako pojedynczą 33-bitową wielkość i przesuwa ją w prawo z wypełnianiem zerami. W przypadku rodziny procesorów Motorola 68000 można tego dokonać za pomocą dwóch instrukcji: *rotate with extend* (*obróć z rozszerzeniem*) w prawo o jedną pozycję oraz logicznego *przesunięcia w prawo* o trzy pozycje (instrukcja roxr w rzeczywistości korzysta z bitu X, ale instrukcja add ustawia jego wartość na taką samą, jak wartość bitu przeniesienia). W przypadku większości maszyn jest jednak wymaga większa liczba instrukcji. Przykładowo na procesorze PowerPC potrzeba trzech instrukcji: wyczyszczenie trzech skrajnie prawych bitów q, dodanie bitu przeniesienia do q oraz obrócenie w prawo o trzy pozycje.

Po zaimplementowaniu w pewien sposób instrukcji shrxi przedstawiony powyżej kod oblicza

$$q = \left\lfloor \left(\left\lfloor \left(\frac{2^{35}+3}{7} - 2^{32} \right) \frac{n}{2^{32}} \right\rfloor + n \right) / 2^3 \right\rfloor = \left\lfloor \frac{n}{7} + \frac{3n}{7 \cdot 2^{35}} \right\rfloor.$$

Dla $0 \le n < 2^{32}$ zachodzi $0 \le 3n / (7 \cdot 2^{35}) < 1/7$, tak więc, na mocy twierdzenia D4, $q = \lfloor n / 7 \rfloor$.

Granlund i Montgomery [GM] opracowali ciekawy schemat postępowania, pozwalający na uniknięcie stosowania instrukcji shrxi. Wymaga on tej samej liczby instrukcji, co przedstawiona powyżej sekwencja trzech instrukcji tworzących shrxi, ale korzysta jedynie z elementarnych instrukcji, które oferują niemal wszystkie procesory i w ogóle nie powoduje powstawania przepełnienia. Wykorzystywaną tożsamością jest

$$\frac{q+n}{2^p} = \left\lfloor \left(\left\lfloor \frac{n-q}{2} \right\rfloor + q \right) / 2^{p-1} \right\rfloor, \quad p \ge 1.$$

Zastosowanie jej do przedstawionego problemu przy $q = \lfloor Mn / 2^{32} \rfloor$, gdzie $0 \le M < 2^{32}$, sprawia, że odejmowanie nie powoduje powstawania przepełnienia, ponieważ

$$n - q = n - \left\lfloor \frac{Mn}{2^{32}} \right\rfloor = \left\lceil n - \frac{Mn}{2^{32}} \right\rceil = \left\lceil n\left(1 - \frac{Mn}{2^{32}}\right) \right\rceil,$$

tak więc oczywiście $0 \le n - q < 2^{32}$. Ponadto operacja *dodawania* także nie powoduje przepełnienia, gdyż

$$\left\lfloor \frac{n-q}{2} \right\rfloor + q = \left\lfloor \frac{n-q}{2} + q \right\rfloor = \left\lfloor \frac{n+q}{2} \right\rfloor$$

i $0 \le n, q < 2^{32}$.

Wykorzystanie przedstawionej koncepcji prowadzi do poniższego kodu dzielenia bez znaku przez 7:

```
li    M,0x24924925  Załadowanie liczby magicznej (2**33+1)/3
mulhu q,M,n         q = floor(M*n/2**32).
sub   t,n,q         t = n - q.
shri  t,t,1         t = (n - q)/2.
add   t,t,q         t = (n - q)/2 + q = (n + q)/2.
shri  q,t,2         q = (n+Mn/2**32)/8 = floor(n/7).

muli  t,q,7         Obliczenie reszty ze wzoru
sub   r,n,t         r = n - q*7.
```

Aby przedstawiony schemat działał poprawnie, liczba pozycji do przesunięcia w ramach hipotetycznej instrukcji `shrxi` musi być większa od 0. Można wykazać, że jeżeli $d > 1$ a mnożnik $m \ge 2^{32}$ (zatem potrzebna jest instrukcja `shrxi`), to liczba pozycji do przesunięcia jest większa od 0.

10.9. Dzielenie bez znaku przez dzielniki ≥ 1

Mając dany rozmiar słowa $W \ge 1$ oraz dzielnik d, taki że $1 \le d < 2^W$, chcemy znaleźć najmniejszą liczbę całkowitą m oraz liczbę całkowitą p, takie że

$$\left\lfloor \frac{mn}{2^p} \right\rfloor = \left\lfloor \frac{n}{d} \right\rfloor, \text{ dla } 0 \le n < 2^W \tag{22}$$

Gdzie $0 \le m < 2^{W+1}$ oraz $p \ge W$.

W przypadku dzielenia bez znaku liczbę magiczną M określa zależność

$$M = \begin{cases} m, & \text{kiedy } 0 \le m < 2^W \\ m - 2^W, & \text{kiedy } 2^W \le m < 2^{W+1}. \end{cases}$$

Z uwagi na fakt, że zależność (22) musi być spełniona dla $n = d$, zachodzi związek $\lfloor md / 2^p \rfloor = 1$, z czego wynika

$$\frac{md}{2^p} \geq 1. \tag{23}$$

Podobnie jak w przypadku dzielenia ze znakiem, niech n_c będzie największą wartością n, taką że $\mathrm{rem}(n_c, d) = d - 1$. Wartość n_c można obliczyć na podstawie zależności $n_c = \lfloor 2^W / d \rfloor d - 1 = 2^W - \mathrm{rem}(2^W, d) - 1$. Wówczas

$$2^W - d \leq n_c \leq 2^W - 1 \tag{24a}$$

oraz

$$n_c \geq d - 1. \tag{24b}$$

Implikuje to, że $n_c \geq d - 1$.

Z uwagi na fakt, że zależność (22) musi być spełniona dla $n = n_c$ wiadomo, że

$$\left\lfloor \frac{mn_c}{2^p} \right\rfloor = \left\lfloor \frac{n_c}{d} \right\rfloor = \frac{n_c - (d-1)}{d}$$

lub

$$\frac{mn_c}{2^p} < \frac{n_c + 1}{d}.$$

Połączenie tych zależności z nierównością (23) daje

$$\frac{2^p}{d} \leq m < \frac{2^p}{d} \frac{n_c + 1}{n_c}. \tag{25}$$

Ponieważ wartość m ma być najmniejszą liczbą całkowitą spełniającą zależność (25), jest ona najmniejszą liczbą całkowitą większą lub równą $2^p / d$, to znaczy

$$\boxed{m = \frac{2^p + d - 1 - \mathrm{rem}(2^p - 1, d)}{d}.} \tag{26}$$

W połączeniu z prawą stroną zależności (4), upraszczając wyrażenie, otrzymujemy

$$\boxed{2^p > n_c(d - 1 - \mathrm{rem}(2^p - 1, d)).} \tag{27}$$

Algorytm (dzielenia bez znaku)

Algorytm polega na znalezieniu metodą prób i błędów najmniejszego $p \geq W$, spełniającego nierówność (27). Wówczas m oblicza się ze wzoru (26). Jest to najmniejsza dopuszczalna wartość m spełniająca równanie (22) dla $p \geq W$. Podobnie jak w przypadku

dzielenia ze znakiem, jeżeli nierówność (27) jest spełniona dla pewnej wartości p, to jest spełniona dla wszystkich większych wartości p. Dowód jest dokładnie taki sam, jak w przypadku twierdzenia DC1, z tą różnicą, że korzysta się z twierdzenia D5(b) zamiast twierdzenia D5(a).

Dowód wykonalności algorytmu (dzielenia bez znaku)

Należy wykazać, że nierówność (27) zawsze ma rozwiązanie oraz że $0 \le m < 2^{W+1}$.

Dla każdej nieujemnej liczby całkowitej x istnieje potęga liczby 2 większa od x oraz mniejsza lub równa $2x + 1$, zatem na podstawie nierówności (27) można określić, że

$$n_c(d - 1 - \mathrm{rem}(2^p - 1, d)) < 2^p \le 2n_c(d - 1 - \mathrm{rem}(2^p - 1, d)) + 1.$$

Z uwagi na fakt, że $0 \le \mathrm{rem}(2^{p-1}, d) \le d - 1$

$$1 \le 2^p \le 2n_c(d - 1) + 1. \tag{28}$$

$n_c, d \le 2^{W-1} - 1$, więc nierówność (28) przyjmuje postać

$$1 \le 2^p \le 2(2^W - 1)(2^W - 2) + 1$$

lub

$$0 \le p \le 2W. \tag{29}$$

Zatem nierówność (27) zawsze ma rozwiązanie.

Jeżeli p nie musi być równe W, wówczas na podstawie nierówności (25) i (28) można zapisać

$$\frac{1}{d} \le m < \frac{2n_c(d-1)+1}{d}\frac{n_c+1}{n_c},$$

$$1 \le m < \frac{2d - 2 + 1/n_c}{d}(n_c + 1),$$

$$1 \le m < 2(n_c + 1) \le 2^{W+1}.$$

Jeżeli p musi być równe W, wówczas na podstawie nierówności (25) można zapisać

$$\frac{2^W}{d} \le m < \frac{2^W}{d}\frac{n_c+1}{n_c}.$$

$1 \le d \le 2^W - 1$ oraz $n_c \ge 2^{W-1}$, a więc

$$\frac{2^W}{2^W - 1} \le m < \frac{2^W}{1}\frac{2^{W-1}+1}{2^{W-1}},$$

$$2 \le m \le 2^W + 1.$$

Stąd w obu przypadkach m należy do przedziału ograniczonego schematem kodu przedstawionego w przykładzie dzielenia bez znaku przez 7.

Dowód poprawności wyniku (dzielenia bez znaku)

Należy wykazać, że jeżeli p i m zostaną obliczone na podstawie zależności (27) i (26), to będzie spełnione równanie (22).

Równanie (26) oraz nierówność (27) w oczywisty sposób implikują nierówność (25). Nierówność ta ma niemal taką samą postać, co nierówność (4) a reszta dowodu jest niemal identyczna, jak w przypadku dzielenia ze znakiem dla $n \geq 0$.

10.10. Zawieranie obsługi w kompilatorze

Istnieje pewna trudność w przypadku implementacji algorytmu opartego na bezpośrednim obliczaniu wyrażeń użytych w dowodzie. Chociaż $p \leq 2^W$, co dowiedziono powyżej, może wystąpić przypadek $p = 2^W$ (np. dla $d = 2^W - 2$, gdzie $W \geq 4$). Kiedy $p = 2^W$, powstaje trudność w obliczeniu wartości m, ponieważ dzielna ze wzoru (26) jest za duża, aby móc ją reprezentować jako 2^W-bitowe słowo.

Jednakże można go zaimplementować przy użyciu techniki „dzielenia przyrostowego z resztą" algorytmu *magic*. Algorytm podano na listingu 10.2 dla $W = 32$. Przekazuje on znacznik `a`, który określa, czy należy wygenerować instrukcję `add` (w przypadku dzielenia ze znakiem program wywołujący rozpoznaje to na podstawie wartości `M` i `d`, które mają przeciwne znaki).

Listing 10.2. *Obliczanie liczby magicznej dla dzielenia bez znaku*

```
struct mu {unsigned M;      // Liczba magiczna,
           int a;           // znacznik "add"
           int s;};         // oraz liczba pozycji przesunięcia.

struct mu magicu(unsigned d) {
                            // Musi zachodzić 1 <= d <= 2**32-1.
   int p;
   unsigned nc, delta, q1, r1, q2, r2;
   struct mu magu;

   magu.a = 0;              // Inicjalizacja znacznika "add".
   nc = - 1 - (-d)%d;       // Arytmetyka bez znaku.
   p = 31;                  // Inicjalizacja p.
   q1 = 0x80000000/nc;      // Inicjalizacja q1 = 2**p/nc.
   r1 = 0x80000000 - q1*nc;// Inicjalizacja r1 = rem(2**p, nc).
   q2 = 0x7FFFFFFF/d;       // Inicjalizacja q2 = (2**p -1 )/d.
   r2 = 0x7FFFFFFF - q2*d; // Inicjalizacja r2 = rem(2**p -1, d).
   do {
      p = p + 1;
      if (r1 >= nc - r1) {
         q1 = 2*q1 + 1;              // Modyfikacja q1.
         r1 = 2*r1 - nc;             // Modyfikacja r1.
```

```
        else {
           q1 = 2*q1;
           r1 = 2*r1;}
        if (r2 + 1 >= d - r2) {
           if (q2 >= 0x7FFFFFFF) magu.a = 1;
           q2 = 2*q2 + 1;              // Modyfikacja q2.
           r2 = 2*r2 + 1 - d;}         // Modyfikacja r2.
        else {
           if (q2 >= 0x80000000) magu.a = 1;
           q2 = 2*q2;
           r2 = 2*r2 + 1;}
        delta = d - 1 - r2;
     } while (p < 64 &&
             (q1 < delta || (q1 == delta && r1 == 0)));

     magu.M = q2 + 1;          // Zwracana liczba magiczna oraz
     magu.s = p - 32;          // liczba pozycji przesunięcia.
     return magu;              // (magu.a) ustawiono wcześniej.
  }
```

Najważniejsze uwagi, pomagające w zrozumieniu tego algorytmu to:

- przepełnienie bez znaku może wystąpić w kilku miejscach i powinno być ignorowane;
- $n_c = 2^W - \mathrm{rem}(2^W, d) - 1 = (2^W - 1) - \mathrm{rem}(2^W - d, d)$;
- iloraz i reszta z dzielenia 2^p przez n_c nie mogą być aktualizowane w ten sam sposób, jak miało to miejsce w przypadku algorytmu *magic*, ponieważ tym razem wielkość `2*r1` może spowodować przepełnienie. Stąd algorytm zawiera sprawdzenie `if (r1 >= nc - r1)`, choć bardziej naturalnym zapisem byłoby `if (2*r1 >= nc)`. Podobna uwaga ma zastosowanie wobec obliczania ilorazu i reszty z dzielenia $2^p - 1$ przez d;
- $0 \le \delta \le d - 1$, tak więc δ można reprezentować jako 32-bitową liczbę całkowitą bez znaku;
- $m = (2^p + d - 1 - \mathrm{rem}(2^p - 1, d)) / d = \lfloor (2^p - 1) / d \rfloor + 1 = q_2 + 1$;
- odejmowanie 2^W w momencie, gdy liczba magiczna `M` przekroczy wartość $2^W - 1$ nie zostało jawnie zawarte w programie. Występuje ono, kiedy obliczenie `q2` powoduje przepełnienie;
- znacznik „add", `magu.a`, nie może zostać określony na podstawie prostego porównania `M` z 2^{32} lub `q2` z $2^{32} - 1$ z uwagi na występowanie przepełnienia. Zamiast tego program sprawdza wartość `q2` zanim może nastąpić przepełnienie. Jeżeli `q2` kiedykolwiek osiąga wartość $2^{32} - 1$, tak że `M` jest większe lub równe 2^{32}, wartość `magu.a` zostaje ustawiona na 1. Jeżeli wartość `q2` pozostaje poniżej $2^{32} - 1$, wartość `magu.a` pozostaje na domyślnym poziomie 0;
- nierówność (27) jest równoważna wyrażeniu $2^p / n_c > \delta$;
- sprawdzenie warunku pętli wymaga zapisu `p < 64`, ponieważ bez niego przepełnienie `q1` spowodowałoby wykonanie zbyt dużej liczby iteracji pętli, powodując otrzymanie błędnych wyników.

10.11. Inne zagadnienia (dzielenia bez znaku)

twierdzenie

TWIERDZENIE DC2U

Najmniejszy mnożnik m jest nieparzysty, jeżeli p nie musi być równe W.

twierdzenie

TWIERDZENIE DC3U

Dla danego dzielnika d istnieje tylko jeden mnożnik m, charakteryzujący się minimalną wartością p, jeżeli p nie musi być równe W.

Dowody powyższych twierdzeń są bardzo podobne do odpowiednich dowodów dla dzielenia ze znakiem.

Dzielniki w najlepszych programach

W przypadku dzielenia bez znaku w celu znalezienia dzielników (o ile istnieją) za pomocą optymalnych programów, składających się z dwóch instrukcji obliczających iloraz (`li`, `mulhu`), można dokonać analizy podobnej do przedstawionej w przypadku dzielenia ze znakiem (patrz podrozdział *Dzielniki w najlepszych programach* na stronie 202). Wynika z tego, że takie dzielniki są czynnikami rozkładu wartości 2^W lub $2^W + 1$, oprócz przypadku $d = 1$. Dla najczęściej używanych rozmiarów słów pozostawia to bardzo niewiele niebanalnych dzielników, które posiadają optymalne programy dla dzielenia bez znaku. Dla $W = 16$ nie istnieją żadne. Dla $W = 32$ istnieją tylko dwa: 641 oraz 6 700 417. Dla $W = 64$ także istnieją tylko dwa: 274 177 oraz 67 280 421 310 721.

Przypadek $d = 2^k$, $k = 1, 2, \ldots$ zasługuje na szczególną uwagę. W tym przypadku algorytm *magicu* zwraca $p = W$ (wymuszone), $m = 2^{32-k}$. Jest to minimalna wartość m, ale nie jest to minimalna wartość M. Lepsze wyniki, jeżeli użyje się $p = W + k$, zostaną zwrócone po wykonaniu odpowiednich uproszczeń. Wówczas $m = 2^W$, $M = 0$, $a = 1$ oraz $s = k$. Generowany kod zawiera mnożenie przez 0 i może zostać uproszczony do pojedynczej instrukcji *shift right k* (*przesuń w prawo o k pozycji*). Praktyczne znaczenie ma fakt, że dzielniki będące potęgami liczby 2 prawdopodobnie stanowiłyby przypadki szczególne, gdyby nie używać algorytmu *magicu*. To zjawisko nie występuje w przypadku dzielenia ze znakiem, ponieważ wówczas m nie może być potęgą 2. *Dowód*: dla $d > 0$ nierówność (4) w połączeniu z (3b) implikuje, że $d - 1 < 2^p / m < d$. Stąd $2^p / m$ nie może być liczbą całkowitą. Dla $d < 0$ rozumowanie przebiega podobnie dla nierówności (16) oraz (15b).

W przypadku dzielenia bez znaku kod dla $m \geq 2^W$ jest znacząco słabszy od kodu dla $m < 2^W$, jeżeli maszyna nie obsługuje instrukcji `shrxi`. Stąd istotne znaczenie ma wyobrażenie o tym, jak często pojawiają się duże mnożniki. Dla $W = 32$ wśród liczb całkowitych mniejszych od 100 występuje 31 „złych" dzielników: 1, 7, 14, 19, 21, 27, 28, 31, 35, 37, 38, 39, 42, 45, 53, 54, 55, 56, 57, 62, 63, 70, 73, 74, 76, 78, 84, 90, 91, 95 oraz 97.

Użycie mnożenia ze znakiem zamiast mnożenia bez znaku i odwrotnie

Jeśli maszyna nie obsługuje instrukcji `mulhu`, ale obsługuje instrukcję `mulhs` (lub długie mnożenie ze znakiem), sztuczka przedstawiona w podrozdziale *Konwersje między bardziej znacząca połową 64-bitowego iloczynu ze znakiem i bez znaku* na stronie 155 może sprawić, że przedstawiona metoda dzielenia bez znaku przez stałą wciąż będzie użyteczna.

We wspomnianym podrozdziale przedstawiono sekwencję siedmiu instrukcji, umożliwiającą wykonanie operacji `mulhu` przy użyciu instrukcji `mulhs`. W omawianym przypadku zostanie ona nieco uproszczona, gdyż z góry znana jest liczba magiczna *M*. Zatem kompilator może sprawdzić najbardziej znaczący bit liczby magicznej i wygenerować kod podobny do poniższego dla operacji `mulhu q,M,n`. Symbol `t` oznacza rejestr tymczasowy.

```
  M31 = 0          M31 = 1
mulhs q,M,n      mulhs q,M,n
shrsi t,n,31     shrsi t,n,31
and   t,t,M      and   t,t,M
add   q,q,t      add   t,t,n
                 add   q,q,t
```

Biorąc pod uwagę inne instrukcje używane z instrukcją `mulhu`, w sumie wykorzystywanych jest sześć do ośmiu instrukcji w celu otrzymania ilorazu odpowiadającego dzieleniu bez znaku przez stałą na maszynie, która nie obsługuje bezpośrednio mnożenia bez znaku.

Można również postąpić odwrotnie, otrzymując instrukcję `mulhs` z `mulhu`. Kod jest identyczny jak powyżej, z tą różnicą, że instrukcja `mulhs` zostaje zamieniona na `mulhu` a końcowa instrukcja `add` w obu kolumnach zostaje zamieniona na `sub`.

Prostszy algorytm (dzielenia bez znaku)

Odrzucenie ograniczenia określającego, że liczba magiczna musi być minimalna, pozwala na otrzymanie prostszej wersji algorytmu. Zamiast nierówności (27) można użyć

$$2^p > 2^W (d - 1 - \mathrm{rem}(2^p - 1, d)), \tag{30}$$

a następnie, jak poprzednio, wykorzystać równanie (26) w celu obliczenia *m*.

Należy wyjaśnić, że formalnie algorytm ten jest poprawny (to znaczy obliczona wartość *m* spełnia równanie (22)), ponieważ jedyna różnica między nim a poprzednim algorytmem polega na tym, że tym razem obliczana dla pewnego *d* wartość *p* niekoniecznie musi być duża. Można udowodnić, że wartość *m* obliczona na podstawie zależności (30) i (26) jest mniejsza od 2^{W+1}. Pominiemy jednak ten dowód i przedstawimy po prostu algorytm (listing 10.3).

Listing 10.3. *Uproszczony algorytm obliczania liczby magicznej dla dzielenia bez znaku*

```
struct mu {unsigned M;      // Liczba magiczna,
           int a;           // znacznik "add"
           int s;};         // oraz liczba pozycji przesunięcia.

struct mu magicu2(unsigned d) {
                            // Musi zachodzić 1 <= d <= 2**32-1.
   int p;
   unsigned p32, q, r, delta;
   struct mu magu;

   magu.a = 0;              // Inicjalizacja znacznika "add".
   p = 31;                  // Inicjalizacja p.
   q = 0x7FFFFFFF/d;        // Inicjalizacja q = (2**p - 1)/d.
   r = 0x7FFFFFFF - q*d;    // Inicjalizacja r = rem(2**p - 1, d).
   do {
      p = p + 1;
      if (p == 32) p32 = 1;          // Ustawienie p32 = 2**(p-32).
      else p32 = 2*p32;
      if (r + 1 >= d - r) {
         if (q >= 0x7FFFFFFF) magu.a = 1;
         q = 2*q + 1;               // Modyfikacja q.
         r = 2*r + 1 - d;           // Modyfikacja r.
      }
      else {
         if (q2 >= 0x80000000) magu.a = 1;
         q = 2*q;
         r = 2*r + 1;
      }
      delta = d - 1 - r;
   } while (p < 64 && p32 < delta);

   magu.M = q + 1;          // Zwracana liczba magiczna oraz
   magu.s = p - 32;         // liczba pozycji przesunięcia.
   return magu;             // (magu.a) ustawiono wcześniej.
}
```

Alverson [Alv] podał znacznie prostszy algorytm, omówiony w jednym z kolejnych podrozdziałów, ale daje on dość duże wartości m. Istota algorytmu *magicu2* polega na tym, że niemal zawsze daje on minimalną wartość m, kiedy $d \le 2^{W-1}$. Dla $W = 32$ najmniejszy dzielnik, dla którego algorytm *magicu2* nie daje minimalnego mnożnika, to $d = 102\,807$, gdzie *magicu* oblicza $m = 2\,737\,896\,999$, zaś *magicu2* — $m = 5\,475\,793\,997$.

Istnieje odpowiednik algorytmu *magicu2* dla dzielenia ze znakiem przez dodatnie dzielniki, ale nie działa on zbyt dobrze dla dzielenia ze znakiem przez dowolne dzielniki.

10.12. Zastosowalność w przypadku dzielenia z modułem i dzielenia z funkcją podłoga

Może się wydawać, że zamiana dzielenia z modułem lub dzielenia z funkcją *podłoga* przez stałą na operację *mnożenia* byłaby prostsza o tyle, że krok „dodaj 1, jeżeli dzielna jest ujemna" można by pominąć. Jednakże nie jest tak. Przedstawione powyżej metody nie mogą być stosowane w żaden oczywisty sposób w przypadku dzielenia z modułem lub z funkcją *podłoga*. Być może dałoby się opracować pewien algorytm — zapewne wiązałby się on z pewnymi modyfikacjami mnożnika m w zależności od znaku dzielnej.

10.13. Podobne metody

Zamiast zapisywać algorytm *magic*, można utworzyć tabelę, określającą liczby magiczne oraz liczby pozycji przesunięć dla kilku małych dzielników. Dzielniki równe dzielnikom stabularyzowanym pomnożonym przez potęgę 2 można z łatwością obsługiwać w następujący sposób:

1. Zliczamy liczbę końcowych zer w d i oznaczamy ją jako k.
2. Jako argumentu wyszukiwania używamy $d / 2^k$ (przesunięcie w prawo o k pozycji).
3. Używamy liczby magicznej odnalezionej w tabeli.
4. Używamy liczby pozycji przesunięcia znalezionej w tabeli i zwiększamy ją o k.

Zatem, jeżeli tabela zawiera dzielniki 3, 5, 25 i tak dalej, można obsłużyć dzielniki 6, 10, 100 i tak dalej.

Opisana procedura zwykle daje najmniejszą liczbę magiczną ale nie zawsze. Najmniejszy dodatni dzielnik, dla którego nie udaje się tego zrobić dla $W = 32$ to $d = 334\ 972$, kiedy otrzymuje się wartości $m = 3\ 361\ 176\ 179$ oraz $s = 18$. Jednak minimalna liczba magiczna dla $d = 334\ 972$ to $m = 840\ 294\ 045$, gdzie $s = 16$. Procedura ta kończy się również niepowodzeniem w przypadku $d = -6$. W obu przypadkach spada jakość kodu wynikowego.

Alverson [Alv] jest pierwszą osobą znaną Autorowi, która stwierdziła, że opisana poniżej metoda działa poprawnie dla wszystkich dzielników. Używając wprowadzonej notacji, jego metodę dzielenia liczb całkowitych bez znaku przez d możemy opisać jako ustalenie liczby pozycji przesunięcia $p = W + \lceil \log_2 d \rceil$ oraz mnożnika $m = \lceil 2^p / d \rceil$, a następnie wykonanie dzielenia przez $n \div d = \lfloor mn / 2^p \rfloor$ (to znaczy mnożenia i przesunięcia w prawo). Alverson udowodnił, że mnożnik m jest mniejszy od 2^{W+1} oraz że metoda pozwala otrzymać dokładny iloraz dla wszystkich n możliwych do wyrażenia za pomocą W bitów.

Metoda Alversona stanowi odmianę metody zaprezentowanej w poprzednim podrozdziale. Jest ona prostsza o tyle, że nie wymaga określania metodą prób i błędów wartości p a dzięki temu jest odpowiedniejsza do wykorzystania w rozwiązaniach sprzętowych, co stanowi jej główną sferę zastosowań. W tym przypadku jednak mnożnik m jest zawsze większy lub równy 2^W, a zatem w przypadku rozwiązań programowych zawsze daje kod przedstawiony w przykładzie dzielenia przez 7 (to znaczy, zawsze zawiera instrukcje `add` i `shrxi` lub alternatywne rozwiązanie oparte na czterech instrukcjach). Większość małych dzielników można obsłużyć przy użyciu mnożnika mniejszego od 2^W, zatem godne uwagi wydaje się być wyszukanie tych przypadków.

W przypadku dzielenia ze znakiem Alverson zaleca znalezienie mnożnika dla $|d|$ oraz słowa o długości $W-1$ (wówczas $2^{W-1} \le m < 2^W$), pomnożenie go przez dzielną oraz zanegowanie wyniku, jeżeli operandy mają przeciwne znaki (mnożnik musi dawać poprawny wynik, kiedy dzielna ma wartość 2^{W-1}, stanowiącą wartość bezwzględną maksymalnej liczby ujemnej). Wydaje się możliwe, że owa sugestia może dać lepszy kod od przedstawionego wcześniej dla przypadku mnożnika $m \ge 2^W$. Zastosowanie jej w przypadku dzielenia ze znakiem przez 7 daje kod przedstawiony poniżej, gdzie skorzystano z relacji $-x = \bar{x} + 1$ w celu uniknięcia rozgałęzienia:

```
abs   an,n
li    M,0x92492493  Liczba magiczna, (2**34+5)/7.
mulhu q,M,an        q = floor(M*an/2**32).
shri  q,q,2
shrsi t,n,31        Te trzy instrukcje
xor   q,q,t         negują q, jeżeli
sub   q,q,t         n jest ujemne.
```

Nie jest on tak dobry, jak kod przedstawiony dla dzielenia przez 7 ze znakiem (tam sześć, tu siedem instrukcji), ale jest on przydatny w przypadku maszyn, które oferują instrukcje `abs` i `mulhu` a nie oferują `mulhs`.

10.14. Przykładowe liczby magiczne

Tabela 10.1. *Niektóre liczby magiczne dla W = 32*

	Ze znakiem		Bez znaku		
d	**M (heks.)**	**s**	**M (heks.)**	**a**	**s**
-5	99999999	1			
-3	55555555	1			
-2^k	7FFFFFFF	$k-1$			
1	-	–	0	1	0
2^k	80000001	$k-1$	2^{32-k}	0	0
3	55555556	0	AAAAAAAB	0	1
5	66666667	1	CCCCCCCD	0	2

Tabela 10.1. *Niektóre liczby magiczne dla W = 32 (ciąg dalszy)*

	Ze znakiem		Bez znaku		
d	**M (heks.)**	**s**	**M (heks.)**	**a**	**s**
6	2AAAAAAB	0	AAAAAAAB	0	2
7	92492493	2	24924925	1	3
9	38E38E39	1	38E38E39	0	1
10	66666667	2	CCCCCCCD	0	3
11	2E8BA2E9	1	BA2E8BA3	0	3
12	2AAAAAAB	1	AAAAAAAB	0	3
25	51EB851F	3	51EB851F	0	3
125	10624DD3	3	10624DD3	0	3
625	68DB8BAD	8	D1B71759	0	9

Tabela 10.2. *Niektóre liczby magiczne dla W = 64*

	Ze znakiem		Bez znaku		
d	**M (heks.)**	**s**	**M (heks.)**	**a**	**s**
–5	9999999999999999	1			
–3	5555555555555555	1			
-2^k	7FFFFFFFFFFFFFFF	$k-1$			
1	-	–	0	1	0
2^k	8000000000000001	$k-1$	2^{32-k}	0	0
3	5555555555555556	0	AAAAAAAAAAAAAAAB	0	1
5	6666666666666667	1	CCCCCCCCCCCCCCCD	0	2
6	2AAAAAAAAAAAAAAB	0	AAAAAAAAAAAAAAAB	0	2
7	4924924924924925	1	2492492492492493	1	3
9	1C71C71C71C71C72	0	E38E38E38E38E38F	0	3
10	6666666666666667	2	CCCCCCCCCCCCCCCD	0	3
11	2E8BA2E8BA2E8BA3	1	2E8BA2E8BA2E8BA3	0	1
12	2AAAAAAAAAAAAAAB	1	AAAAAAAAAAAAAAAB	0	3
25	A3D70A3D70A3D70B	4	47AE147AE147AE15	1	5
125	20C49BA5E353F7CF	4	0624DD2F1A9FBE77	1	7
625	346DC5D63886594B	7	346DC5D63886594B	0	7

10.15. Dzielenie dokładne przez stałe

Przez *dzielenie dokładne* (ang. *exact division*) będziemy rozumieć dzielenie, o którym (w pewien sposób) z góry wiadomo, że otrzymana reszta będzie równa 0. Choć taka

sytuacja nie występuje zbyt często, to jednak można znaleźć odpowiednie przykłady, chociażby odejmowanie dwóch wskaźników w języku C. Wynik działania $p - q$, gdzie p i q to wskaźniki, jest dobrze zdefiniowany i przenośny tylko wtedy, gdy p i q wskazują na obiekty w tej samej tablicy [H&S, sec. 7.6.2]. Jeżeli rozmiar elementów tablicy wynosi s, kod dla różnicy $p - q$ oblicza wartość $(p - q) / s$.

Na treść niniejszego podrozdziału główny wpływ miała praca [GM, sec. 9].

Prezentowana metoda ma zastosowanie zarówno w przypadku dzielenia dokładnego ze znakiem, jak i bez znaku i opiera się na następującym twierdzeniu.

TWIERDZENIE M1

twierdzenie

Jeżeli a i m są względnie pierwszymi liczbami całkowitymi, to istnieje liczba całkowita $\overline{a}$, $1 \le \overline{a} < m$, taka że

$$a\overline{a} \equiv 1 \pmod{m}.$$

Zatem $\overline{a}$ jest multiplikatywną odwrotnością a, modulo m. Istnieje kilka sposobów udowodnienia tego twierdzenia; trzy dowody podano w pozycji [NZM, 52]. Poniższy dowód wymaga tylko podstawowej znajomości zagadnienia kongruencji.

Dowód. Udowodnimy bardziej ogólne założenie niż tylko twierdzenie. Jeżeli a i m są względnie pierwszymi liczbami całkowitymi (a stąd są różne od zera), to wraz z przyjmowaniem przez x kolejno wszystkich m wartości modulo m, ax przyjmuje wszystkie m wartości modulo m. Przykładowo, jeżeli $a = 3$ i $m = 8$, to w miarę jak x przyjmuje wartości od 0 do 7, $ax = 0, 3, 6, 9, 12, 15, 18, 21$ lub, po zredukowaniu modulo 8, $ax = 0, 3, 6, 1, 4, 7, 2, 5$. Warto zauważyć, że w ostatnim ciągu występują wszystkie wartości od 0 do 7.

W celu przyjrzenia się przypadkowi ogólnemu, załóżmy że twierdzenie jest fałszywe. Wówczas istnieją oddzielne wartości całkowitoliczbowe, które są odwzorowywane na tę samą wartość w przypadku mnożenia przez a; oznacza to, że istnieją liczby x i y, gdzie $x \ne y \pmod{m}$, takie że

$$ax \equiv ay \pmod{m}.$$

Jednak wówczas istnieje liczba całkowita k, taka że

$$ax - ay = km, \quad \text{czyli}$$
$$a(x - y) = km.$$

a nie ma żadnego czynnika rozkładu takiego jak m, zatem musi być spełniona zależność określająca, że $x - y$ jest wielokrotnością m. To znaczy

$$x \equiv y \pmod{m}.$$

To jednak stoi w sprzeczności z postawioną hipotezą.

ax przyjmuje wszystkie m różnych wartości modulo m wraz z przyjmowaniem przez x kolejnych m wartości, więc dla pewnego x musi przyjąć wartość 1.

Dowód pokazuje, że istnieje tylko jedna wartość (modulo m) x, dla której $ax \equiv 1 \pmod{m}$ — to znaczy, że multiplikatywna odwrotność jest unikatowa, oprócz addytywnych wielokrotności m. Pokazuje także, że istnieje unikatowa (modulo m) liczba całkowita x, taka że $ax \equiv b \pmod{m}$, gdzie b jest dowolną liczbą całkowitą.

Jako przykład rozpatrzymy przypadek $m = 16$. Wówczas $\overline{3} = 11$, ponieważ $3 * 11 = 33 \equiv 1 \pmod{16}$. Można by równie dobrze przyjąć $\overline{3} = -5$, ponieważ $3 * (-5) = -15 \equiv 1 \pmod{16}$. Podobnie $-\overline{3} = 5$, ponieważ $(-3) * 5 = -15 \equiv 1 \pmod{16}$.

Obserwacje te są istotne, gdyż pokazują że wprowadzone pojęcia mają zastosowanie zarówno w przypadku liczb ze znakiem, jak i bez znaku. W przypadku działania na liczbach całkowitych bez znaku na maszynie 4-bitowej przyjmujemy $\overline{3} = 11$. W przypadku liczb całkowitych ze znakiem przyjmujemy $\overline{3} = -5$. Jednak liczby 11 i -5 mają taką samą reprezentację w notacji dopełnienia dwójkowego (ponieważ różnią się o 16), tak więc ta sama zawartość słów w komputerze może być używana w obu przypadkach jako multiplikatywna odwrotność.

Twierdzenie ma bezpośrednie zastosowanie do problemu dzielenia (ze znakiem i bez znaku) przez nieparzystą liczbę całkowitą d na komputerze W-bitowym. Ponieważ każda nieparzysta liczba całkowita jest względnie pierwsza do 2^W, twierdzenie określa, że jeżeli d jest nieparzyste, istnieje liczba całkowita $\overline{d}$ (unikatowa w przedziale od 0 do $2^W - 1$ lub w przedziale od -2^{W-1} do $2^{W-1} - 1$), taka że

$$d\overline{d} \equiv 1 \pmod{2^W}.$$

Stąd dla dowolnej liczby całkowitej n, która jest wielokrotnością d, zachodzi

$$\frac{n}{d} \equiv \frac{n}{d}(d\overline{d}) \equiv n\overline{d} \pmod{2^W}.$$

Innymi słowy, n / d można obliczyć mnożąc n przez $\overline{d}$ i zachowując tylko W prawych bitów iloczynu.

Jeżeli dzielnik d jest parzysty, niech $d = d_0 \cdot 2^k$, gdzie d_0 jest nieparzyste oraz $k \geq 1$. Następnie przesuwamy po prostu n w prawo o k pozycji (pozbywając się zer), a następnie mnożymy przez d_0 (przesunięcie można równie dobrze wykonać po wykonaniu mnożenia).

Poniżej przedstawiono kod dzielenia n przez 7, gdzie n jest wielokrotnością 7. Kod ten daje poprawny wynik zarówno w przypadku dzielenia ze znakiem, jak i bez znaku.

```
li    M,0xB6DB6DB7  Multiplikatywna odwrotność, (5*2**32 + 1)/7.
mul   q,M,n         q = n/7.
```

Obliczanie multiplikatywnej odwrotności za pomocą algorytmu Euklidesa

Pojawia się pytanie, w jaki sposób można obliczać multiplikatywną odwrotność. Standardową metodą jest wykorzystanie „rozszerzonego algorytmu Euklidesa". Poniżej zawarto pobieżny opis interesujących nas zagadnień, ale bardziej dociekliwy Czytelnik może sięgnąć po pozycję [NZM, 13] oraz [Knu2, podrozdz. 4.5.2], gdzie znajdzie szczegółowy opis.

Przy danym nieparzystym dzielniku d chcemy znaleźć x na podstawie zależności

$$dx \equiv 1 \pmod{m},$$

gdzie (w naszym przypadku) $m = 2^W$ oraz W jest rozmiarem słowa używanym przez daną maszynę. Można tego dokonać, jeżeli szukając liczb całkowitych x i y (dodatnich, ujemnych lub równych 0) rozwiąże się równanie

$$dx + my = 1.$$

W tym celu najpierw zapewniamy, aby wartość d była dodatnia, dodając do niej wystarczającą liczbę wielokrotności m (liczby d oraz $d + km$ mają tę samą multiplikatywną odwrotność). Następnie zapisujemy poniższe równania (gdzie $d, m > 0$):

$$d(-1) + m(1) = m - d \qquad \text{(i)}$$
$$d(1) + m(0) = d. \qquad \text{(ii)}$$

Jeżeli $d = 1$, kończy to wykonywane działania, ponieważ równanie (ii) pokazuje, że $x = 1$. W przeciwnym wypadku obliczamy

$$q = \left\lfloor \frac{m-d}{d} \right\rfloor.$$

Następnie mnożymy równanie (ii) przez q i odejmujemy je stronami od (i), otrzymując

$$d(-1-q) + m(1) = m - d - qd = \mathrm{rem}(m-d, d).$$

Równanie to jest spełnione, ponieważ po prostu pomnożyliśmy jedno równanie przez stałą i odjedliśmy stronami od drugiego. Jeżeli $\mathrm{rem}(m - d, d) = 1$, kończymy obliczenia; wówczas powyższe równanie jest rozwiązaniem i $x = -1 - q$.

Powtarzamy opisane działania na dwóch ostatnich równaniach, otrzymując czwarte, i kontynuujemy je do momentu przyjęcia przez prawą stronę równania wartości 1. Mnożnik wartości d, zredukowany modulo m, jest wówczas szukaną odwrotnością d.

Przypadkowo okazuje się, że jeżeli $m - d < d$, czyli kiedy pierwszy iloraz wynosi 0, to trzeci wiersz stanowi kopię pierwszego, tak więc drugi iloraz jest niezerowy. Ponadto zwykle pierwszy wiersz zapisuje się jako

$$d(0) + m(1) = m,$$

jednak w naszym przypadku wartość $m = 2^W$ nie jest reprezentowana przez maszynę.

Cały proces najlepiej zilustruje konkretny przykład. Niech $m = 256$ oraz $d = 7$. Wówczas obliczenia przebiegają jak poniżej. W celu otrzymania trzeciego wiersza wystarczy zauważyć, że $q = \lfloor 249 / 7 \rfloor = 35$.

```
 7(-1) + 256( 1) = 249
 7( 1) + 256( 0) = 7
7(-36) + 256( 1) = 4
7( 37) + 256(-1) = 3
7(-73) + 256( 2) = 1
```

Zatem multiplikatywną odwrotnością 7, modulo 256, jest –73 lub, w ramach przedziału od 0 do 255, 183. Wystarczy sprawdzić, że $7 \cdot 183 = 1281 \equiv 1 \pmod{256}$.

Od trzeciego wiersza liczby całkowite w prawej kolumnie są resztami w odniesieniu do liczby znajdującej się powyżej jako dzielnika (d jest dzielną), tak więc tworzą one ciąg malejących nieujemnych liczb całkowitych. Zatem ciąg ten musi kończyć się 0 (tak, jak miałoby to miejsce powyżej, gdyby dodać jeden dodatkowy krok). Ponadto wartością tuż przed 0 musi być 1, co uzasadnia następujące rozumowanie. Załóżmy, że ciąg kończy się wartością b, po której występuje 0, gdzie $b \neq 1$. Liczba całkowita poprzedzająca b musi być wielokrotnością b, na przykład k_1b, aby kolejną resztą mogło być 0. Liczba całkowita poprzedzająca k_1b musi być postaci $k_1k_2b + b$, aby kolejną resztą mogło być b. Obliczając wartości ciągu okazuje się, że każda liczba musi być wielokrotnością b, wliczając w to dwie pierwsze (na powyższym schemacie są to liczby 249 i 7). Jednak to jest niemożliwe, ponieważ dwie pierwsze liczby całkowite to $m - d$ i d, które są względnie pierwsze.

To stanowi nieformalny dowód, że opisany powyżej proces kończy się z wartością 1 w prawej kolumnie, a stąd pozwala znaleźć multiplikatywną odwrotność d.

W celu jego wykonania na komputerze najpierw zauważmy, że jeżeli $d < 0$, należy do niego dodać 2^W. Jednak w arytmetyce uzupełnienia dwójkowego nie ma w rzeczywistości potrzeby wykonywania w tym momencie jakichkolwiek działań. Należy po prostu zinterpretować d jako liczbę bez znaku, bez względu na sposób jej interpretacji przez aplikację.

Obliczenie q wymaga zastosowania dzielenia bez znaku.

Należy zauważyć, że obliczenia można wykonać modulo m, ponieważ nie zmienia to wartości w prawej kolumnie (poza tym wartości te i tak należą do przedziału od 0 do $m - 1$). Jest to istotne, ponieważ pozwala, aby obliczenia były wykonywane „z pojedynczą precyzją" przy użyciu arytmetyki komputera modulo-2^W bez znaku.

Większość wartości w tabeli nie musi być reprezentowana. Kolumna zawierająca wielokrotności 256 nie musi być reprezentowana, gdyż rozwiązanie równania $dx + my = 1$ nie wymaga wartości y. Nie ma potrzeby reprezentowania d w pierwszej kolumnie. Zredukowane do najważniejszych elementów obliczenia z powyższego przykładu można zatem zapisać następująco:

```
255  249
  1    7
220    4
 37    3
183    1
```

Program w C przeprowadzający opisane obliczenia przedstawiono na listingu 10.4.

Listing 10.4. *Obliczanie multiplikatywnej odwrotności modulo 2^{32} za pomocą algorytmu Euklidesa*

```
unsigned mulinv(unsigned d) {        // d musi być nieparzyste.
   unsigned x1, v1, x2, v2, x3, v3, q;

   x1 = 0xFFFFFFFF;      v1 = -d;
   x2 = 1;               v2 = d;
   while (v2 > 1) {
      q = v1/v2;
      x3 = x1 - q*x2;    v3 = v1 - q*v2;
      x1 = x2;           v1 = v2;
      x2 = x3;           v2 = v3;
   }
   return x2;
}
```

Powodem, dla którego warunek zakończenia pętli ma postać `(v2 > 1)` zamiast bardziej naturalnej `(v2 != 1)` jest to, że gdyby użyto drugiego z tych warunków, pętla nigdy nie kończyłaby działania w przypadku wywołania programu z parzystą wartością argumentu. Najlepiej, jeśli zapewni się, że program nigdy nie będzie przechodził przez pętlę, której wykonywania nie można zakończyć, nawet w przypadku jego nieprawidłowego użycia (jeżeli argument `d` jest parzysty, `v2` nigdy nie przyjmuje wartości 1, ale przyjmuje wartość 0).

Co jednak oblicza program w przypadku przekazania jako argumentu liczby parzystej? Zgodnie z tym co powiedziano powyżej, oblicza on wartość liczby x, takiej że $dx \equiv 0 \pmod{2^{32}}$, która prawdopodobnie jest nieprzydatna. Jednakże po wprowadzeniu drobnej modyfikacji warunku wykonywania pętli do postaci `(v2 != 0)` oraz zwróceniu wartości `x1` zamiast `x2`, program oblicza liczbę x, taką że $dx \equiv g \pmod{2^{32}}$, gdzie g jest największym wspólnym dzielnikiem d i 2^{32} — czyli największą potęgą 2, która dzieli d. Zmodyfikowany program wciąż oblicza multiplikatywną odwrotność d dla nieparzystego d, ale wymaga wykonania o jedną więcej iteracji niż wersja pierwotna.

Jeśli chodzi o liczbę iteracji (dzieleń) wymaganych przez powyższy program, dla d nieparzystego i mniejszego od 20 wartością maksymalną jest 3, a średnią 1,7. Dla d bliskiego wartości 1000 wartość maksymalna wynosi 11 a średnia około 6.

Obliczanie multiplikatywnej odwrotności za pomocą metody Newtona

Ogólnie znanym faktem jest to, że w przypadku liczb rzeczywistych wartość $1/d$, dla $d \neq 0$, można obliczać z rosnącą dokładnością poprzez iteracyjne określanie wartości wyrażenia

$$x_{n+1} \equiv x_n\,(2 - dx_n), \tag{31}$$

zakładając, że początkowe oszacowanie x_0 jest wystarczająco bliskie wartości $1/d$. Liczba cyfr znaczących mniej więcej ulega podwojeniu w każdej iteracji.

O wiele mniej znanym faktem jest to, że ten sam wzór można wykorzystać do znalezienia multiplikatywnej odwrotności w zakresie arytmetyki modułowej na liczbach całkowitych. Przykładowo, w celu znalezienia multiplikatywnej odwrotności 3, modulo 256, rozpoczynamy od wartości $x_0 = 1$ (mogłaby to być dowolna liczba nieparzysta). Wówczas

$$\begin{aligned}
x_1 &\equiv 1\,(2 - 3\cdot 1) = -1,\\
x_2 &\equiv -1\,(2 - 3(-1)) = -5,\\
x_3 &\equiv -5\,(2 - 3(-5)) = -85,\\
x_4 &\equiv -85\,(2 - 3(-85)) = -21845 \equiv -85 \pmod{256}.
\end{aligned}$$

Iteracje osiągnęły stały punkt modulo 256, tak więc –85, czyli 171, jest multiplikatywną odwrotnością 3 (modulo 256). Wszystkie obliczenia można wykonać modulo 256.

Należy wyjaśnić, dlaczego opisany schemat działa poprawnie. Po pierwsze, jeżeli x_n spełnia zależność

$$dx_n \equiv 1 \pmod{m}$$

oraz jeżeli x_{n+1} definiuje wzór (31), to

$$dx_{n+1} \equiv 1 \pmod{m^2}.$$

Aby to sprawdzić, przyjmijmy, że $dx_n = 1 + km$. Wówczas

$$\begin{aligned}
dx_{n+1} &= dx_n\,(2 - dx_n)\\
&= (1 + km)(2 - (1 + km))\\
&= (1 + km)(1 - km)\\
&= 1 - k^2 m^2\\
&\equiv 1 \pmod{m^2}.
\end{aligned}$$

W naszym przypadku m jest potęgą 2, powiedzmy 2^N. Wówczas jeżeli

$$dx_n \equiv 1 \pmod{2^N}, \text{ to}$$
$$dx_{n+1} \equiv 1 \pmod{2^{2N}}.$$

W pewnym sensie, jeżeli x_n potraktujemy jako rodzaj przybliżenia $\overline{d}$, to każda iteracja zależności (31) podwaja liczbę bitów „dokładności" przybliżenia.

Okazuje się, że w przypadku operacji modulo 8 multiplikatywna odwrotność dowolnej (nieparzystej) liczby d jest tą samą liczbą. Zatem przyjęcie $x_0 = d$ jest rozsądnym i prostym określeniem początkowej wartości $\overline{d}$. Wówczas zależność (31) daje wartości x_1, x_2, ..., takie że

$$dx_1 \equiv 1 \pmod{2^6},$$
$$dx_2 \equiv 1 \pmod{2^{12}},$$
$$dx_3 \equiv 1 \pmod{2^{24}},$$
$$dx_4 \equiv 1 \pmod{2^{48}} \text{ i tak dalej.}$$

Zatem cztery iteracje wystarczą do znalezienia multiplikatywnej odwrotności modulo 2^{32} (jeżeli $x \equiv 1 \pmod{2^{48}}$, to $x \equiv 1 \pmod{2^n}$ dla $n \leq 48$). To prowadzi do napisania programu w C przedstawionego na listingu 10.5, w którym wszystkie obliczenia są wykonywane modulo 2^{32}.

Listing 10.5. *Obliczanie multiplikatywnej odwrotności modulo 2^{32} za pomocą metody Newtona*

```
unsigned mulinv(unsigned d) {        // d musi być nieparzyste.
   unsigned xn, t;

   xn = d;
loop: t = d*xn;
      if (t == 1) return xn;
      xn = xn*(2 - t);
      goto loop;
}
```

Dla około połowy wartości d program wykonuje 4 1/2 iteracji, czyli dziewięć mnożeń. Dla drugiej połowy (wartości, dla których początkowa wartość `xn` jest „poprawna co do 4 bitów", to znaczy $d^2 \equiv 1 \pmod{16}$), wykonuje siedem lub mniej (zwykle siedem) mnożeń. Zatem przeciętnie wykonuje on osiem mnożeń.

Pewna odmiana programu polega po prostu na wykonaniu pętli cztery razy, bez względu na wartość d, eliminując sterowanie pętli (osiem mnożeń). Inną odmianą jest zapewnienie w pewien sposób, aby początkowe oszacowanie x_0 było „poprawne co do 4 bitów" (to znaczy znalezienia x_0, które spełnia zależność $dx_0 \equiv 1 \pmod{16}$). Wówczas potrzebne są tylko trzy iteracje pętli. Dwa sposoby określenia początkowego oszacowania to

$$\boldsymbol{x_0 \leftarrow d + 2((d+1)\,\&\,4)} \text{ oraz}$$
$$\boldsymbol{x_0 \leftarrow d^2 + d - 1}.$$

W tym przypadku mnożenie przez 2 to przesunięcie w lewo, zaś obliczenia są wykonywane modulo 2^{32} (przepełnienia są ignorowane). Ponieważ druga zależność wykorzystuje mnożenie, pozwala zaoszczędzić tylko jedną instrukcję.

Owa troska o czas wykonania jest oczywiście całkowicie nieistotna w przypadku aplikacji kompilatora. W takiej aplikacji opisana procedura byłaby wykonywana na tyle rzadko, że jej kodowanie powinno uwzględniać raczej minimalną zajętość pamięci. Jednak w wielu innych aplikacjach istotne znaczenie może mieć konieczność szybkiego obliczania odwrotności multiplikatywnej.

Przykładowe odwrotności multiplikatywne

Podrozdział kończymy listą niektórych odwrotności multiplikatywnych, zawartą w tabeli 10.3.

Tabela 10.3. *Przykładowe odwrotności multiplikatywne*

d	$\overline{d}$		
(dzies.)	**mod 16 (dzies.)**	**mod 232 (heks.)**	**mod 264 (heks.)**
−7	−7	49249249	9249249249249249
−5	3	33333333	3333333333333333
−3	5	55555555	5555555555555555
−1	−1	FFFFFFFF	FFFFFFFFFFFFFFFF
1	1	1	1
3	11	AAAAAAAB	AAAAAAAAAAAAAAAB
5	13	CCCCCCCD	CCCCCCCCCCCCCCCD
7	7	B6DB6DB7	6DB6DB6DB6DB6DB7
9	9	38E38E39	8E38E38E38E38E39
11	3	BA2E8BA3	2E8BA2E8BA2E8BA3
13	5	C4EC4EC5	4EC4EC4EC4EC4EC5
15	15	EEEEEEEF	EEEEEEEEEEEEEEEF
25		C28F5C29	8F5C28F5C28F5C29
125		26E978D5	1CAC083126E978D5
625		3AFB7E91	D288CE703AFB7E91

Jak widać, w kilku przypadkach (d = 3, 5, 9, 11) multiplikatywna odwrotność d jest taka sama jak liczba magiczna dla dzielenia bez znaku przez d (patrz podrozdział 10.14 *Przykładowe liczby magiczne* na stronie 216). Jest to mniej lub bardziej przypadkowa zbieżność. Okazuje się, że dla tych liczb liczba magiczna M jest wielokrotnością m a wielokrotności takie mają postać $(2^p + 1) / d$, gdzie $p \geq 32$. W tym przypadku należy zauważyć, że

$$Md = \left(\frac{2^p + 1}{d}\right) d \equiv 1 \pmod{2^{32}},$$

tak więc $M \equiv \overline{d} \pmod{2^{32}}$.

10.16. Sprawdzanie zerowej reszty po wykonaniu dzielenia przez stałą

Multiplikatywna odwrotność dzielnika d może być wykorzystywana do sprawdzania zerowej reszty po wykonaniu dzielenia przez d [GM].

Dzielenie bez znaku

W pierwszej kolejności rozpatrzymy dzielenie bez znaku przez nieparzysty dzielnik d. Multiplikatywną odwrotność d będziemy oznaczać jako $\overline{d}$. Wówczas, ze względu na fakt, że $d\overline{d} \equiv 1 \pmod{2^W}$, gdzie W jest rozmiarem słowa maszynowego, określonym w bitach, $\overline{d}$ także jest nieparzyste. Stąd $\overline{d}$ jest względnie pierwsze do 2^W i, jak pokazano wcześniej w dowodzie twierdzenia MI, w miarę jak n przyjmuje kolejno wszystkie 2^W wartości modulo 2^W, $n\overline{d}$ przyjmuje wszystkie 2^W wartości modulo 2^W.

W poprzednim podrozdziale wykazano, że jeżeli n jest wielokrotnością d, to

$$\frac{n}{d} \equiv n\overline{d} \pmod{2^W}.$$

Oznacza to, że dla $n = 0, d, 2d, \ldots, \lfloor(2^W - 1)/d\rfloor d$, $n\overline{d} \equiv 0, 1, 2, \ldots, \lfloor(2^W - 1)/d\rfloor \pmod{2^W}$. Stąd dla n *nie* będących wielokrotnościami d, wartości $n\overline{d}$, zredukowane modulo 2^W do przedziału od 0 do $2^W - 1$, muszą być większe od $\lfloor(2^W - 1)/d\rfloor$.

Własność tę można wykorzystać w celu sprawdzenia zerowej reszty. Przykładowo, w celu sprawdzenia, czy liczba całkowita n jest wielokrotnością 25, mnożymy n przez $\overline{25}$ i porównujemy W skrajnie prawych bitów z wartością $\lfloor(2^W - 1)/25\rfloor$. W przypadku podstawowych instrukcji RISC można zapisać:

```
li      M,0xC28F5C29  Załadowanie multiplikatywnej odwrotności 25.
mul     q,M,n         q = prawa połowa M*n.
li      c,0x0A3D70A3  c = floor((2**32-1)/25).
cmpleu  t,q,c         Porównanie q z c oraz skok,
bt      t,is_mult     jeżeli n jest wielokrotnością 25.
```

W celu objęcia procedurą parzystych dzielników, niech $d = d_0 \cdot 2^k$, gdzie d_0 jest nieparzyste oraz $k \geq 1$. Wówczas, z uwagi na fakt, że liczba całkowita jest podzielna przez d wtedy i tylko wtedy, gdy jest podzielna przez d_0 oraz przez 2^k, i z uwagi na fakt, że n i $n\overline{d_0}$ posiadają tę samą liczbę końcowych zer ($\overline{d_0}$ jest nieparzyste), sprawdzenie czy n jest wielokrotnością d polega na:

nadaniu wartości $q = \mathrm{mod}(n\overline{d_0}, 2^W)$;

$q \leq \lfloor(2^W - 1)/d_0\rfloor$ i q kończy się k lub większą liczbą bitów 0,

gdzie funkcja „mod" jest rozumiana jako zredukowanie wartości $n\overline{d_0}$ do przedziału $[0, 2^W - 1]$.

Bezpośrednia implementacja tej metody wymaga dwóch sprawdzeń i warunkowych rozgałęzień, ale można je zredukować do jednej instrukcji *compare-branch* (*porównaj i skocz*) w dość efektywny sposób, jeżeli maszyna obsługuje instrukcję *rotate-shift* (*przesunięcie z obrotem*). Wynika to z następującego twierdzenia, w którym $a \overset{rot}{>>} k$ oznacza obrócenie słowa a w prawo o k pozycji ($0 \leq k \leq 32$).

TWIERDZENIE ZRU

twierdzenie

$x \overset{u}{\le} a$ oraz x kończy się k bitami zerowymi wtedy i tylko wtedy, gdy $x \overset{rot}{>>} k \overset{u}{\le} \lfloor a/2^k \rfloor$.

Dowód (przyjmujemy 32-bitową maszynę). Załóżmy, że $x \overset{u}{\le} a$ oraz x kończą się k bitami zerowymi. Wówczas, z uwagi na fakt, że $x \overset{u}{\le} a$, $\lfloor x/2^k \rfloor \overset{u}{\le} \lfloor a/2^k \rfloor$. Jednak $\lfloor x/2^k \rfloor = x \overset{rot}{>>} k$. Stąd $x \overset{rot}{>>} k \overset{u}{\le} \lfloor a/2^k \rfloor$. Jeżeli x nie kończy się k bitami zerowymi, to $x \overset{rot}{>>} k$ nie rozpoczyna się k bitami zerowymi, co ma miejsce w przypadku $\lfloor a/2^k \rfloor$, tak więc $x \overset{rot}{>>} k \overset{u}{>} \lfloor a/2^k \rfloor$. W końcu, jeżeli $x \overset{u}{>} a$ oraz x kończy się k bitami zerowymi, to liczba całkowita utworzona z pierwszych $32 - k$ bitów wartości x musi być większa od liczby utworzonej z pierwszych $32 - k$ bitów wartości a, czyli $\lfloor x/2^k \rfloor \overset{u}{>} \lfloor a/2^k \rfloor$.

Wykorzystanie tego twierdzenia pozwala określić, że sprawdzenie czy n jest wielokrotnością d, gdzie n i $d \ge 1$ są liczbami całkowitymi bez znaku oraz $d = d_0 \cdot 2^k$ dla d_0 nieparzystego, polega na wykonaniu następujących działań:

$$\boxed{\begin{array}{l} q \leftarrow \mathrm{mod}(n\overline{d_0}, 2^W); \\ q \overset{rot}{>>} k \overset{u}{\le} \lfloor (2^W - 1)/d \rfloor \end{array}}$$

Wykorzystano tu zależność $\lfloor \lfloor (2^W - 1)/d_0 \rfloor / 2^k \rfloor = \lfloor (2^W - 1)/(d_0 \cdot 2^k) \rfloor = \lfloor (2^W - 1)/d \rfloor$.

Jako przykład przedstawiamy kod, który sprawdza liczbę całkowitą bez znaku n, czy jest wielokrotnością 100:

```
li      M,0xC28F5C29  Zaladowanie multiplikatywnej odwrotności 25.
mul     q,M,n         q = prawa polowa M*n.
shrri   q,q,2         Obrócenie w prawo o dwie pozycje.
li      c,x'028F5C28' c = floor((2**32-1)/100).
cmpleu  t,q,c         Porównanie q z c oraz skok,
bt      t,is_mult     jeżeli n jest wielokrotnością 100.
```

Dzielenie ze znakiem, dzielnik ≥ 2

W przypadku dzielenia ze znakiem w poprzednim podrozdziale wykazano, że jeżeli n jest wielokrotnością d i d jest nieparzyste, to

$$\frac{n}{d} \equiv n\overline{d} \pmod{2^W}.$$

Zatem dla $n = \lceil -2^{W-1}/d \rceil \cdot d, \ldots, -d, 0, d, \ldots, \lfloor (2^{W-1} - 1)/d \rfloor \cdot d$ mamy $n\overline{d} \equiv \lceil -2^{W-1}/d \rceil, \ldots, -1, 0, 1, \ldots, \lfloor (2^{W-1} - 1)/d \rfloor \pmod{2^W}$. Ponadto, z uwagi na fakt, że $\overline{d}$ jest względnie pierwsze do 2^W, w miarę jak n przyjmuje kolejno wszystkie 2^W wartości modulo 2^W, $n\overline{d}$ przyjmuje wszystkie 2^W wartości modulo 2^W. Stąd n jest wielokrotnością d wtedy i tylko wtedy, gdy

$$\lceil -2^{W-1}/d \rceil \le \mathrm{mod}(n\overline{d}, 2^W) \le \lfloor (2^{W-1} - 1)/d \rfloor,$$

gdzie funkcja „mod" jest rozumiana jako zredukowanie wartości $n\overline{d}$ do przedziału $[-2^{W-1}, 2^{W-1} - 1]$.

Całość można nieco uprosić zauważając, że skoro d jest nieparzyste oraz, zgodnie z przyjętym założeniem, dodatnie i różne od 1, to nie dzieli 2^{W-1}. Stąd

$$\lceil -2^{W-1}/d \rceil = \lceil (-2^{W-1}+1)/d \rceil = -\lfloor (2^{W-1}-1)/d \rfloor$$

Zatem dla liczb ze znakiem sprawdzenie, czy n jest wielokrotnością d, gdzie $d = d_0 \cdot 2^k$ i d_0 jest nieparzyste, polega na:

nadaniu wartości $q = \mathrm{mod}(n\overline{d_0}, 2^W)$;

$-\lfloor (2^{W-1}-1/d_0 \rfloor \le q \le \lfloor (2^{W-1}-1)/d_0 \rfloor$ i q kończy się k lub większą liczbą bitów 0.

Z pozoru wydaje się, że wymaga to wykonania trzech sprawdzeń i skoków. Jednakże, podobnie jak w przypadku dzielenia bez znaku, można je zredukować do jednej instrukcji *compare-branch* dzięki wykorzystaniu poniższego twierdzenia.

twierdzenie

TWIERDZENIE ZRS

Jeżeli $\boldsymbol{a} \ge \mathbf{0}$, poniższe założenia są równoważne:

(1) $-\boldsymbol{a} \le \boldsymbol{x} \le \boldsymbol{a}$ oraz $\boldsymbol{x}$ kończy się k lub większą liczbą bitów zerowych;

(2) $\mathrm{abs}(\boldsymbol{x}) \overset{rot}{>>} k \overset{u}{\le} \lfloor \boldsymbol{a}/2^k \rfloor$;

(3) $\boldsymbol{x} + \boldsymbol{a}' \overset{rot}{>>} k \overset{u}{\le} \lfloor 2\boldsymbol{a}'/2^k \rfloor$;

gdzie $\boldsymbol{a}'$ jest równe wartości $\boldsymbol{a}$, w której k prawych bitów ustawiono na wartość 0 (to znaczy $\boldsymbol{a}' = \boldsymbol{a} \,\&\, -\mathbf{2}^k$).

Dowód (przyjmujemy 32-bitową maszynę). Aby sprawdzić, że zależność (1) jest równoważna (2), wystarczy zauważyć, że założenie $-\boldsymbol{a} \le \boldsymbol{x} \le \boldsymbol{a}$ jest równoważne $\mathrm{abs}(\boldsymbol{x}) \le \boldsymbol{a}$. Zatem ma zastosowanie twierdzenie ZRU, ponieważ obie strony tej nierówności są nieujemne.

Aby sprawdzić, że zależność (1) jest równoważna (3), wystarczy zauważyć, że założenie (1) jest równoważne samemu sobie, gdy $\boldsymbol{a}$ zastąpi się przez $\boldsymbol{a}'$. Zatem na mocy twierdzenia o sprawdzaniu ograniczeń ze strony 72, jest ono z kolei równoważne zależności

$$\boldsymbol{x} + \boldsymbol{a}' \overset{u}{\le} \mathbf{2}\boldsymbol{a}'.$$

Ponieważ wartość $\boldsymbol{x} + \boldsymbol{a}'$ kończy się k bitami zerowymi wtedy i tylko wtedy, gdy tak samo jest w przypadku wartości $\boldsymbol{x}$, ma tu zastosowanie twierdzenie ZRU, dając oczekiwany wynik.

Wykorzystanie części (3) twierdzenia pozwala określić, że sprawdzenie czy n jest wielokrotnością d, gdzie n i $d \ge 2$ są liczbami całkowitymi ze znakiem oraz $d = d_0 \cdot 2^k$ dla d_0 nieparzystego, polega na wykonaniu następujących działań:

$$\boxed{\begin{aligned} & q \leftarrow \mathrm{mod}(n\overline{d_0}, 2^W); \\ & a' \leftarrow \lfloor (2^{W-1}-1)/d_0 \rfloor \& -2^k. \\ & q + a' \overset{rot}{>>} k \overset{u}{\leq} \lfloor (2a')/2^k \rfloor \end{aligned}}$$

(a' można obliczać w czasie kompilacji, gdyż d jest stałą).

Jako przykład przedstawiamy kod, który sprawdza liczbę całkowitą ze znakiem n czy jest wielokrotnością 100. Należy zauważyć, że stałą $\lfloor 2a'/2^k \rfloor$ zawsze można obliczyć na podstawie stałej a' poprzez przesunięcie o $k-1$ bitów, co pozwala na zaoszczędzenie instrukcji lub operacji ładowania z pamięci w celu określenia wartości porównania.

```
li      M,0xC28F5C29  Załadowanie multiplikatywnej odwrotności 25.
mul     q,M,n         q = prawa połowa M*n.
li      c,0x51EB850   c = floor((2**31-1)/25) & -4.
add     q,q,c         Dodanie c.
shrri   q,q,2         Obrócenie w prawo o dwie pozycje.
shri    c,c,1         Obliczenie stałej do porównania.
cmpleu  t,q,c         Porównanie q z c oraz skok,
bt      t,is_mult     jeżeli n jest wielokrotnością 100.
```

Nie da się, myślę, przewidzieć wraz,
Kiedy nadejdzie dzielenia czas.

Tej operacji, którą się zgaduje,
A potem, mnożąc, weryfikuje.

Tej operacji kosztownej tak,
W cyklach trwonionej wciąż co dnia.

Kod dzielenia to dżungla często,
Długie dzielenie — katorga jest to.

Dowody rozum odebrać mogą,
W obłęd wpędzają sufit z podłogą.

Dobry kod dzielenia tutaj
To wynik jest przemyśleń Knutha.

Bo nawet Bóg się nie ośmieli
Nigdy niczego przez zero dzielić!

Rozdział 11.

Niektóre funkcje podstawowe

11.1. Całkowitoliczbowy pierwiastek kwadratowy

Przez *całkowitoliczbowy pierwiastek kwadratowy* (ang. *integer square root*) rozumiemy funkcję $\lfloor\sqrt{x}\rfloor$. Aby rozszerzyć jej zakres stosowalności i uniknąć konieczności określania, co należy zrobić w przypadku ujemnego argumentu, zakładamy, że x jest wartością bez znaku. Zatem $0 \le x \le 2^{32} - 1$.

Metoda Newtona

W przypadku liczb zmiennopozycyjnych pierwiastek kwadratowy niemal zawsze oblicza się za pomocą metody Newtona. Rozpoczyna się ona od określenia w pewien sposób początkowego oszacowania wartości g_0 dla $\sqrt{a}$. Następnie kolejne, coraz dokładniejsze oszacowania otrzymuje się na podstawie zależności:

$$g_{n+1} = \left(g_n + \frac{a}{g_n}\right)/2.$$

Iteracja podlega konwergencji kwadratowej, to znaczy, jeżeli w pewnym momencie g_n jest poprawne co do n bitów, to g_{n+1} jest poprawne co do $2n$ bitów. Program musi mieć możliwość określenia, kiedy wykonano wystarczającą liczbę iteracji, by mógł zakończyć działanie.

Miłą niespodzianką jest fakt, że metoda Newtona doskonale spełnia swoje zadanie w dziedzinie liczb całkowitych. Aby to sprawdzić, musimy określić następujące twierdzenie:

twierdzenie

Niech $g_{n+1} = \lfloor (g_n + \lfloor a / g_n \rfloor) / 2 \rfloor$, gdzie g_n oraz a są liczbami całkowitymi większymi od 0. Wówczas:

(a) jeżeli $g_n > \lfloor \sqrt{a} \rfloor$, to $\lfloor \sqrt{a} \rfloor \le g_{n+1} < g_n$;

(b) jeżeli $g_n = \lfloor \sqrt{a} \rfloor$, to $\lfloor \sqrt{a} \rfloor \le g_{n+1} \le \lfloor \sqrt{a} \rfloor + 1$.

Oznacza to, że jeżeli całkowitoliczbowa wartość g_n próby odgadnięcia wartości $\lfloor \sqrt{a} \rfloor$ jest za duża, to kolejna próba, g_{n+1}, będzie mniejsza od poprzedniej, ale nie mniejsza od $\lfloor \sqrt{a} \rfloor$. Stąd, jeżeli rozpoczniemy od zbyt dużej odgadywanej wartości, ciąg kolejnych odgadywanych wartości będzie zbieżny monotonicznie. Jeżeli wartość odgadywana $g_n = \lfloor \sqrt{a} \rfloor$, to kolejna odgadywana wartość jest albo równa, albo o jeden większa od g_n. Zapewnia to prosty sposób określenia, kiedy ciąg wartości zbiegnie się: jeżeli rozpoczynamy $g_0 \ge \lfloor \sqrt{a} \rfloor$, zbieżność następuje w momencie, gdy $g_n + 1 \ge g_n$ i wówczas wynik wynosi dokładnie g_n.

Jednak przypadek $a = 0$ należy potraktować szczególnie, ponieważ przedstawiona procedura doprowadziłaby do dzielenia 0 przez 0.

Dowód. (a) Ponieważ g_n jest liczbą całkowitą, to

$$g_{n+1} = \left\lfloor \left(g_n + \left\lfloor \frac{a}{g_n} \right\rfloor \right) / 2 \right\rfloor = \left\lfloor \left\lfloor g_n + \frac{a}{g_n} \right\rfloor / 2 \right\rfloor = \left\lfloor \left(g_n + \frac{a}{g_n} \right) / 2 \right\rfloor = \left\lfloor \frac{g_n^2 + a}{2g_n} \right\rfloor.$$

Ponieważ $g_n > \lfloor \sqrt{a} \rfloor$ oraz g_n jest liczbą całkowitą, $g_n > \sqrt{a}$. Definiujemy ε jako $g_n = (1+\varepsilon)\sqrt{a}$. Wówczas $\varepsilon > 0$ oraz

$$\left\lfloor \frac{g_n^2 + a}{2g_n} \right\rfloor = g_{n+1} \le \frac{g_n^2 + a}{2g_n},$$

$$\left\lfloor \frac{(1+\varepsilon)^2 a + a}{2(1+\varepsilon)\sqrt{a}} \right\rfloor = g_{n+1} < \frac{g_n^2 + g_n^2}{2g_n},$$

$$\left\lfloor \frac{2 + 2\varepsilon + \varepsilon^2}{2(1+\varepsilon)} \sqrt{a} \right\rfloor = g_{n+1} < g_n,$$

$$\left\lfloor \frac{2 + 2\varepsilon}{2(1+\varepsilon)} \sqrt{a} \right\rfloor \le g_{n+1} < g_n,$$

$$\lfloor \sqrt{a} \rfloor \le g_{n+1} < g_n.$$

(b) Ponieważ $g_n = \lfloor \sqrt{a} \rfloor$, $\sqrt{a} - 1 < g_n \le \sqrt{a}$, tak więc $g_n^2 \le a < (g_n + 1)^2$. Stąd otrzymujemy

$$\left\lfloor \frac{g_n^2 + g_n^2}{2g_n} \right\rfloor \le g_{n+1} \le \left\lfloor \frac{g_n^2 + (g_n+1)^2}{2g_n} \right\rfloor,$$

$$\lfloor g_n \rfloor \le g_{n+1} \le \left\lfloor g_n + 1 + \frac{1}{2g_n} \right\rfloor,$$

$$\lfloor \sqrt{a} \rfloor \le g_{n+1} \le \lfloor g_n + 1 \rfloor \text{ (bo } g_n \text{ to liczba całkowita oraz } \frac{1}{2g_n} < 1),$$

$$\lfloor \sqrt{a} \rfloor \le g_{n+1} \le \lfloor g_n \rfloor + 1 = \lfloor \sqrt{a} \rfloor + 1.$$

Najtrudniejszym etapem wykorzystania metody Newtona do obliczenia $\lfloor \sqrt{x} \rfloor$ jest określenie pierwszej wartości odgadywanej. Procedura z listingu 11.1 określa pierwszą odgadywaną wartość g_0 jako równą najmniejszej potędze liczby 2, która jest większa lub równa $\sqrt{x}$. Przykładowo dla $x = 4$, $g_0 = 2$, zaś dla $x = 5$, $g_0 = 4$.

Listing 11.1. *Obliczanie całkowitoliczbowego pierwiastka kwadratowego metodą Newtona*

```
int isqrt(unsigned x) {
   unsigned x1;
   int s, g0, g1;

   if (x <= 1) return x;
   s = 1;
   x1 = x - 1;
   if (x1 > 65535) {s = s + 8; x1 = x1 >> 16;}
   if (x1 > 255)   {s = s + 4; x1 = x1 >> 8;}
   if (x1 > 15)    {s = s + 2; x1 = x1 >> 4;}
   if (x1 > 3)     {s = s + 1;}

   g0 = 1 << s;                // g0 = 2**s.
   g1 = (g0 + (x >> s)) >> 1;  // g1 = (g0 + x/g0)/2.

   while (g1 < g0) {           // Wykonywanie przy zmniejszającym się
      g0 = g1;                 // poziomie przybliżenia.
      g1 = (g0 + (x/g0)) >> 1;
   }
   return g0;
}
```

Wartość g_0 pierwszej próby odgadnięcia jest potęgą liczby 2, zatem nie ma potrzeby wykonywania dzielenia rzeczywistego w celu otrzymania wartości g_1; zamiast tego w zupełności wystarczy instrukcja *przesunięcia w prawo*.

Wartość pierwszej próby odgadnięcia jest dokładna mniej więcej co do 1 bitu a metodę Newtona charakteryzuje konwergencja kwadratowa (liczba bitów dokładności podwaja się w każdej iteracji), zatem można by oczekiwać, że procedura spowoduje konwergencję po mniej więcej pięciu iteracjach (w przypadku maszyny 32-bitowej), co wymaga czterech operacji *dzielenia* (gdyż w pierwszej iteracji wykorzystuje się *przesunięcie w prawo*). Przeprowadzenie szczegółowych badań wykazuje, że maksymalna liczba dzieleń wynosi pięć lub cztery dla argumentów o wartościach do 16 785 407.

Jeżeli dostępna jest instrukcja *number of leading zeros* (nlz, *liczba zer wiodących*), wówczas określenie pierwszej odgadywanej wartości jest bardzo proste: należy zastąpić pierwsze siedem wierszy w powyższej procedurze następującymi:

```
if (x <= 1) return x;
s = 16 - nlz(x - 1)/2;
```

Innym rozwiązaniem, jeżeli instrukcja *number of leading zeros* nie jest dostępna, jest obliczenie s przy użyciu drzewa przeszukiwania binarnego. Metoda ta pozwala na otrzymywanie nieco lepszych wartości g_0: najmniejszej potęgi liczby 2, która jest większa lub równa wartości $\lfloor\sqrt{x}\rfloor$. Dla pewnych wartości x daje to mniejszą wartość g_0, jednak jest ona dostatecznie duża, aby kryterium zbieżności twierdzenia było zachowane. Różnice między tymi schematami działania przedstawiono poniżej.

Zakres wartości x dla listingu 11.1	**Zakres wartości x dla listingu 11.2**	**Pierwsza wartość odgadywana g_0**
0	0	0
1	1 do 3	1
od 2 do 4	4 do 8	2
od 5 do 16	9 do 24	4
od 17 do 64	25 do 80	8
od 65 do 256	81 do 288	16
...	...	...
od $2^{28}+1$ do 2^{30}	od $(2^{14}+1)^2$ do $(2^{15}+1)^2-1$	2^{15}
od $2^{30}+1$ do $2^{32}-1$	od $(2^{15}+1)^2$ do $2^{32}-1$	2^{16}

Omawianą procedurę przedstawiono na listingu 11.2. Wygodnie jest traktować małe wartości x ($0 \le x \le 24$) w sposób specjalny, tak aby nie zachodziła konieczność wykonywania dzielenia.

Listing 11.2. *Obliczanie całkowitoliczbowego pierwiastka kwadratowego, przeszukiwanie binarne dla pierwszej wartości odgadywanej*

```
int isqrt(unsigned x) {
   int s, g0, g1;

   if (x <= 4224)
      if (x <= 24)
         if (x <= 3) return (x + 3) >> 2;
         else if (x <= 8) return 2;
         else return (x >> 4) + 3;
      else if (x <= 288)
         if (x <= 80) s = 3; else s = 4;
      else if (x <= 1088) x = 5; else s = 6;
   else if (x <= 1025*1025 - 1)
      if (x <= 257*257 - 1)
         if (x <= 129*129 - 1) s = 7; else s = 8;
      else if (x <= 513*513 - 1) s = 9; else s = 10;
```

```
else if (x <= 4097*4097 - 1)
   if (x <= 2049*2049 - 1) s = 11; else s = 12;
else if (x <= 16385*16385 - 1)
   if (x <= 8193*8193 - 1) s = 13; else s = 14;
else if (x <= 32769*32769 - 1) s = 15; else s = 16;
g0 = 1 << s;                    // g0 = 2**s.

// Ciąg dalszy jak na listingu 11.1.
```

Czas wykonania w najmniej sprzyjającym przypadku algorytmu z listingu 11.1, dla podstawowego procesora RISC, wynosi około $26 + (D + 6)n$ cykli, gdzie D jest czasem wykonania instrukcji dzielenia określonym w cyklach, zaś n jest liczbą powtórzeń pętli `while`. Czas wykonania w najmniej sprzyjającym przypadku algorytmu z listingu 11.2 wynosi około $27 + (D + 6)n$ cykli, zakładając (w obu przypadkach), że wykonanie instrukcji *branch* (*rozgałęzienia*) zajmuje 1 cykl. Poniższa tabela zawiera średnią liczbę wykonań pętli w przypadku obu algorytmów dla x o rozkładzie jednostajnym w określonych zakresach wartości.

x	**Listing 11.1**	**Listing 11.2**
od 0 do 9	0,80	0
od 0 do 99	1,46	0,83
od 0 do 999	1,58	1,44
od 0 do 9999	2,13	2,06
od 0 do $2^{32} - 1$	2,97	2,97

Jeżeli przyjmiemy, że czas wykonania instrukcji *dzielenia* wynosi 20 cykli oraz x posiada rozkład jednostajny w zakresie od 0 do 9999, to oba algorytmy są wykonywane w około 81 cyklach.

Przeszukiwanie binarne

Ze względu na fakt, że algorytmy bazujące na metodzie Newtona rozpoczynają działanie od swego rodzaju przeszukiwania binarnego w celu otrzymania pierwszej odgadywanej wartości, pojawia się pytanie, czy nie można by całości obliczeń zaimplementować w postaci przeszukiwania binarnego. W przypadku takiej metody należałoby rozpocząć od określenia dwóch wartości granicznych, najprawdopodobniej 0 i 2^{16}. Pierwsza odgadywana wartość znajdowałaby się w połowie między wartościami granicznymi. Jeżeli kwadrat wartości punktu środkowego jest większy od argumentu x, górnym ograniczeniem staje się punkt środkowy. Jeżeli kwadrat wartości punktu środkowego jest mniejszy od argumentu x, dolnym ograniczeniem staje się punkt środkowy. Proces kończy się w momencie, gdy górne i dolne ograniczenie różnią się o 1 a dolne ograniczenie jest wówczas wynikiem.

Pozwala to na uniknięcie wykonywania dzielenia, ale wymaga kilku operacji mnożenia — 16, jeżeli jako początkowe wartości graniczne zostaną użyte 0 i 2^{16} (metoda w każdej iteracji pozwala uzyskać dodatkowy jeden bit precyzji). Na listingu 11.3 przedstawiono

odmianę tej procedury, gdzie początkowe wartości graniczne zostały dobrane nieco lepiej niż 0 i 2^{16}. Procedura ta pozwala również na zaoszczędzenie jednego cyklu na pętlę w przypadku większości procesorów RISC poprzez zamianę a i b w ten sposób, że dokonywane porównanie ma postać $b \geq a$ a nie $b - a \geq 1$.

Listing 11.3. *Obliczanie całkowitoliczbowego pierwiastka kwadratowego, proste przeszukiwanie binarne*

```
int isqrt(unsigned x) {
   unsigned a, b, m;                // Ograniczenia i punkt środkowy.

   a = 1;
   b = (x >> 5) + 8;                // Patrz opis.
   if (b > 65535) b = 65535;
   do {
      m = (a + b) >> 1;
      if (m*m > x) b = m - 1;
      else         a = m + 1;
   } while (b >= a);
   return a - 1;
}
```

Predykaty, które muszą zostać zachowane na początku każdej iteracji to $a \leq \lfloor\sqrt{x}\rfloor + 1$ oraz $b \geq \lfloor\sqrt{x}\rfloor$. Początkowa wartość b powinna być taka, aby można ją było łatwo obliczyć i powinna być bliska wartości $\lfloor\sqrt{x}\rfloor$. Rozsądnymi wartościami początkowymi są x, $x \div 4 + 1$, $x \div 8 + 2$, $x \div 16 + 4$, $x \div 32 + 8$, $x \div 64 + 16$ i tak dalej. Wyrażenia z początku listy stanowią lepszą wartość początkową dla małych wartości x a wyrażenia z końca listy są lepsze dla dużych wartości x (wartość $x \div 2 + 1$ także jest akceptowalna, ale prawdopodobnie mało przydatna, ponieważ $x \div 4 + 1$ jest zawsze lepszym lub równie dobrym ograniczeniem).

Za pomocą mniej lub bardziej zautomatyzowanych procedur można opracować kilka odmian programu przedstawionego na listingu 11.3, podstawiając $a + 1$ za a lub $b - 1$ za b albo też zmieniając $m = (a + b) \div 2$ na $m = (a + b + 1) \div 2$, czy też stosując pewną kombinację tych podstawień.

Czas wykonania procedury przedstawionej na listingu 11.3 wynosi około $6 + (M + 7{,}5)n$, gdzie M jest czasem wykonania instrukcji *mnożenia* określonym w cyklach, zaś n jest liczbą wykonań pętli. Poniższa tabela zawiera informacje o średniej liczbie wykonań pętli dla x o rozkładzie jednostajnym w określonym przedziale wartości.

x	Średnia liczba powtórzeń pętli
od 0 do 9	3,00
od 0 do 99	3,15
od 0 do 999	4,68
od 0 do 9999	7,04
od 0 do $2^{32} - 1$	16,00

Jeżeli przyjmiemy, że czas wykonania instrukcji *mnożenia* wynosi 5 cykli a x posiada rozkład jednostajny w przedziale od 0 do 9999, algorytm jest wykonywany średnio w ciągu 94 cykli. Maksymalny czas wykonania (dla $n = 16$) wynosi około 206 cykli.

Jeżeli dostępna jest instrukcja *number of leading zeros*, początkowe ograniczenia można określić na podstawie zależności

```
b = (1 << (33 - nlz(x))/2 - 1;
a = (b + 3)/2;
```

Oznacza to, że $b = 2^{(33-\mathrm{nlz}(x))\div 2} - 1$. Są to bardzo dobre wartości graniczne dla małych wartości x (jedna iteracja pętli dla $0 \le x \le 15$), ale w przypadku dużych wartości x stanowią tylko niewielką poprawę w stosunku do ograniczeń obliczonych na listingu 11.3. Dla x z przedziału od 0 do 9999 średnia liczba iteracji wynosi około 5,45, co daje czas wykonania wynoszący około 74 cykli, przy tych samych założeniach, co powyżej.

Algorytm sprzętowy

Istnieje algorytm oparty na operacjach *przesuwania* i *odejmowania*, służący do obliczania pierwiastka kwadratowego, który jest bardzo podobny do sprzętowego algorytmu dzielenia przedstawionego na listingu 9.2 na stronie 174. Wbudowany w osprzętowanie maszyny 32-bitowej algorytm ten wykorzystuje 64-bitowy rejestr, inicjalizowany wartością składającą się z 32 zer oraz argumentu x. W każdej iteracji 64-bitowy rejestr jest przesuwany w lewo o dwie pozycje a bieżący wynik y (początkowo równy 0) jest przesuwany w lewo o jedną pozycję. Następnie od lewej połowy rejestru 64-bitowego zostaje odjęta wartość $2y + 1$. Jeżeli wynik odejmowania jest nieujemny, zastępuje on lewą połowę rejestru 64-bitowego, zaś do y zostaje dodane 1 (operacja ta nie wymaga wykorzystania sumatora, ponieważ w tym momencie y kończy się bitem 0). Jeżeli wynik odejmowania jest ujemny, rejestr 64-bitowy i wartość y pozostają niezmienione. Iteracje są powtarzane 16 razy.

Algorytm ten został opisany w 1945 roku [JVN].

Co zaskakujące, opisane działania są wykonywane w czasie równym mniej więcej połowie czasu wykonania przez wspomniany sprzętowy algorytm dzielenia dla operacji $64 \div 32 \Rightarrow 32$, ponieważ wykonuje mniej więcej o połowę mniej iteracji a każda z nich jest tak samo złożona w przypadku obu algorytmów.

W celu zakodowania algorytmu programowo prawdopodobnie najlepszym rozwiązaniem jest uniknięcie użycia rejestru przesuwnego o rozmiarze podwójnego słowa, w którego przypadku przesunięcie wymaga około czterech instrukcji. Algorytm przedstawiony na listingu 11.4 [GLS1] osiąga to poprzez przesunięcie y i bitu maski m w prawo. Jest on wykonywany za pomocą około 149 podstawowych instrukcji RISC (przeciętnie). Dwa wyrażenia `y | m` można by również zapisać jako `y + m`.

Listing 11.4. *Obliczanie całkowitoliczbowego pierwiastka kwadratowego, algorytm sprzętowy*

```
int isqrt(unsigned x) {
   unsigned m, y, b;

   m = 0x40000000;
   y = 0;
   while (m != 0) {                    // Powtórzenie 16 razy
      b = y | m;
      y = y >> 1;
      if (x >= b) {
         x = x - b;
         y = y | m;
      }
      m = m >> 2;
   }
   return y;
}
```

Działanie tego algorytmu jest podobne do metody, której uczy się w szkole podstawowej. Zobrazowano je poniżej, gdzie na maszynie 8-bitowej szukamy wartości $\lfloor\sqrt{179}\rfloor$.

```
 1011 0011  x0   Początkowo x = 179 (0xB3).
- 1         b1
 ---------
 0111 0011  x1   0100 0000  y1
- 101       b2   0010 0000  y2
 ---------
 0010 0011  x2   0011 0000  y2
-  11 01    b3   0001 1000  y3
 ---------
 0010 0011  x3   0001 1000  y3   (Nie można odjąć).
-   1 1001  b4   0000 1100  y4
 ---------
 0000 1010  x4   0000 1101  y4
```

Wynik wynosi 13 z resztą 10, znajdującą się w rejestrze `x`.

Istnieje możliwość wyeliminowania instrukcji warunkowej `if x >= b` za pomocą często stosowanej sztuczki związanej z instrukcją *shift right signed 31* (*przesuń w prawo ze znakiem o 31 pozycji*). Można udowodnić, że najbardziej znaczący bit `b` ma zawsze wartość 0 (w rzeczywistości $b \leq 5 \cdot 2^{28}$), co pozwala uprościć predykat `x >= b` (patrz strona 35). Zatem blok instrukcji `if` można zastąpić następującym

```
t = (int)(x | -(x - b)) >> 31;   // -1, jeżeli x >= 0, w przeciwnym razie 0.
x = x - (b & t);
y = y | (m & t);
```

Powoduje to zastąpienie średnio trzech cykli siedmioma, zakładając że maszyna obsługuję operację `or not`, jednak może być wskazane, jeżeli skok warunkowy w tym kontekście zajmuje więcej niż pięć cykli.

Wydaje się, że metoda ta powinna być łatwiejsza od wykonania około stu cykli w celu obliczenia całkowitoliczbowego pierwiastka kwadratowego programowo. Na zakończenie przedstawiamy wyrażenia obliczane dla bardzo małych wartości argumentu. Może to być przydatne do przyspieszenia niektórych z przedstawionych algorytmów, jeżeli można oczekiwać, że argument będzie miał niedużą wartość.

Wyrażenie	jest poprawne w przedziale	i używa tylu instrukcji (pełny zestaw RISC)
x	od 0 do 1	0
$x > 0$	od 0 do 3	1
$(x+3) \overset{u}{\div} 4$	od 0 do 3	2
$x \overset{u}{>>} (x \overset{u}{\div} 2)$	od 0 do 3	2
$x \overset{u}{>>} (x > 1)$	od 0 do 5	2
$(x+12) \overset{u}{\div} 8$	od 1 do 8	2
$(x+15) \overset{u}{\div} 8$	od 4 do 15	2
$(x > 0) + (x > 3)$	od 0 do 8	3
$(x > 0) + (x > 3) + (x > 8)$	od 0 do 15	5

Nadzieja to doprawdy złudna,

Że obliczyć pierwiastek to rzecz nietrudna.

Bo nie wymyślę ja nic lepszego

Niż posłużenie się dwóch potęgą

Newtona sposób — nic innego.

11.2 Całkowitoliczbowy pierwiastek sześcienny

W przypadku *pierwiastków sześciennych* (ang. *cube root*) metoda Newtona nie sprawdza się zbyt dobrze. Wzór iteracyjny jest dość złożony:

$$x_{n+1} = \frac{1}{3}\left(2x_n + \frac{a}{x_n^2}\right),$$

a ponadto istnieje oczywiście problem określenia dobrej wartości początkowej x_0.

Jednakże istnieje algorytm sprzętowy, podobny do algorytmu sprzętowego obliczającego pierwiastek kwadratowy, który jest również całkiem dobrym rozwiązaniem programowym. Przedstawiono go na listingu 11.5.

Listing 11.5. *Obliczanie całkowitoliczbowego pierwiastka sześciennego, algorytm sprzętowy*

```
int icbrt(unsigned x) {
   int s;
   unsigned y, b;

   s = 30;
   y = 0;
   while (s >= 0) {                    // Powtórzenie 11 razy
      y = 2*y;
      b = (3*y*(y + 1) + 1) << s;
      s = s - 3;
      if (x >= b) {
         x = x - b;
         y = y + 1;
      }
   }
   return y;
}
```

Trzy instrukcje *dodawania* wartości 1 można zastąpić instrukcjami *alternatywy logicznej* z 1, ponieważ zwiększana wartość jest parzysta. Nawet po wprowadzeniu tej zmiany wartościowość algorytmu stoi pod znakiem zapytania w przypadku implementacji sprzętowej, głównie ze względu na występowanie operacji *mnożenia* $y * (y + 1)$.

Mnożenia tego można z łatwością uniknąć stosując optymalizację kompilatora redukcji złożoności do składnika y do kwadratu. Należy wprowadzić dodatkową zmienną bez znaku y2, która będzie przechowywała wartość y do kwadratu poprzez aktualizowanie jej wartości zawsze, kiedy y będzie przyjmować nową wartość. Tuż przed przypisaniem `y = 0` należy wykonać `y2 = 0`, a przed `y = 2*y` należy wstawić `y2 = 4*y2`. Należy zmienić przypisanie do zmiennej `b` na `b = (3*y2 + 3*y + 1) << s` (i wyciągnąć przed nawias 3). Tuż przed instrukcją `y = y + 1` należy wstawić `y2 = y2 + 2*y + 1`. Wynikowy program nie będzie zawierał żadnych operacji *mnożenia* innych niż przez małe wartości stałe, które można zamienić na instrukcje *przesunięć* i *dodawań*. Program zawiera trzy instrukcje *dodawania* wartości 1, które można zamienić na instrukcje *alternatywy* logicznej. Będzie on działał szybciej, chyba że dana maszyna wykonuje instrukcję *mnożenia* w czasie dwóch lub mniej cykli.

Uwaga: w pracy [GLS1] podkreślono fakt, że kod z listingu 11.5 oraz jego wersja pochodna o zredukowanej złożoności nie działają w przypadku, gdy w bezpośredni sposób przeniesie się je na maszynę 64-bitową. Przypisanie do zmiennej `b` może wówczas spowodować przepełnienie. Problemu tego można uniknąć pomijając operację *przesunięcia w lewo* o `s` pozycji z instrukcji przypisania do `b`, a wstawiając po tym przypisaniu do `b` instrukcję `bs = b << s` i zmieniając dwa dodatkowe wiersze `if (x >= b) {x = x - b...` na `if (x >= bs&&b == (bs >> s)) {x = x - bs....`

11.3. Całkowitoliczbowe podnoszenie do potęgi

Obliczanie wartości x^n poprzez rozkład binarny n

Dobrze znaną techniką obliczania wartości x^n, kiedy n jest nieujemną liczbą całkowitą, jest zastosowanie reprezentacji binarnej wartości n. Technika ta ma zastosowanie w przypadku obliczania wyrażeń postaci $x \bullet x \bullet x \bullet \ldots \bullet x$, gdzie symbol • oznacza dowolną operację posiadającą własność łączności, taką jak *dodawanie*, *mnożenie* (w tym mnożenie macierzy) czy *złączanie* ciągów znaków (zgodnie z notacją $(\text{'ab'})^3 = \text{'ababab'}$). Jako przykład przyjmijmy, że chcemy obliczyć wartość $y = x^{13}$. 13 w zapisie binarnym wyraża się jako 1101 (to znaczy 13 = 8 + 4 + 1),

$$x^{13} = x^{8+4+1} = x^8 \cdot x^4 \cdot x^1.$$

Zatem wartość x^{13} można obliczyć w sposób następujący:

$$\begin{aligned} t_1 &\leftarrow x^2 \\ t_2 &\leftarrow t_1^2 \\ t_3 &\leftarrow t_2^2 \\ y &\leftarrow t_3 \cdot t_2 \cdot x. \end{aligned}$$

Wymaga to pięciu operacji *mnożenia* a zatem znacząco mniej od dwunastu, wymaganych w oryginalnym przypadku 12 mnożeń x.

Jeżeli wykładnik jest zmienną, o której wiadomo, że jest nieujemną liczbą całkowitą, technikę tę można zastosować w formie podprogramu, który przedstawiono na listingu 11.6.

Listing 11.6. *Obliczanie wartości x^n poprzez rozkład binarny wartości n*

```
int iexp(int x, unsigned n) {
   int p, y;

   y = 1;                    // Inicjalizacja wyniku
   p = x;                    // oraz p.
   while (1) {
      if (n & 1) y = p*y;    // Jeżeli n jest nieparzyste, mnożenie przez p.
      n = n >> 1;            // Pozycja następnego bitu wartości n.
      if (n == 0) return y;  // Jeżeli brak dalszych bitów w n.
      p = p*p;               // Potęga dla kolejnego bitu n.
   }
}
```

Liczba wykonywanych operacji *mnożenia* w przypadku tej metody dla wykładnika $n \geq 1$ wynosi

$$\lfloor \log_2 n \rfloor + \text{nbits}(n) - 1.$$

Nie zawsze jest to minimalna liczba wykonywanych operacji *mnożenia*. Przykładowo, dla $n = 27$ metoda rozkładu binarnego oblicza

$$x^{16} \cdot x^{8} \cdot x^{2} \cdot x^{1},$$

co wymaga siedmiu operacji *mnożenia*. Jednakże schemat działania określony zapisem

$$((x^3)^3)^3$$

wymaga tylko sześciu. Najmniejsza liczba, dla której metoda rozkładu binarnego nie jest optymalna to $n = 15$ (wskazówka: $x^{15} = (x^3)^5$).

Co być może zaskakujące, nie jest znana żadna prosta metoda, która dla wszystkich n znajdowałaby optymalną sekwencję operacji *mnożenia* w celu obliczenia wartości x^n. Jedyne znane metody wykorzystują wyszukiwanie obszerne. Problem ten omówiono w pozycji [Knu2, podrozdz. 4.6.3].

Istnieje pewna odmiana metody rozkładu binarnego, w której binarną reprezentację wykładnika przegląda się od strony lewej do prawej [Rib, 32], co stanowi analogię do metody konwersji wartości binarnej na dziesiętną od strony lewej do prawej. Inicjalizujemy wynik y wartością 1 i przeglądamy wykładnik od strony lewej do prawej. Kiedy napotykamy 0, podnosimy y do kwadratu. Kiedy napotykamy 1, podnosimy y do kwadratu i wynik mnożymy przez x. Pozwala to obliczyć $x^{13} = x^{1101_2}$ jako

$$(((1^2 \cdot x)^2 \cdot x)^2)^2 \cdot x.$$

Metoda ta zawsze wymaga wykonania tej samej liczby (nietrywialnych) operacji *mnożenia*, co metoda przeglądania od strony prawej do lewej z listingu 11.6.

Wartość 2^n w języku Fortran

Kompilator IBM XL Fortran przyjmuje następującą definicję funkcji podnoszenia liczby 2 do potęgi

$$\mathrm{pow2}(n) = \begin{cases} 2^n, & 0 \le n \le 30, \\ -2^{31}, & n = 31, \\ 0, & n < 0 \text{ lub } n \ge 32. \end{cases}$$

Zakładamy, że n i wynik są interpretowane jako liczby całkowite ze znakiem. Standard ANSI/ISO języka Fortran wymaga, aby wynik był równy 0, jeżeli $n < 0$. Powyższa definicja dla $n \ge 31$ wydaje się rozsądnym rozwiązaniem o tyle, że zwraca poprawny wynik modulo 2^{32} i zgadza się z wynikiem, który zwróciłoby wielokrotne mnożenie.

Standardowym sposobem obliczania wartości 2^n jest załadowanie do rejestru wartości 1 i przesunięcie go w lewo o n pozycji. Nie spełnia to jednak definicji języka Fortran, ponieważ wartości przesunięć są zwykle traktowane jako modulo 64 lub modulo 32 (w przypadku maszyny 32-bitowej), co powoduje niepoprawne wyniki dla dużych lub ujemnych wartości przesunięć.

Jeżeli maszyna obsługuje instrukcję *number of leading zeros*, wartość pow2(n) można obliczyć za pomocą czterech instrukcji w sposób następujący [Shep]:

$x \leftarrow \text{nlz}(n \overset{u}{>>} 5)$; // $x \leftarrow 32$, jeżeli $0 \le n \le 31$, w przeciwnym wypadku $x < 32$.

$x \leftarrow x \overset{u}{>>} 5$; // $x \leftarrow 1$, jeżeli $0 \le n \le 31$, w przeciwnym wypadku 0.

$\text{pow2} \leftarrow x << n$;

Operacje *przesunięcia w prawo* są „logiczne" (bez propagacji znaku), choć n stanowi wielkość ze znakiem.

Jeżeli maszyna nie obsługuje instrukcji „nlz", można ją zastąpić jednym ze sprawdzeń $x = 0$ przedstawionych w podrozdziale *Predykaty porównań* na stronie 35, zmieniając wyrażenie $x \overset{u}{>>} 5$ na $x \overset{u}{>>} 31$. Być może lepszym rozwiązaniem jest wykorzystanie spostrzeżenia, że predykat $0 \le x \le 31$ jest równoważny $x \overset{u}{<} 32$, a następnie uproszczenie wyrażenia dla $x \overset{u}{<} y$, podanego we wspomnianym podrozdziale. Przyjmie ono wówczas postać $\neg x \,\&\, (x - 32)$. Pozwala to na otrzymanie rozwiązania w pięciu instrukcjach (czterech, jeżeli obsługiwana jest instrukcja `and not`):

$x \leftarrow \neg n \,\&\, (n - 32)$; // $x < 0$, jeżeli $0 \le n \le 31$.

$x \leftarrow x \overset{u}{>>} 31$; // $x = 1$, jeżeli $0 \le n \le 31$, w przeciwnym wypadku 0.

$\text{pow2} \leftarrow x << n$;

11.4. Logarytm całkowitoliczbowy

Przez funkcję *logarytmu całkowitoliczbowego* (ang. *integer logarithm*) rozumiemy funkcję $\lfloor \log_b x \rfloor$, gdzie x jest dodatnią liczbą całkowitą, zaś b jest liczbą całkowitą większą lub równą 2. Zwykle $b = 2$ lub $b = 10$ i funkcje te oznaczamy, odpowiednio, jako „ilog2" oraz „ilog10". Nazwy „ilog" używamy w sytuacji, gdy podstawa logarytmu nie jest określona.

Wygodnym rozwiązaniem jest rozszerzenie definicji na wartość $x = 0$ przez zdefiniowanie $\text{ilog}(0) = -1$ [CJS]. Za taką definicją przemawia kilka względów:

- funkcja ilog2(x) jest wówczas w prosty sposób powiązana z funkcją *number of leading zeros*, nlz(x), poprzez wzór przedstawiony poniżej, wliczając w to przypadek $x = 0$. Zatem jeżeli jedna z tych funkcji jest zaimplementowana sprzętowo lub programowo, z łatwością można określać wartości drugiej:

$$\text{ilog2}(x) = 31 - \text{nlz}(x);$$

- nie ma problemu z obliczeniem wartości $\lceil \log(x) \rceil$, wykorzystując wzór przedstawiony poniżej. Dla $x = 1$ wzór ten określa, że $\text{ilog}(0) = -1$:

$$\lceil \log(x) \rceil = \text{ilog}(x - 1) + 1;$$

- powoduje, że poniższa tożsamość jest spełniona dla $x = 1$ (ale nie dla $x = 0$):

$$\text{ilog2}(x \div 2) = \text{ilog2}(x) - 1;$$

- zachowuje prawdziwość tożsamości matematycznej:

$$\lfloor \log_{10} x \rfloor = \lfloor (\log_{10} 2) \log_2 x \rfloor;$$

- powoduje, że wynikiem ilog(x) jest mały, zwarty zbór liczb całkowitych (od –1 do 31 dla ilog2(x) na maszynie 32-bitowej, gdzie x jest wartością bez znaku), co bezpośrednio predysponuje ją do indeksowania tablic;
- wynika w naturalny sposób z kilku algorytmów obliczających ilog2(x) oraz ilog10(x).

Niestety nie jest to poprawna definicja „liczby cyfr w liczbie x", którą określa zależność ilog(x) + 1 dla wszystkich x oprócz $x = 0$. Jednak najlepszym rozwiązaniem wydaje się traktowanie tego przypadku jako anomalii.

Dla $x < 0$ wartość funkcji ilog(x) jest niezdefiniowana. W celu rozszerzenia zakresu stosowalności funkcji definiujemy ją jako odwzorowanie liczb całkowitych bez znaku na liczby całkowite ze znakiem. Zatem ujemny argument nie może się pojawić.

Logarytm całkowitoliczbowy o podstawie 2

Obliczanie wartości funkcji ilog2(x) w zasadzie jest identyczne z obliczaniem liczby zer wiodących, co omówiono w podrozdziale *Zliczanie zer wiodących* na stronie 97. Wszystkie algorytmy z tego podrozdziału można z łatwością zmodyfikować, tak aby obliczały wartość ilog2(x) bezpośrednio, a nie poprzez obliczenie wartości nlz(x) i odjęcie wyniku od 31 (w przypadku algorytmu z listingu 5.11 na stronie 101 należy zmienić wiersz `return pop(-x)` na `return pop(x) - 1`).

Logarytm całkowitoliczbowy o podstawie 10

Funkcja ta ma zastosowanie w przypadku konwersji liczby do postaci dziesiętnej w celu umieszczenia jej w wierszu z pominięciem zer wiodących. W procesie konwersji sukcesywnie dzielimy przez 10, zwracając najmniej znaczącą cyfrę jako pierwszą. Przydatna byłaby wcześniejsza znajomość miejsca, w którym należy wstawić najmniej znaczącą cyfrę w celu uniknięcia konieczności umieszczenia konwertowanej liczby w tymczasowej lokacji a następnie przenoszenia jej.

Rozsądnym rozwiązaniem problemu obliczenia wartości ilog10(x) jest metoda przeszukiwania tablicy. Może to być przeszukiwanie binarne, ale z uwagi na fakt, że tablica ma niewielki rozmiar i w wielu zastosowaniach x jest zwykle małe, najlepszym rozwiązaniem jest prawdopodobnie proste przeszukiwanie liniowe. Taki dość oczywisty program przedstawiono na listingu 11.7.

Listing 11.7. *Całkowitoliczbowy logarytm o podstawie 10, proste przeszukiwanie tablicy*

```
int ilog10(unsigned x) {
   int i;
   static unsigned table[11] = {0, 9, 99, 999, 9999, 99999,
      999999, 9999999, 99999999, 999999999, 0xFFFFFFFF};

   for (i = -1; ; i++) {
      if (x <= table[i+1]) return i;
   }
}
```

W przypadku prostego procesora RISC program ten można zaimplementować w sposób umożliwiający jego wykonanie za pomocą $9 + 4\lfloor \log_{10} x \rfloor$ instrukcji. Zatem jest on wykonywane za pomocą od 5 do 45 instrukcji, przy wartości 13 (dla $10 \le x \le 99$) jako typowej.

Program z listingu 11.7 można z łatwością przerobić na wersję „rejestrową" (bez wykorzystania tablicy). Część wykonawczą takiego programu przedstawiono na listingu 11.8. Może okazać się on przydatny, jeżeli maszyna oferuje szybki sposób mnożenia przez 10.

Listing 11.8. *Całkowitoliczbowy logarytm o podstawie 10, wielokrotne mnożenie przez 10*

```
p = 1;
for (i = -1; i <= 8; i++) {
   if (x < p) return i;
   p = 10*p;
}
return i;
```

Program ten można zaimplementować w sposób umożliwiający jego wykonanie za pomocą $10 + 6\lfloor \log_{10} x \rfloor$ instrukcji prostego procesora RISC (licząc mnożenie jako pojedynczą instrukcję). Oznacza to 16 instrukcji dla $10 \le x \le 99$.

Można również skorzystać z przeszukiwania binarnego, tworząc algorytm niezawierający pętli i niekorzystający z tablicy. Taki algorytm może porównywać x z 10^4, następnie albo z 10^2, albo 10^6 i tak dalej, do momentu aż zostanie znaleziony wykładnik n, taki że $10^n \le x \le 10^{n+1}$. Ścieżki działań są wykonywane za pomocą od 10 do 18 instrukcji, z których cztery lub pięć to instrukcje *skoku* (wliczając w to końcowy skok bezwarunkowy).

Program przedstawiony na listingu 11.9 stanowi modyfikację przeszukiwania binarnego — w dowolnej ścieżce zawiera maksymalnie cztery instrukcje *skoku* i został zapisany w sposób faworyzujący małe wartości x. Program jest wykonywany za pomocą sześciu podstawowych instrukcji procesora RISC dla $10 \le x \le 99$ oraz za pomocą od 11 do 16 instrukcji dla $x \ge 100$.

Listing 11.9. *Całkowitoliczbowy logarytm o podstawie 10, zmodyfikowane przeszukiwanie binarne*

```
int ilog10(unsigned x) {
   if (x > 99)
      if (x < 1000000)
         if (x < 10000)
            return 3 + ((int)(x - 1000) >> 31);
         else
            return 5 + ((int)(x - 100000) >> 31);
      else
         if (x < 100000000)
            return 7 + ((int)(x - 10000000) >> 31);
         else
            return 9 + ((int)((x-1000000000)&~x) >> 31);
   else
      if (x > 9) return 1;
      else       return ((int)(x - 1) >> 31);
}
```

Instrukcje *przesunięcia* w tym programie oznaczają *przesunięcia ze znakiem*, co wyjaśnia użycie konstrukcji rzutowania typów `(int)`. Jeżeli maszyna nie obsługuje takich instrukcji, można użyć jednego z przedstawionych poniżej alternatywnych rozwiązań, które wykorzystują przesunięcia bez znaku. Zilustrowano je dla przypadku pierwszej instrukcji `return`. Niestety pierwsze dwa rozwiązania, w celu zapewnienia wydajnej implementacji, wymagają użycia instrukcji *subtract from immediate* (*odejmij od wartości natychmiastowej*), której większość maszyn nie obsługuje. Ostatnie rozwiązanie jest związane z dodaniem stałej o dużej wartości (dwie instrukcje), ale nie ma to znaczenia w przypadku drugiej i trzeciej instrukcji `return`, które i tak wymagają stałej o dużej wartości. Stała ma wartość $2^{31} - 1000$.

```
return 3 - ((x - 1000) >> 31);
return 2 + ((999 - x) >> 31);
return 2 + ((x + 2147482648) >> 31);
```

Alternatywą dla czwartej instrukcji `return` jest

```
return 8 + ((x + 1147483648) | x) >> 31;
```

gdzie stała ma wartość $2^{31} - 10^9$. Pozwala ona uniknąć konieczności używania zarówno instrukcji `and` `not`, jak i przesunięcia ze znakiem.

Alternatywnymi rozwiązaniami w przypadku ostatniej konstrukcji `if-else` są

```
return ((int)(x - 1) >> 31) | ((unsigned)(9 - x) >> 31);
return (x > 9) + (x > 0) -1;
```

W obydwóch przypadkach unika się instrukcji *skoku*.

Jeżeli dostępna jest instrukcja nlz(x) lub ilog2(x), istnieją lepsze i bardziej interesujące sposoby obliczania ilog10(x). Przykładowo, program z listingu 11.10 wykonuje to za pomocą przeglądu dwóch tablic [CJS].

Listing 11.10. *Całkowitoliczbowy logarytm o podstawie 10 z logarytmu o podstawie 2, przegląd dwóch tablic*

```
int ilog10(unsigned x) {
   int y;
   static unsigned char table1[33] = {9, 9, 9, 8, 8, 8,
      7, 7, 7, 6, 6, 6, 6, 5, 5, 5, 4, 4, 4, 3, 3, 3, 3,
      2, 2, 2, 1, 1, 1, 0, 0, 0, 0};
   static unsigned table2[10] = {1, 10, 100, 1000, 10000,
      100000, 1000000, 10000000, 100000000, 1000000000};

   y = table1[nlz(x)];
   if (x < table2[y]) y = y - 1;
   return y;
}
```

Na podstawie tablicy `table1` jest otrzymywane przybliżenie wartości ilog10(x). Przybliżenie to jest zwykle poprawną wartością, ale jest o jeden za duże dla $x = 0$ oraz dla x z przedziałów od 8 do 9, od 64 do 99, od 512 do 999, od 8192 do 9999 i tak dalej. Druga tablica podaje wartość, poniżej której oszacowanie musi zostać skorygowane przez odjęcie od niego 1.

Taki schemat działania wykorzystuje w sumie 73 bajty dla tablic i może zostać zakodowany za pomocą tylko sześciu instrukcji na komputerze IBM System/370 [CJS] (w tym celu wartości w tablicy `table1` muszą być czterokrotnie większe od przedstawionych powyżej). Wykonanie zajmuje około dziesięciu instrukcji w przypadku procesora RISC, obsługującego instrukcję *number of leading zeros*, ale nieobsługującego żadnej innej złożonej instrukcji. Inne omawiane metody stanowią odmiany tej opisanej.

Pierwsza odmiana polega na wyeliminowaniu instrukcji *skoków warunkowych*, które wynikają z użycia instrukcji *if*. W rzeczywistości program z listingu 11.10 można zakodować w sposób zapewniający uniknięcie instrukcji *skoków*, jeżeli maszyna obsługuje instrukcję *set less than unsigned* (*ustaw mniej niż wartość bez znaku*), ale opisana poniżej metoda może być wykorzystana w przypadku maszyn, które nie obsługują żadnych niestandardowych instrukcji (innych niż *number of leading zeros*).

Metoda polega na zastąpieniu instrukcji *if* odejmowaniem oraz przesunięciem w prawo o 31 pozycji, tak aby można było odjąć bit znaku od y. Pewna trudność pojawia się w przypadku dużych wartości x ($x \geq 2^{31} + 10^9$), ale można ją obejść, dodając wpis do tablicy `table2`, zgodnie z listingiem 11.11.

Listing 11.11. *Całkowitoliczbowy logarytm o podstawie 10 z logarytmu o podstawie 2, przegląd dwóch tablic, brak rozgałęzień*

```
int ilog10(unsigned x) {
   int y;
   static unsigned char table1[33] = {10, 9, 9, 8, 8, 8,
      7, 7, 7, 6, 6, 6, 6, 5, 5, 5, 4, 4, 4, 3, 3, 3, 3,
      2, 2, 2, 1, 1, 1, 0, 0, 0, 0};
   static unsigned table2[10] = {1, 10, 100, 1000, 10000,
      100000, 1000000, 10000000, 100000000, 1000000000,
      0};
```

```
    y = table1[nlz(x)];
      y = y - ((x - table2[y]) >> 31);
      return y;
}
```

Program ten wymaga wykonania około 11 instrukcji w przypadku procesora RISC, który obsługuje instrukcję *number of leading zeros*, ale poza tym jest „prosty". Można go zmodyfikować, tak aby zwracał wartość 0 zamiast –1 dla $x = 0$ (co jest rozwiązaniem preferowanym w przypadku problemu konwersji do postaci dziesiętnej) poprzez zamianę ostatniej wartości w tablicy `table1` na 1 (to znaczy, zmieniając „0, 0, 0, 0" na „0, 0, 0, 1").

Kolejna odmiana polega na zastąpieniu przeglądania pierwszej tabeli operacjami odejmowania, mnożenia i przesunięcia. Wydaje się to możliwe, gdyż wartości $\log_{10}x$ oraz $\log_2 x$ są ze sobą powiązane przez stałą multiplikatywną a dokładnie $\log_{10}2 = 0{,}30103\ldots$. Zatem istnieje możliwość obliczenia $\mathrm{ilog10}(x)$ poprzez obliczenie $\lfloor c\,\mathrm{ilog2}(x)\rfloor$ dla odpowiedniej wartości $c \approx 0{,}30103$ oraz skorygowanie wyniku przy użyciu tablicy, takiej jak `table2` z listingu 11.11.

Aby prześledzić cały proces, przyjmijmy, że $\log_{10}2 = c + \varepsilon$, gdzie $c > 0$ jest wymiernym przybliżeniem wartości $\log_{10}2$, będącej wykorzystywanym mnożnikiem, zaś $\varepsilon > 0$. Wówczas dla $x \geq 1$:

$$\begin{aligned}
\mathrm{ilog10}(x) &= \lfloor \log_{10} x \rfloor = \lfloor (c+\varepsilon)\log_2 x \rfloor \\
\lfloor c \log_2 x \rfloor \leq \mathrm{ilog10}(x) &= \lfloor c \log_2 x + \varepsilon \log_2 x \rfloor \\
\lfloor c\,\mathrm{ilog2}(x) \rfloor \leq \mathrm{ilog10}(x) &\leq \lfloor c\,(\mathrm{ilog2}(x)+1) + \varepsilon \log_2 x \rfloor \\
&\leq \lfloor c\,\mathrm{ilog2}(x) + c + \varepsilon \log_2 x \rfloor \\
&\leq \lfloor c\,\mathrm{ilog2}(x) \rfloor + \lfloor c + \varepsilon \log_2 x \rfloor + 1.
\end{aligned}$$

Zatem, jeżeli wybierzemy c tak, aby $c + \varepsilon \log_2 x < 1$, to $\lfloor c\,\mathrm{ilog2}(x) \rfloor$ aproksymuje $\mathrm{ilog10}(x)$ z błędem 0 lub +1. Ponadto, jeżeli przyjmiemy $\mathrm{ilog2}(0) = \mathrm{ilog10}(0) = -1$, to $\lfloor c\,\mathrm{ilog2}(0) \rfloor = \mathrm{ilog10}(0)$ (ponieważ $0 < c \leq 1$), tak więc nie musimy rozpatrywać tego przypadku oddzielnie (istnieją inne definicje, które można by tu zastosować, na przykład $\mathrm{ilog2}(0) = \mathrm{ilog10}(0) = 0$).

Ponieważ $\varepsilon = \log_{10} 2 - c$, musimy c wybrać tak, aby

$$\begin{aligned}
&c + (\log_{10} 2 - c)\log_2 x < 1, \text{ czyli} \\
&c(\log_2 x - 1) > (\log_{10} 2)\log_2 x - 1.
\end{aligned}$$

Warunek ten jest spełniony dla $x = 1$ (ponieważ $c < 1$) oraz dla $x = 2$. Dla większych x musi zachodzić

$$c > \frac{(\log_{10} 2)\log_2 x - 1}{\log_2 x - 1}.$$

Najostrzejsze wymagania odnośnie c występują dla dużych wartości x. W przypadku maszyny 32-bitowej $x < 2^{32}$, tak więc wybranie wartości

$$c > \frac{0{,}30103 \cdot 32 - 1}{32 - 1} \approx 0{,}27848.$$

jest wystarczająco dobre. Ponieważ $c < 0{,}30103$ (gdyż $\varepsilon > 0$), $c = 9/32 = 0{,}28125$ jest odpowiednią wartością. Doświadczenia dowodzą, że mniej dokładne wartości, takie jak 5/16 lub 1/4, są nieodpowiednie.

Prowadzi to do opracowania schematu działania przedstawionego na listingu 11.12, który pozwala na oszacowanie niższej wartości, a następnie jej skorygowanie przez dodanie 1. Program jest wykonywany w około 11 instrukcjach w przypadku procesora RISC obsługującego instrukcję *number of leading zeros* oraz licząc operację *mnożenia* jako pojedynczą instrukcję.

Listing 11.12. *Całkowitoliczbowy logarytm o podstawie 10 z logarytmu o podstawie 2, przegląd jednej tablicy*

```
static unsigned table2[11] = {0, 9, 99, 999, 9999,
   99999, 999999, 9999999, 99999999, 999999999,
   0xFFFFFFFF};

y = (19*(31 - nlz(x))) >> 5;
if (x > table2[y+1]) y = y + 1;
return y;
```

Można go zamienić na wersję pozbawioną rozgałęzień ale ponownie pojawia się trudność w przypadku dużych wartości x ($x > 2^{31} + 10^9$), którą można obejść na dwa sposoby. Jeden z nich polega na wykorzystaniu innego mnożnika (19/64) oraz nieco rozszerzonej wersji tablicy. Taki program przedstawiono na listingu 11.13 (około 11 instrukcji w przypadku procesora RISC obsługującego instrukcję *number of leading zeros* oraz licząc operację *mnożenia* jako pojedynczą instrukcję).

Listing 11.13. *Całkowitoliczbowy logarytm o podstawie 10 z logarytmu o podstawie 2, przegląd jednej tablicy, brak rozgałęzień*

```
int ilog10(unsigned x) {
   int y;
   static unsigned table2[11] = {0, 9, 99, 999, 9999,
      99999, 999999, 9999999, 99999999, 999999999,
      0xFFFFFFFF};

   y = (19*(31 - nlz(x))) >> 6;
   y = y + ((table2[y+1] - x) >> 31);
   return y;
}
```

Inna „poprawka" polega na dodaniu instrukcji `or x` do wyniku odejmowania w celu wymuszenia ustawienia bitu znaku dla $x \geq 2^{31}$; oznacza to, że należy zmienić drugi wiersz części wykonywalnej programu z listingu 11.12 na

```
y = y + (((table2[y+1] - x) | x) >> 31);
```

Jest to preferowany sposób postępowania w przypadku, gdy mnożenie przez 19 jest znacznie trudniejsze do wykonania niż mnożenie przez 9 (na przykład za pomocą sekwencji instrukcji *przesunięcia* i *dodawania*).

W przypadku maszyny 64-bitowej powinno wystarczyć wybranie wartości

$$c > \frac{0{,}30103 \cdot 64 - 1}{64 - 1} \approx 0{,}28993.$$

Wartość 19/64 = 0,296875 jest także prawidłowa, ale doświadczenia pokazują, że żadna inna mniej dokładna wartość nie jest prawidłowa. Program (wersja bez rozgałęzień) ma postać

```
unsigned table2[20] = {0, 9, 99, 999, 9999, ....,
   9999999999999999999};
y = ((19*(63 - nlz(x))) >> 6;
y = y + ((table2[y+1] - x) >> 63);
return y;
```

Rozdział 12.

Niezwykłe podstawy systemów liczbowych

W niniejszym rozdziale zostanie omówionych kilka niezwykłych pozycyjnych systemów liczbowych. Stanowią one interesujące ciekawostki i prawdopodobnie nie będą miały praktycznego znaczenia. Rozważania zostaną ograniczone do liczb całkowitych, ale można je rozszerzyć, tak aby uwzględniały część ułamkową — co zwykle, choć nie zawsze, oznacza przejście do dziedziny liczb niecałkowitych.

12.1. Podstawa –2

Poprzez użycie jako podstawy wartości –2 zarówno dodatnie, jak i ujemne liczby całkowite można wyrazić bez konieczności stosowania jawnego znaku lub pewnych nieregularności, takich jak ujemna waga najbardziej znaczącego bitu (Knu3). Używanymi cyframi są 0 i 1, tak jak w przypadku podstawy +2. Oznacza to, że wartość reprezentowana przez ciąg jedynek i zer jest interpretowana jako

$$(a_n \ldots a_3\, a_2\, a_1\, a_0) = a_n(-2)^n + \ldots + a_3(-2)^3 + a_2(-2)^2 + a_1(-2)^1 + a_0.$$

Widać więc, że procedura znajdowania reprezentacji liczby całkowitej w systemie o podstawie –2 („ujemnobinarnym") polega na stopniowym dzieleniu liczby przez –2 i zachowywaniu reszt. Dzielenie musi zawsze dawać resztę 0 lub 1 (używane cyfry), co oznacza, że musi to być dzielenie modulo. Przykładowo, poniżej przedstawiono sposób znajdowania reprezentacji w systemie o podstawie –2 liczby –3.

$$\frac{-3}{-2} = 2 \text{ rem } 1$$

$$\frac{2}{-2} = -1 \text{ rem } 0$$

$$\frac{-1}{-2} = 1 \text{ rem } 1$$

$$\frac{1}{-2} = 0 \text{ rem } 1$$

Ponieważ osiągnęliśmy iloraz o wartości 0, proces zostaje zakończony (w razie jego kontynuowania wszystkie pozostałe ilorazy i reszty miałyby wartości 0). Zatem, odczytując od końca obliczone reszty, możemy stwierdzić, że liczba –3 w systemie o podstawie –2 posiada zapis 1101.

W tabeli 12.1 (w lewej części) przedstawiono, w jaki sposób są interpretowane wzorce bitów od 0000 do 1111 w systemie o podstawie –2. W prawej części tabeli przedstawiono sposób reprezentacji liczb całkowitych od –15 do 15.

Tabela 12.1. *Konwersje między systemem dziesiętnym a systemem o podstawie –2*

n (podstawa –2)	*n* (dziesiętnie)	*n* (dziesiętnie)	*n* (podstawa –2)	–*n* (podstawa –2)
0	0	0	0	0
1	1	1	1	11
10	–2	2	110	10
11	–1	3	111	1101
100	4	4	100	1100
101	5	5	101	1111
110	2	6	11010	1110
111	3	7	11011	1001
1000	–8	8	11000	1000
1001	–7	9	11001	1011
1010	–10	10	11110	1010
1011	–9	11	11111	110101
1100	–4	12	11100	110100
1101	–3	13	11101	110111
1110	–6	14	10010	110110
1111	–5	15	10011	110001

Nie jest rzeczą oczywistą, że 2^n możliwych wzorców bitów w n-bitowym słowie jednoznacznie reprezentuje wszystkie liczby całkowite z określonego przedziału, ale można to wykazać poprzez indukcję. Hipoteza indukcyjna określa, że n-bitowe słowo reprezentuje wszystkie liczby całkowite z przedziałów

$$\text{od } -(2^{n+1}-2)/3 \text{ do } (2^n-1)/3 \text{ dla } n \text{ parzystego,} \tag{1a}$$

$$\text{od } (-(2^n-2)/3) \text{ do } ((2^{n+1}-1)/3) \text{ dla } n \text{ nieparzystego.} \tag{1b}$$

Załóżmy najpierw, że n jest parzyste. Dla $n = 2$ możliwe do reprezentowania liczby całkowite w systemie o podstawie –2 to 10, 11, 00 oraz 01, czyli

$$-2, -1, 0, 1.$$

Zgadza się to z przedziałem (1a) i każda liczba całkowita z tego przedziału jest reprezentowana tylko raz.

Słowo o rozmiarze $n + 1$ bitów może, przy zerowym bicie wiodącym, reprezentować wszystkie liczby całkowite określone w przedziale (1a). Ponadto, przy bicie wiodącym o wartości 1, może ono reprezentować wszystkie liczby całkowite przesunięte o $(-2)^n = 2^n$. Nowy przedział to

$$\text{od } 2^n - (2^{n+1} - 2)/3 \text{ do } 2^n + (2^n - 1)/3$$

lub

$$\text{od } (2^n - 1)/3 + 1 \text{ do } (2^{n+2} - 1)/3.$$

Jest to przedział przyległy do przedziału (1a), tak więc w przypadku słowa o rozmiarze $n + 1$ bitów wszystkie liczby całkowite z przedziału

$$\text{od } -(2^{n+1} - 2)/3 + 1 \text{ do } (2^{n+2} - 1)/3$$

są reprezentowane tylko raz. Zgadza się to z przedziałem (1b), przy n zastąpionym $n + 1$.

Dowód, że (1a) wynika z (1b), dla n nieparzystego, oraz że wszystkie liczby całkowite w odpowiednim przedziale są reprezentowane unikatowo, ma podobną postać.

W przypadku dodawania i odejmowania wciąż, oczywiście, mają zastosowanie znane reguły, na przykład $0 + 1 = 1$ lub $1 - 1 = 0$. Ponieważ 2 jest zapisywane jako 110, zaś -1 jako 11 itd., mają zastosowanie poniższe reguły. Wraz ze wspomnianymi oczywistymi regułami wystarczają one do wykonywania wszystkich obliczeń.

$$1 + 1 = 110$$
$$11 + 1 = 0$$
$$1 + 1 + 1 = 111$$
$$0 - 1 = 11$$
$$11 - 1 = 10$$

W czasie dodawania lub odejmowania niekiedy mogą wystąpić dwa bity przeniesienia. Powinny one zostać *dodane* do odpowiadających im kolumn, nawet w przypadku odejmowania. Wygodną konwencją jest umieszczenie ich obu nad kolejnym bitem z lewej strony i uproszczenie (kiedy to możliwe) za pomocą reguły $11 + 1 = 0$. Jeżeli 1 jest przenoszone do kolumny zawierającej dwa 0, należy zapisać 1 i przenieść 1 dalej. Poniżej przedstawiono adekwatne przykłady.

```
          Dodawanie                          Odejmowanie
  11 1 11    11                   1 11  1     1
     1  0  1  1  1    19                1  0  1  0  1    21
  + 1 1  0  1  0  1 +(-11)          - 1  0  1  1  1  0 -(-38)
   ---------------- ------            ---------------- ------
   0  1  1  0  0  0     8          1  0  0  1  1  1  1    59
```

Jedyne możliwe przeniesienia to 0, 1 oraz 11. Przepełnienie występuje wtedy, gdy następuje przeniesienie (1 lub 11) z najbardziej znaczącej pozycji. Uwagi te mają zastosowanie zarówno w przypadku dodawania, jak i odejmowania.

Z uwagi na fakt, że istnieją trzy możliwości przeniesienia, sumator dla liczb o podstawie –2 byłby bardziej złożony od sumatora liczb zapisanych w systemie uzupełnienia dwójkowego.

Istnieją dwie możliwości zanegowania liczby. Można ją dodać do siebie po przesunięciu w lewo o jedną pozycję (to znaczy pomnożyć przez –1) lub odjąć ją od 0. Nie istnieje żadna prosta reguła analogiczna do mającej zastosowanie w arytmetyce uzupełnienia dwójkowego, „uzupełnij i dodaj 1". W arytmetyce uzupełnienia dwójkowego reguła ta jest używana w celu konstruowania układów odejmujących na podstawie sumatorów (w celu obliczenia $\boldsymbol{A} - \boldsymbol{B}$ wykonujemy działanie $\boldsymbol{A} + \overline{\boldsymbol{B}} + \mathbf{1}$). W przypadku systemu o podstawie –2 nie istnieje prawdopodobnie żaden podobny, równie prosty układ.

Mnożenie liczb całkowitych w systemie o podstawie –2 jest proste. Korzysta się jedynie z reguł mówiących, że 1 * 1 = 1 oraz 0 razy 0 lub 1 wynosi 0, a następnie dodaje kolumny korzystając z reguł dodawania obowiązujących w tym systemie.

Jednak dzielenie jest dość skomplikowanym działaniem. Opracowanie odpowiedniego algorytmu dla dzielenia sprzętowego (czyli takiego, którego działanie bazuje na wielokrotnych operacjach *odejmowania* i *przesunięcia*) stanowi nie lada wyzwanie. Na listingu 12.1 przedstawiono algorytm przeznaczony dla maszyn 8-bitowych. Wykonuje on dzielenie typu modulo (z nieujemną resztą).

Listing 12.1. *Dzielenie liczb w systemie o podstawie –2*

```
int divbm2(int n, int d) {          // q = n/d w systemie o podstawie -2.
   int r, dw, c, q, i;

   r = n;                           // Inicjalizacja reszty.
   dw = (-128)*d;                   // Pozycja d.
   c = (-43)*d;                     // Inicjalizacja wartości porównawczej.
   if (d > 0) c = c + d;
   q = 0;                           // Inicjalizacja ilorazu.
   for (i = 7; i >= 0; i--) {
      if (d > 0 ^ (i&1) == 0 ^ r >= c) {
         q = q | (1 << i);          // Ustawienie bitu ilorazu.
         r = r - dw;                // Odjęcie d przesuniętego.
      }
      dw = dw/(-2);                 // Pozycja d.
      if (d > 0) c = c - 2*d;       // Ustawienie wartości porównawczej dla
      else c = c + d;               // kolejnej iteracji.
      c = c/(-2);
   }
   return q;                        // Zwrócenie ilorazu jako
                                    // liczby w systemie o podstawie -2.
                                    // Reszta znajduje się w r.
}                                   // 0 <= r < |d|.
```

Choć przedstawiony program napisano w języku C i przetestowano na maszynie pracującej w arytmetyce uzupełnienia dwójkowego, nie ma to znaczenia — kod ten należy postrzegać w kategoriach abstrakcji. Wartości wejściowe n i d oraz wszystkie zmienne

wewnętrzne oprócz `q` stanowią po prostu liczby bez jakiejś konkretnej reprezentacji. Wartość wyjściowa `q` jest ciągiem bitów, które należy interpretować jako liczbę w systemie o podstawie –2.

Całość wymaga pewnych wyjaśnień. Gdyby wartości wejściowe znajdowały się w postaci odpowiadającej systemowi o podstawie –2, algorytm byłoby bardzo trudno zapisać w postaci wykonawczej. Przykładowo, test `if (d > 0)` musiałby polegać na sprawdzeniu czy najbardziej znaczący bit wartości `d` znajduje się na parzystej pozycji. Dodawanie `c = c + d` musiałoby być dodawaniem wartości w systemie o podstawie –2. Taki kod byłby bardzo trudny do analizowania. Sposób, w jaki zapisano kod każe postrzegać wartości `d` i `n` jako nie posiadające żadnej konkretnej reprezentacji. Kod przedstawia konieczne do wykonania operacje arytmetyczne, bez względu na używany zapis wartości liczbowych. Jeżeli liczby są zapisane w systemie o podstawie –2, jak byłoby w przypadku rozwiązania sprzętowego, implementującego przedstawiony algorytm, to mnożenie przez wartość –128 stanowi przesunięcie w lewo o siedem pozycji, zaś dzielenie przez –2 oznacza przesunięcie w prawo o jedną pozycję.

Przykładowo, kod oblicza odpowiednie wartości w sposób następujący:

divbm2(6, 2) = 7 (sześć dzielone przez dwa to 111_{-2})

divbm2(–4, 3) = 2 (minus cztery dzielone przez trzy to 10_{-2})

divbm2(–4, –3) = 6 (minus cztery dzielone przez minus trzy to 110_{-2})

Instrukcja `q = q | (1 << i);` reprezentuje po prostu ustawienie bitu `i` wartości `q`. Kolejny wiersz — `r = r - dw` — reprezentuje redukcję reszty o dzielnik `d` przesunięty w lewo.

Algorytm jest trudno opisać w szczegółach, ale poniżej zostanie przedstawiona ogólna idea jego działania.

Rozważmy problem określenia wartości pierwszego bitu ilorazu, czyli bitu 7. wartości `q`. W systemie o podstawie –2 liczby 8-bitowe, których najbardziej znaczący bit jest ustawiony, należą do przedziału od –170 do –43. Stąd, ignorując możliwość wystąpienia przepełnienia, pierwszy (najbardziej znaczący) bit ilorazu będzie miał wartość 1 wtedy i tylko wtedy, gdy iloraz będzie algebraicznie mniejszy lub równy –43.

Z uwagi na fakt, że $n = qd + r$ oraz dla dodatniego dzielnika $r \le d - 1$, dla dodatniego dzielnika pierwszy bit ilorazu będzie równy 1, jeżeli $n \le -43d + (d - 1)$ lub $n < -43d + d$. Dla ujemnego dzielnika pierwszy bit ilorazu będzie równy 1, jeżeli $n \ge -43d$ ($r \ge 0$ dla dzielenia modulo).

Stąd pierwszy bit ilorazu jest równy 1, jeżeli

$$(d > 0 \,\&\, \neg(n \ge -43d + d)) \mid (d < 0 \,\&\, n \ge -43d).$$

Ignorując możliwość wystąpienia sytuacji $d = 0$, wyrażenie to można zapisać jako

$$d > 0 \oplus n \ge c,$$

gdzie $c = -43d + d$, jeżeli $d > 0$, oraz $c = -43d$, jeżeli $d < 0$.

W ten sposób określa się wartości bitów ilorazu, znajdujących się na pozycjach nieparzystych. W przypadku pozycji parzystych stosuje się odwrotne zasady. Zatem test zawiera składnik `(i&1) == 0` (użyty w programie znak ^ oznacza operację *różnicy wykluczającej — exclusive or*).

W każdej iteracji zmiennej `c` jest przypisywana wartość najmniejszej (najbliższej zeru) liczby całkowitej, która musi zawierać bit o wartości 1 na pozycji `i` po podzieleniu przez `d`. Jeżeli bieżąca wartość reszty `x` przekracza ją, bit `i` wartości `q` jest ustawiany na 1, zaś `r` jest korygowane poprzez odjęcie wartości 1 na tej pozycji, mnożąc przez dzielnik `d`. Nie ma potrzeby wykonywania faktycznej operacji *mnożenia*; należy jedynie `d` odpowiednio ustawić i odjąć.

Algorytm nie jest zbyt elegancki. Jego implementacja nastręcza pewnych trudności, ponieważ zawiera kilka operacji *dodawania*, *odejmowania* i *porównywania*, jak również *mnożenia* (przez stałą), które należy wykonać na początku. Można by przypuszczać, że istnieje pewien algorytm „uniwersalny" — taki, który nie sprawdza znaków argumentów i wykonuje różne działania w zależności od wyników. Jednak algorytm taki prawdopodobnie nie istnieje w przypadku liczb w systemie o podstawie –2 (lub w przypadku arytmetyki uzupełnienia dwójkowego). Wynika to z faktu, że takie dzielenie ze swej istoty jest procesem niejednorodnym. Rozważmy najprostszy algorytm oparty na *przesuwaniu* i *odejmowaniu*. Taki algorytm w ogóle nie wykonywałby przesuwania, ponieważ dla argumentów dodatnich wielokrotnie odejmowałby po prostu dzielnik od dzielnej, zliczając liczbę wykonywanych operacji *odejmowania* do momentu aż reszta przyjmie wartość mniejszą od dzielnika. Jeśli jednak dzielna jest ujemna (a dzielnik dodatni), proces polega na wielokrotnym dodawaniu dzielnika do momentu aż reszta przyjmie wartość 0 lub większą; iloraz ma wówczas wartość przeciwną do otrzymanej liczby zliczeń. Proces różni się w przypadku, gdy to dzielnik jest ujemny.

Pomimo tego operacja *dzielenia jest* procesem jednorodnym w przypadku reprezentacji liczb jako modułu ze znakiem. W przypadku takiej reprezentacji moduły są dodatnie, tak więc algorytm może po prostu odejmować moduły i zliczać te operacje do momentu aż reszta przyjmie wartość ujemną, a następnie ustawić bit znaku ilorazu jako wynik funkcji różnicy wykluczającej z argumentów. Następnie bit znaku reszty jest ustawiany jako równy znakowi dzielnej (co sprowadza się do zwykłego dzielenia z obcinaniem).

Przedstawiony powyżej algorytm można, w pewnym sensie, uczynić nieco bardziej jednorodnym poprzez wykonanie najpierw uzupełnienia dzielnika, jeżeli jest ujemny, a następnie wykonanie zaproponowanych działań, ale dla nieco uproszczonego przypadku, gdzie $d > 0$. Poprawka otrzymanego wyniku byłaby wykonywana na końcu. W przypadku dzielenia z modułem poprawka polega na zanegowaniu ilorazu i pozostawieniu reszty bez zmian. To powoduje przesunięcie części testów poza pętlę, jednak algorytm jako całość wciąż nie jest zanadto elegancki.

Interesującą rzeczą jest zestawienie ze sobą najczęściej używanych reprezentacji liczb z reprezentacją o podstawie –2 w kwestii tego, czy układy komputerowe traktują liczby w sposób jednorodny, czy nie, wykonując cztery podstawowe działania arytmetyczne. Nie posiadamy ścisłej definicji „jednorodności", jednak ogólnie rzecz biorąc oznacza ona brak operacji, które mogą ale nie muszą być wykonywane w zależności od znaków argumentów. Operację polegającą na ustawieniu wartości bitu znaku wyniku na podstawie funkcji różnicy wykluczającej znaków argumentów traktujemy jako jednorodną. W tabeli 12.2 przedstawiono, w przypadku których operacji operandy są traktowane jednorodnie dla różnych reprezentacji liczbowych.

Tabela 12.2. *Operacje jednorodne w różnych systemach kodowania*

	Znak-moduł	**Uzupełnienie jedynkowe**	**Uzupełnienie dwójkowe**	**Podstawa –2**
dodawanie	nie	tak	tak	tak
odejmowania	nie	tak	tak	tak
mnożenie	tak	nie	nie	tak
dzielenie	tak	nie	nie	nie

Dodawanie i odejmowanie w systemie uzupełnienia jedynkowego są wykonywane jednorodnie za pomocą sztuczki polegającej na wykorzystaniu „przeniesienia cyklicznego". W przypadku dodawania wszystkie bity, w tym bit znaku, są dodawane w normalny dla systemu binarnego sposób a przeniesienie ze skrajnie lewego bitu (bitu znaku) jest dodawane do najmniej znaczącej pozycji. Proces ten zawsze kończy się od razu (to znaczy, dodanie przeniesienia nie może spowodować wygenerowania kolejnego przeniesienia z pozycji bitu znaku).

W przypadku uzupełnienia dwójkowego i operacji *mnożenia* odpowiedź na postawione pytanie o jednorodność brzmi twierdząco tylko wówczas, gdy potrzebna jest tylko prawa połowa iloczynu o długości dwóch słów.

Omówienie systemu liczbowego o podstawie –2 zakończymy przedstawieniem pewnych uwag odnośnie do sposobów konwertowania między zapisem w zwykłym systemie binarnym a systemie o podstawie –2.

W celu przekonwertowania liczby z zapisu w systemie o podstawie –2 do zapisu binarnego tworzymy słowo, które zawiera tylko bity o dodatnich wagach i odejmujemy od niego słowo, które zawiera tylko bity o ujemnych wagach, korzystając z reguł odejmowania arytmetyki binarnej. Alternatywna metoda, która może okazać się nieco prostsza, polega na wyodrębnieniu bitów występujących na pozycjach o ujemnych wagach, przesunięcie ich o jedną pozycję w lewo i odjęcie wyodrębnionej liczby od liczby początkowej przy użyciu reguł odejmowania zwykłej arytmetyki binarnej.

W celu przekonwertowania liczby do zapisu w systemie o podstawie –2 z zapisu binarnego należy wyodrębnić bity występujące na nieparzystych pozycjach (pozycjach o wagach 2^n dla n nieparzystego), przesunąć je o jedną pozycję w lewo i dodać do siebie obie liczby przy użyciu reguł dodawania w systemie o podstawie –2. Oto dwa przykłady:

Z systemu binarnego do systemu o podstawie –2

```
    110111  (-13)
  - 1 0 1   (odejmowanie binarne)
  ---------
...111110011 (-13)
```

Z systemu o podstawie –2 do systemu binarnego

```
    110111  (55)
  + 1 0 1   (dodawanie dla podstawy -2)
  ---------
    1001011 (55)
```

Na komputerze o stałym rozmiarze słowa omówione konwersje działają w przypadku liczb ujemnych, jeżeli przeniesienia z najbardziej znaczącej pozycji są po prostu ignorowane. Przykładowo, przedstawiona powyżej po prawej stronie konwersja może być traktowana jako zamiana liczby –9 z postaci binarnej do postaci w systemie o podstawie –2, jeżeli długość słowa wynosi sześć bitów.

Powyższego algorytmu konwertowania do systemu o podstawie –2 nie można w prosty sposób zaimplementować programowo na komputerze binarnym, ponieważ wymaga on wykonywanie dodawania w systemie o podstawie –2. Schroeppel [HAK, item 128] zdołał pokonać ten problem za pomocą znacznie bardziej przemyślanej i przydatnej metody wykonywania konwersji w obu kierunkach. Jego metoda konwersja do postaci binarnej to

$$\boldsymbol{B} \leftarrow (\boldsymbol{N} \oplus \mathbf{0b10\ldots1010}) - \mathbf{0b10\ldots1010}.$$

Aby stwierdzić, że przedstawiona metoda jest poprawna, przyjmijmy, że liczba w systemie o podstawie –2 składa się z czterech cyfr *abcd*. Interpretowana (błędnie) bezpośrednio w postaci binarnej liczba ta ma wartość $8a + 4b + 2c + d$. Po wykonaniu operacji *exclusive or* interpretacja binarna ma postać $8(1 - a) + 4b + 2(1 - c) + d$. Po odjęciu (binarnie) $8 + 2$, przyjmuje ona postać $-8a + 4b - 2c + d$, która jest odpowiednią wartością interpretowaną w systemie o podstawie –2.

Wzór Schroeppela można z łatwością rozwiązać dla $\boldsymbol{N}$ względem $\boldsymbol{B}$, co da metodę konwertowania w drugą stronę, składającą się z trzech instrukcji. Podsumowując, posiadamy następujące wzory pozwalające na konwertowanie do postaci binarnej w przypadku maszyny 32-bitowej:

$$\boldsymbol{B} \leftarrow (\boldsymbol{N} \,\&\, \mathbf{0x55555555}) - (\boldsymbol{N} \,\&\, \neg\mathbf{0x55555555}),$$
$$\boldsymbol{B} \leftarrow \boldsymbol{N} - ((\boldsymbol{N} \,\&\, \mathbf{0xAAAAAAAA}) << 1),$$
$$\boldsymbol{B} \leftarrow (\boldsymbol{N} \oplus \mathbf{0xAAAAAAAA}) - \mathbf{0xAAAAAAAA},$$

oraz poniższy, pozwalający na konwertowanie do systemu o podstawie –2:

$$\boldsymbol{N} \leftarrow (\boldsymbol{B} + \mathbf{0xAAAAAAAA}) \oplus \mathbf{0xAAAAAAAA}).$$

12.2. Podstawa –1 + i

W przypadku użycia jako podstawy wartości $-1 + i$, gdzie i jest równe $\sqrt{-1}$, wszystkie zespolone liczby całkowite (liczby zespolone posiadające część rzeczywistą oraz część urojoną) mogą wyrazić pojedynczą „liczbę" bez konieczności jawnego stosowania znaku

lub innej formy nieregularności. Co zaskakujące, można tego dokonać używając jedynie 0 i 1 jako cyfr i wszystkie liczby będą reprezentowano unikatowo. Nie będziemy dowodzić tej własności omawianego systemu liczbowego, ale opiszemy go skrótowo.

Nie jest rzeczą całkowicie banalną sposób zapisu liczby całkowitej 2[1]. Jednakże można określić to algorytmicznie poprzez wielokrotne dzielenie 2 przez podstawę i zapamiętywanie reszt. Pojawia się pytanie, co w takim kontekście oznacza „reszta"? Chcemy, aby reszta z dzielenia przez $-1 + i$ wynosiła 0 lub 1, jeśli to możliwe. Aby sprawdzić, że jest to zawsze możliwe, załóżmy, że mamy podzielić pewną liczbę zespoloną $a + bi$ przez $-1 + i$. Chcemy znaleźć q i r, takie że q jest liczbą zespoloną, $r = 0$ lub 1 oraz

$$a + bi = (q_r + q_i i))(-1 + i) + r,$$

gdzie q_r oraz q_i oznaczają część rzeczywistą oraz urojoną liczby q. Przyrównanie części rzeczywistej oraz urojonej i rozwiązanie dwóch równań na q daje

$$q_r = \frac{b - a + r}{2},$$

$$q_i = \frac{-a - b + r}{2}.$$

Oczywiście jeżeli liczby a i b są obie parzyste lub obie nieparzyste, to wybranie $r = 0$ powoduje, że q jest liczbą zespoloną. Ponadto, jeżeli jedna z liczb a i b jest parzysta a druga nieparzysta, to wybranie $r = 1$ powoduje, że q jest liczbą zespoloną.

Zatem liczbę całkowitą 2 można przekonwertować do systemu o podstawie $-1 + i$ przy użyciu schematu opisanego poniżej.

Ponieważ zarówno część rzeczywista, jak i urojona liczby 2 jest parzysta, wykonujemy po prostu dzielenie, wiedząc, że resztą będzie 0:

$$\frac{2}{-1+i} = \frac{2(-1-i)}{(-1+i)(-1-i)} = -1 - i \text{ rem } 0.$$

Ponieważ zarówno część rzeczywista, jak i urojona liczby $-1 - i$ jest nieparzysta, ponownie wykonujemy po prostu dzielenie, wiedząc, że resztą będzie 0:

$$\frac{-1-i}{-1+i} = \frac{(-1-i)(-1-i)}{(-1+i)(-1-i)} = i \text{ rem } 0.$$

Ponieważ część rzeczywista liczby i jest parzysta, zaś część urojona jest nieparzysta, reszta wyniesie 1. Najprostszym sposobem jest wzięcie tego faktu pod uwagę na samym początku i odjęcie 1 od dzielnej:

$$\frac{i-1}{-1+i} = 1 \text{ (reszta wynosi 1).}$$

[1] Zainteresowany Czytelnik może spróbować zmierzyć się z tym wyzwaniem.

Ponieważ część rzeczywista liczby 1 jest nieparzysta, zaś urojona parzysta, kolejna reszta wyniesie 1. Odjęcie jej od dzielnej daje:

$$\frac{1-1}{-1+i} = 0 \text{ (reszta wynosi 1).}$$

Doszliśmy do ilorazu równego 0, co oznacza zakończenie działań, a reprezentacja w systemie o podstawie $-1 + i$ liczby 2 to 1100 (odczytując reszty od końca).

W tabeli 12.3 przedstawiono sposób interpretacji każdego wzorca bitowego od 0000 do 1111 w systemie o podstawie $-1 + i$ oraz sposób reprezentacji liczb całkowitych z przedziału od -15 do 15.

Tabela 12.3. *Konwersje między systemem dziesiętnym a systemem o podstawie –2*

n **(podstawa –1 + *i*)**	***n*** **(dziesiętnie)**	***n*** **(dziesiętnie)**	***n*** **(podstawa –1 + *i*)**	**–*n*** **(podstawa –1 + *i*)**
0	0	0	0	0
1	1	1	1	11101
10	$-1+i$	2	1100	11100
11	i	3	1101	10001
100	$-2i$	4	111010000	10000
101	$1-2i$	5	111010001	11001101
110	$-1-i$	6	111011100	11001100
111	$-i$	7	111011101	11000001
1000	$2+2i$	8	111000000	11000000
1001	$3+2i$	9	111000001	11011101
1010	$1+3i$	10	111001100	11011100
1011	$2+3i$	11	111001101	11010001
1100	2	12	100010000	11010000
1101	3	13	100010001	1110100001101
1110	$1+i$	14	100011100	1110100001100
1111	$2+i$	15	100011101	1110100000001

Reguły dodawania w przypadku podstawy $-1 + i$ (oprócz banalnych, dotyczących bitu 0) są następujące:

$$1+1=1100$$
$$1+1+1=1101$$
$$1+1+1+1=111010000$$
$$1+1+1+1+1=111010001$$
$$1+1+1+1+1+1=111011100$$
$$1+1+1+1+1+1+1=111011101$$
$$1+1+1+1+1+1+1+1=111000000$$

W przypadku dodawania dwóch liczb największa liczba przeniesień, występujących w jednej kolumnie to sześć, tak więc największa suma w kolumnie to 8 (11100000). Powoduje to, że sumator musi być dość skomplikowanym urządzeniem. Gdyby ktoś chciał zbudować maszynę arytmetyki liczb złożonych, bez wątpienia najlepszym rozwiązaniem byłoby oddzielne przechowywanie części rzeczywistej i urojonej[2] i reprezentowanie ich w pewien rozsądny sposób, na przykład w postaci uzupełnienia dwójkowego.

12.3. Inne podstawy

Podstawa $-1 - i$ ma zasadniczo te same właściwości co podstawa $-1 + i$ omówiona powyżej. Jeżeli pewien wzorzec bitowy reprezentuje liczbę $a + bi$ w jednej z tych podstaw, to ten sam wzorzec reprezentuje w drugiej podstawie liczbę $a - bi$.

Podstawy $1 + i$ oraz $1 - i$ także mogą reprezentować wszystkie liczby zespolone przy użyciu tylko cyfr 0 i 1. Te dwie podstawy łączą takie same relacje, jak podstawy $-1 \pm i$. W przypadku podstaw $1 \pm i$ reprezentacja niektórych liczb całkowitych zawiera nieskończony ciąg jedynek po lewej stronie, podobnie jak ma to miejsce w przypadku reprezentacji uzupełnienia dwójkowego liczb ujemnych. Wynika to w sposób bezpośredni z jednorodnego charakteru reguł dodawania i odejmowania, podobnie jak ma to miejsce w przypadku uzupełnienia dwójkowego. Jedną z takich liczb całkowitych jest 2, która (w przypadku obu podstaw) ma zapis ...11101100. Stąd podstawy te mają dość skomplikowaną regułę dodawania 1 + 1 = ...11101100.

Grupując w pary bity w reprezentacji o podstawie –2, otrzymuje się reprezentację o podstawie 4 dla liczb dodatnich i ujemnych, wykorzystującą cyfry –2, –1, 0 oraz 1. Przykładowo:

$$-14_{\text{dzies}} = 110110_{-2} = (-1)(1)(-2)_4 = -1 \cdot 4^2 + 1 \cdot 4^1 - 2 \cdot 4^0.$$

Podobnie, grupując w pary bity w reprezentacji liczby zespolonej o podstawie $-1 + i$, otrzymujemy reprezentację liczb zespolonych o podstawie $-2i$, wykorzystującej cyfry 0, 1, $-1 + i$ oraz i. Jest ona jednak zbyt skomplikowana, by mogła budzić zainteresowanie.

Podobnie wygląda sytuacja z systemem *poczwórnie urojonym* (ang. *quater-imaginary*) [Knu2]. Reprezentuje on liczby zespolone używając jako podstawy $2i$ oraz cyfr 0, 1, 2, 3 (bez znaku). W celu przedstawienia pewnych liczb całkowitych a dokładnie tych, których część urojona jest nieparzysta, należy użyć cyfry po prawej stronie punktu podstawy. Przykładowo, i jest zapisywane jako 10.2 w systemie o podstawie $2i$.

[2] W ten sposób postąpiono w Bell Labs w 1940 roku w przypadku Complex Number Calculator (kalkulatora liczb zespolonych) George'a Stibitza [Irvine].

12.4. Najbardziej wydajna podstawa

Załóżmy, że konstruujemy komputer i chcemy zdecydować, jakiej podstawy użyć do reprezentowania liczb całkowitych. Jeśli chodzi o rejestry, to mamy dostępne układy, które są 2-stanowe (binarne), 3-stanowe, 4-stanowe i tak dalej. Na co należy się zdecydować?

Przyjmijmy, że koszt b-stanowego układu jest proporcjonalny do b. Zatem układ 3-stanowy kosztuje o 50% więcej od układu binarnego, zaś układ 4-stanowy kosztuje dwa razy więcej od układu binarnego i tak dalej.

Chcemy, aby rejestry były w stanie przechowywać liczby całkowite od 0 pewnej wartości maksymalnej M. Zakodowanie liczb całkowitych od 0 do M w systemie o podstawie b wymaga $\lceil \log_b(M+1) \rceil$ cyfr (na przykład reprezentacja liczb całkowitych od 0 do 999 999 w systemie dziesiętnym wymaga $\log_{10}(1\,000\,000) = 6$ cyfr).

Można by oczekiwać, że koszt rejestru będzie równy iloczynowi wymaganej liczby cyfr oraz kosztu reprezentacji każdej cyfry:

$$c = k \log_b(M+1) \cdot b,$$

gdzie c oznacza koszt rejestru, zaś k jest stałą proporcjonalności. Dla danego M chcemy znaleźć b minimalizujące koszt.

Minimum tej funkcji występuje dla tej wartości b, przy której $dc/db = 0$. Stąd mamy

$$\frac{d}{db}(kb\log_b(M+1)) = \frac{d}{db}\left(kb\frac{\ln(M+1)}{\ln b}\right) = k\ln(M+1)\frac{\ln b - 1}{(\ln b)^2}.$$

Wartość 0 występuje dla $\ln b = 1$, czyli $b = e$.

Nie jest to zbyt zadowalający rezultat. Ze względu na fakt, że $e \approx 2{,}718$, najbardziej wydajnymi podstawami całkowitoliczbowymi muszą być 2 i 3. Która wartość jest wydajniejsza? Stosunek kosztu rejestru dla systemu o podstawie 2 do kosztu rejestru dla systemu o podstawie 3 wynosi

$$\frac{c(2)}{c(3)} = \frac{k \cdot 2\log_2(M+1)}{k \cdot 3\log_2(M+1)} = \frac{2\ln(M+1)/\ln(2)}{3\ln(M+1)/\ln(3)} = \frac{2\ln 3}{3\ln 2} \approx 1{,}056.$$

Zatem podstawa 2 jest bardziej kosztowna od podstawy 3, ale tylko o niewielką wartość.

Podobna analiza może wykazać, że podstawa 2 jest kosztowniejsza od podstawy e o współczynnik wynoszący około 1,062.

Rozdział 13.

Kod Graya

13.1. Kod Graya

Czy istnieje możliwość przejścia przez wszystkie 2^n kombinacji n bitów zmieniając przy każdym przejściu tylko jeden bit? Odpowiedź na to pytanie brzmi twierdząco i jest to podstawowa właściwość kodów Graya. Oznacza to, że kod Graya stanowi takie kodowanie liczb całkowitych, że zakodowana nim liczba całkowita oraz jej następnik różnią się tylko na jednej pozycji bitów. Koncepcję tę można uogólnić, tak aby miała zastosowanie względem dowolnej podstawy, na przykład dziesiętnej, ale w niniejszym rozdziale zostaną omówione jedynie binarne kody Graya.

Choć istnieje wiele binarnych kodów Graya, zajmiemy się tylko jednym z nich: odbiciowym binarnym kodem Graya (ang. *reflective binary Gray code*). Kod ten jest zwykle tym, o którym mowa w literaturze tematu, kiedy używa się ogólnego terminu „kod Graya". Poniżej zostanie pokazane, zwykle bez dowodów, w jaki sposób należy wykonywać podstawowe operacje na tej reprezentacji liczb całkowitych oraz zostanie przedstawionych ich kilka zaskakujących własności.

Odbiciowy binarny kod Graya konstruuje się w następujący sposób. Rozpoczynamy od ciągów 0 i 1, reprezentujących liczby całkowite 0 i 1:

0

1

Odbijamy je wzdłuż osi poziomej na dole listy i wstawiamy 1 po lewej stronie nowych wartości oraz 0 po lewej stronie oryginalnych wartości:

00

01

11

10

Jest to odbiciowy binarny kod Graya dla $n = 2$. Aby otrzymać kod dla $n = 3$, odbijamy powyższe wartości, dodając 0 lub 1, jak poprzednio:

000
001
011
010
110
111
101
100

Na podstawie tej konstrukcji łatwo wykazać przez indukcję względem n, że (1) każda z 2^n kombinacji bitów pojawia się na liście dokładnie jeden raz, (2) tylko jeden bit ulega zmianie przy przechodzeniu od jednej wartości na liście do następnej oraz (3) tylko jeden bit ulega zmianie w momencie cyklicznego przejścia od ostatniej wartości na liście do pierwszej. Kody Graya posiadające tę ostatnią własność są nazywane cyklicznymi i odbiciowy binarny kod Graya jest z definicji cykliczny.

Jeżeli $n > 2$, istnieją niecykliczne kody, które przyjmują każdą z 2^n wartości dokładnie jeden raz. Jeden z takich kodów ma postać 000 001 011 010 110 100 101 111.

W tabeli 13.1 przedstawiono liczby całkowite dla $n = 4$, zakodowane binarnie oraz kodem Graya. Przedstawione wzory pokazują, w jaki sposób można przechodzić od jednej reprezentacji do drugiej na poziomie poszczególnych bitów (jak miałoby to miejsce w przypadku implementacji sprzętowej).

Jeśli chodzi o liczbę kodów Graya dla n bitów, należy zauważyć, że po obróceniu listy wartości (rozpoczynając od dowolnej spośród 2^n pozycji i przechodząc całą listę) lub reorganizując kolumny otrzymuje się cykliczny binarny kod Graya. Dowolna kombinacja tych operacji daje odrębny kod. Zatem w przypadku n bitów istnieje co najmniej $2^n \cdot n!$ cyklicznych binarnych kodów Graya. Dla $n \geq 3$ liczba ta jest jeszcze większa.

Reprezentacje w kodzie Graya oraz reprezentacja binarna posiadają następujące powiązania, widoczne na podstawie wzorów podanych w tabeli 13.1:

- bit i liczby całkowitej zapisanej w kodzie Graya określa parzystość bitu i oraz bitu znajdującego się po jego lewej stronie w odpowiedniej binarnej liczbie całkowitej (0, jeżeli po lewej stronie i nie ma więcej bitów);
- bit i liczby całkowitej zapisanej w kodzie binarnym określa parzystość wszystkich bitów znajdujących się na lewo od pozycji i w odpowiedniej liczbie całkowitej zapisanej w kodzie Graya.

Konwersji do kodu Graya z kodu binarnego można dokonać w dwóch instrukcjach:

$$G \leftarrow B \oplus (B \overset{u}{>>} 1).$$

Tabela 13.1. *4-bitowy kod Graya oraz wzory na konwersje*

Kod binarny *abcd*	**Kod Graya** *efgh*		
0000	0000	Konwersja z kodu Graya do binarnego	Konwersja z kodu binarnego do Graya
0001	0001		
0010	0011	$e = a$	$a = e$
0011	0010	$f = a \oplus b$	$b = e \oplus f$
0100	0110	$g = b \oplus c$	$c = e \oplus f \oplus g$
0101	0111	$h = c \oplus d$	$d = e \oplus f \oplus g \oplus h$
0110	0101		
0111	0100		
1000	1100		
1001	1101		
1010	1111		
1011	1110		
1100	1010		
1101	1011		
1110	1001		
1111	1000		

Konwersja z kodu binarnego do kodu Graya jest trudniejsza. Jedną z metod określa zależność:

$$\boldsymbol{B} \leftarrow \bigoplus_{i=0}^{n-1} \boldsymbol{G} \overset{u}{>>} \boldsymbol{i}.$$

Wzór ten już się pojawił w podrozdziale *Wyliczanie parzystości słowa* na stronie 94. Jak wówczas wspomniano, wzór ten można rozpisać w sposób następujący (dla $n = 32$):

```
B = G ^ (G >> 1);
B = B ^ (B >> 2);
B = B ^ (B >> 4);
B = B ^ (B >> 8);
B = B ^ (B >> 16);
```

Stąd w ogólności potrzebnych jest $2 \cdot \lceil \log_2 n \rceil$ instrukcji.

Ze względu na łatwość konwersji z kodu binarnego na kod Graya wygenerowanie kolejnych liczb całkowitych zakodowanych w kodzie Graya jest rzeczą banalną:

```
for (i = 0; i < n; i++) {
   G = i ^ (i >> 1);
   output G;
}
```

13.2. Zwiększanie wartości liczb całkowitych zakodowanych w kodzie Graya

Zadanie zwiększenia wartości 4-bitowej *binarnej* liczby całkowitej *abcd* można wyrazić w sposób następujący, stosując notację algebry boolowskiej:

$$d' = \overline{d}$$
$$c' = c \oplus d$$
$$b' = b \oplus cd$$
$$a' = a \oplus bcd$$

Zatem jednym ze sposobów utworzenia sprzętowego licznika w kodzie Graya jest zbudowanie licznika binarnego przy użyciu powyższych zależności i przekonwertowanie wyjść a', b', c', d' na kod Graya poprzez utworzenie różnicy wykluczającej (*exclusive or*) sąsiednich bitów, zgodnie ze wzorami z tabeli 13.1.

Nieco lepszym rozwiązaniem może być następujące, opisane wzorami:

$$p = e \oplus f \oplus g \oplus h$$
$$h' = h \oplus \overline{p}$$
$$g' = g \oplus hp$$
$$f' = f \oplus g\overline{h}p$$
$$e' = e \oplus f\overline{g}\overline{h}p$$

Oznacza to, że przypadek ogólny można zapisać jako

$$G'_n = G_n \oplus (G_{n-1}\overline{G}_{n-2} \ldots \overline{G}_0 p), \; n \geq 2.$$

Bit parzystości p zmienia naprzemiennie swoją wartość między 0 a 1, zatem układ licznika może przechowywać p w odrębnym rejestrze 1-bitowym i po prostu negować go po każdym cyklu zliczania.

W przypadku oprogramowania najlepszym sposobem znalezienia następnika G' liczby całkowitej G zakodowanej w kodzie Graya jest prawdopodobnie zwykłe przekonwertowanie wartości G na kod binarny, zwiększenie słowa binarnego i przekonwertowanie wartości z powrotem na kod Graya. Innym, interesującym sposobem o niemal tak samo dobrym działaniu jest określenie, który bit w G należy zmienić. Wzorzec jest następujący (wyrażony jako słowo mające zostać poddane operacji różnicy wykluczającej z G):

1 2 1 4 1 2 1 8 1 2 1 4 1 2 1 16

Uważny Czytelnik rozpozna ten wzorzec jako maskę, identyfikującą pozycję skrajnie lewego bitu, który ulega zmianie w czasie zwiększania liczb całkowitych 0, 1, 2, 3, ..., odpowiadającego pozycji w powyższej liście. Zatem w celu zwiększenia liczby całkowitej G zapisanej w kodzie Graya określamy pozycję bitu, który ma zostać zmieniony. Definiuje ją skrajnie lewy bit, który zmienia się w momencie dodania 1 do binarnej liczby całkowitej odpowiadającej liczbie G.

Prowadzi to do następujących algorytmów zwiększania liczby całkowitej G, zapisanej w kodzie Graya. Oba rozwiązania w pierwszej kolejności konwertują G do postaci binarnej, którą określa `index(G)`.

Listing 13.1. *Zwiększanie wartości liczby całkowitej w kodzie Graya*

```
B = index(G);                B = index(G);
B = B + 1;                   M = ~B & (B + 1);
Gp = B ^ (B >> 1);           Gp = G ^ M;
```

Metoda zwiększania liczby całkowitej zapisanej w kodzie Graya za pomocą ołówka i kartki papieru jest następująca:

> Rozpoczynając od prawej strony znajdujemy pierwsze miejsce, w którym parzystość bitów leżących na lewo wraz z bieżącym bitem jest parzysta. Zmieniamy wartość bitu na tej pozycji

Równoważny sposób to:

> Niech p będzie parzystością słowa G. Jeżeli p jest parzyste, zmieniamy wartość skrajnie prawego bitu. Jeżeli p jest nieparzyste, zmieniamy wartość bitu leżącego na lewo od pierwszego od prawej strony bitu o wartości 1.

Druga z podanych reguł została bezpośrednio wyrażona równaniem logicznym podanym powyżej.

13.3. Ujemno-binarny kod Graya

Jeżeli zapiszemy liczby całkowite po kolei w systemie o podstawie –2 i przekonwertujemy je za pomocą metody przesuwania i wykonywania operacji różnicy wykluczającej, zamieniającej prostą postać binarną na kod Graya, otrzymamy kod Graya. 3-bitowy kod Graya posiada indeksy obejmujące przedział 3-bitowych liczb w systemie o podstawie –2, czyli od –2 do 5. Podobnie, 4-bitowy kod Graya odpowiada 4-bitowym liczbom w systemie o podstawie –2, czyli od –10 do 5. Nie jest to odbiciowy kod Graya, choć bardzo do niego podobny. 4-bitowy kod Graya można wygenerować, rozpoczynając od wartości 0 i 1, odbijając je wzdłuż osi poziomej na *szczycie* listy, a następnie odbijając otrzymane wartości wzdłuż osi poziomej na *dole* listy i tak dalej. Jest to kod cykliczny.

W celu przekonwertowania takiego kodu Graya z powrotem do postaci w systemie podstawy –2 korzysta się oczywiście z takich samych reguł, jak w przypadku konwertowania na postać binarną ze zwykłego odbiciowego kodu Graya (ponieważ operacje te są wzajemnie odwrotne, nie ma znaczenia sposób interpretacji ciągów bitów).

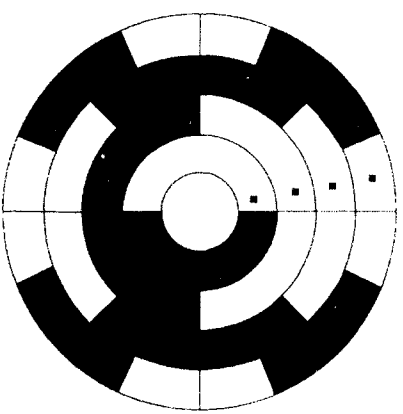
13.4. Rys historyczny i zastosowania

Kody Graya zawdzięczają swoją nazwę Frankowi Grayowi, fizyku z Bell Telephone Laboratories, który w latach 30. XX wieku odkrył metodę używaną obecnie do transmisji kolorowego obrazu telewizyjnego w sposób zgodny z metodami transmisji obrazu czarno-białego, które wówczas istniały. Oznacza to, że kiedy sygnał kolorowy jest odbierany przez odbiornik czarno-biały, obraz jest wyświetlany w formie odcieni szarości.

Martin Gardner [Gard] omawia w swoim artykule zastosowania kodów Graya, w tym układankę chińskich pierścieni, układankę wież z Hanoi oraz drogi Hamiltona w grafach reprezentujących hiperkostki. Przedstawia również sposób konwersji liczby całkowitej z reprezentacji dziesiętnej do dziesiętnego kodu Graya.

Kody Graya są używane w czujnikach położenia. Fragment materiału jest tworzony z obszarów przewodzących i nieprzewodzących, odpowiadających jedynkom i zerom z zapisu liczby całkowitej w kodzie Graya. Każda kolumna zawiera przewodzącą szczotkę drucianą ustawioną w sposób umożliwiający odczyt. Jeżeli szczotka jest ustawiona na linii oddzielającej dwie skwantowane pozycje, tak że jej odczyt jest niejednoznaczny, nie ma znaczenia sposób rozstrzygnięcia tej niejednoznaczności. Może występować tylko jedna niejednoznaczna szczotka i zinterpretowanie jej jako 0 lub 1 określa pozycję sąsiadującą z linią rozdzielającą.

Fragment materiału może też być ciągiem koncentrycznych ścieżek, co umożliwia otrzymanie obrotowego czujnika położenia. W takim przypadku kod Graya musi być cykliczny. Przykład takiego czujnika przedstawiono na rysunku 13.1, gdzie cztery kropki reprezentują szczotki.



Rysunek 13.1. *Obrotowy czujnik położenia*

Rozdział 14.
Krzywa Hilberta

W 1890 roku Giuseppe Peano odkrył krzywą płaską[1] o dość zaskakującej własności *wypełniania przestrzeni* (ang. *space filling*). Krzywa owija się wokół kwadratu jednostkowego i co najmniej jeden raz przechodzi przez punkt (x, y).

Krzywa Peano jest oparta na dzieleniu każdego boku kwadratu jednostkowego na trzy równe części, które dzielą kwadrat na dziewięć mniejszych kwadratów. Jego krzywa przechodzi przez te wszystkie kwadraty w określonej kolejności. Następnie każdy z dziewięciu małych kwadratów jest w podobny sposób dzielony na kolejnych dziewięć jeszcze mniejszych kwadratów i krzywa jest modyfikowana tak, aby przechodziła przez wszystkie kwadraty w określonej kolejności. Krzywą można opisać za pomocą ułamków wyrażonych w systemie o podstawie 3; w rzeczywistości właśnie w taki sposób Peano początkowo je opisał.

W 1891 roku David Hilbert [Hil] odkrył odmianę krzywej Peano, opartą na dzieleniu każdego boku kwadratu jednostkowego na dwie równe części, które dzielą kwadrat na cztery mniejsze kwadraty. Następnie każdy z czterech małych kwadratów jest w podobny sposób dzielony na cztery jeszcze mniejsze kwadraty i tak dalej. Dla każdego etapu takiego podziału Hilbert określił krzywą przechodzącą przez wszystkie kwadraty. Krzywa Hilberta, czasem nazywana krzywą Peano-Hilberta, jest krzywą graniczną takiego procesu podziału. Można ją opisać za pomocą ułamków wyrażonych w systemie o podstawie 2.

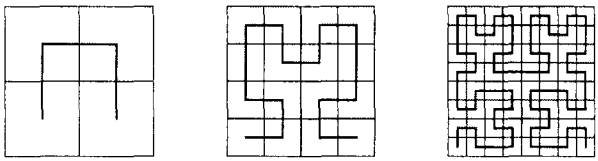
Na rysunku 14.1 przedstawiono pierwsze trzy etapy z sekwencji prowadzącej do utworzenia krzywej Hilberta wypełniającej przestrzeń, tak jak je zobrazowano w pracy z 1891 roku.

My postąpimy nieco inaczej. Będziemy używać terminu „krzywa Hilberta" dla opisu każdej krzywej z sekwencji, której przypadkiem granicznym jest krzywa Hilberta wypełniająca przestrzeń. „Krzywa Hilberta stopnia n" oznacza n-tą krzywą z sekwencji.

---

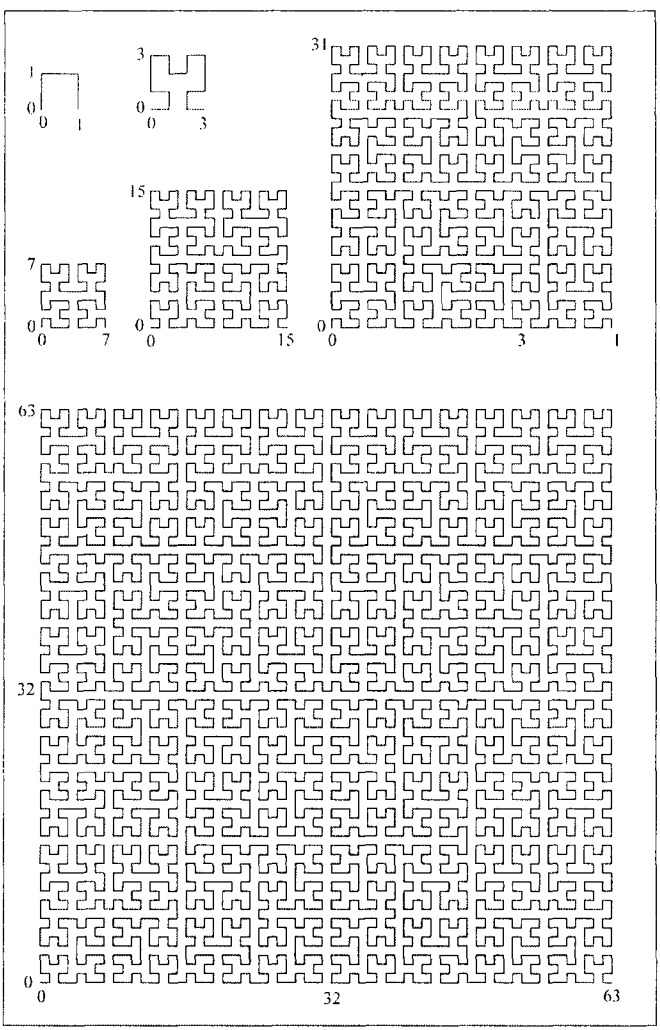
[1] Warto w tym miejscu przypomnieć, że *krzywa* to ciągłe odwzorowanie z przestrzeni jednowymiarowej do przestrzeni n-wymiarowej.

Rysunek 14.1.
Pierwsze trzy krzywe z sekwencji definiującej krzywą Hilberta



Na rysunku 14.1 przedstawiono krzywe stopnia 1., 2. i 3. Przesuwamy krzywe w dół i w lewo, tak aby ich rogi pokrywały się z przecięciami linii w przedstawionych powyżej kwadratach. W końcu skalujemy rozmiar krzywej stopnia n w górę o czynnik 2^n, tak aby współrzędne rogów krzywej były liczbami całkowitymi. Zatem nasza krzywa Hilberta stopnia n posiada rogi w punktach o współrzędnych całkowitoliczbowych od 0 do $2^n - 1$ zarówno na osi x, jak i y. Jako kierunek dodatni wzdłuż krzywej przyjmujemy od $(x, y) = (0, 0)$ do $(2^n - 1, 0)$. Na rysunku 14.2 przedstawiono krzywe Hilberta w naszym rozumieniu stopnia od 1 do 6.

Rysunek 14.2.
Krzywe Hilberta stopnia od 1. do 6.



14.1. Rekurencyjny algorytm generowania krzywej Hilberta

Aby zobaczyć, w jaki sposób można wygenerować krzywą Hilberta, zbadamy krzywe z rysunku 14.2. Krzywa stopnia 1. biegnie do góry, w prawo i w końcu w dół. Krzywa stopnia 2. zachowuje ten ogólny wzorzec. W pierwszej kolejności tworzy ona krzywą w kształcie litery U, która biegnie w górę. Następnie przechodzi w górę o jednostkę. Później tworzy krzywą w kształcie litery U, przechodzi jednostkę i tworzy kolejną krzywą w kształcie litery U, wszystko po prawej stronie. W końcu przechodzi jednostkę w dół oraz tworzy biegnącą w dół krzywą w kształcie litery U.

Krzywa stopnia 1., będąca odwróconą literą U, jest zamieniana na krzywą stopnia 2., posiadającą kształt litery Y.

Krzywą Hilberta dowolnego stopnia można traktować jako ciąg krzywych w kształcie litery U, ustawionych w różnych kierunkach, z których każda oprócz ostatniej poprzedza przesunięcie o jednostkę w określonym kierunku. Podczas przekształcania krzywej Hilberta jednego stopnia w kolejny, każda krzywa w kształcie litery U jest przekształcana na krzywą w kształcie litery Y o tej samej ogólnej orientacji, zaś każdy etap przesunięcia o jednostkę jest przekształcany na przesunięcie w tym samym kierunku.

Przekształcenie krzywej Hilberta stopnia 1. (krzywa w kształcie litery U o końcowym kierunku w prawo oraz orientacji obrotowej zgodnej z ruchem wskazówek zegara) na krzywą Hilberta 2. stopnia przebiega następująco:

1. Narysuj krzywą w kształcie U biegnącą w górę i posiadającą orientację obrotową przeciwną do ruchu wskazówek zegara.
2. Narysuj przejście o jednostkę w górę.
3. Narysuj krzywą w kształcie U biegnącą w prawo i posiadającą orientację obrotową zgodną z ruchem wskazówek zegara.
4. Narysuj przejście o jednostkę w prawo.
5. Narysuj krzywą w kształcie U biegnącą w prawo i posiadającą orientację obrotową zgodną z ruchem wskazówek zegara.
6. Narysuj przejście o jednostkę w dół.
7. Narysuj krzywą w kształcie U biegnącą w dół i posiadającą orientację obrotową przeciwną do ruchu wskazówek zegara.

Można zauważyć, że wszystkie krzywe w kształcie litery U, które mają taką samą orientację, jak krzywa Hilberta stopnia 1., są przekształcane w ten sam sposób. Podobny zestaw reguł można utworzyć dla przekształcania krzywych w kształcie litery U o innej orientacji. Reguły te zawarto w rekurencyjnym programie przedstawionym na listingu 14.1 [Voor]. W programie tym orientacja krzywej U jest charakteryzowana dwiema liczbami całkowitymi, które określają kierunek liniowy oraz obrotowy, kodowany w sposób następujący:

dir = 0: w prawo rot = +1: zgodnie z kierunkiem ruchu wskazówek zegara

dir = 1: w górę rot = –1: przeciwnie do kierunku ruchu wskazówek zegara

dir = 2: w lewo

dir = 3: w dół

Listing 14.1. *Generator krzywej Hilberta*

```
void step(int);

void Hilbert(int dir, int rot, int order) {
   if (order == 0) return;

   dir = dir + rot;
   hilbert (dir, -rot, order - 1);
   step(dir);
   dir = dir - rot;
   hilbert (dir, rot, order - 1);
   step(dir);
   hilbert (dir, rot, order - 1);
   dir = dir - rot;
   step(dir);
   hilbert(dir, -rot, order - 1);
}
```

W rzeczywistości zmienna `dir` może przyjmować dowolne inne wartości, ale znaczenie ma jej kongruencja modulo 4.

Na listingu 14.2 przedstawiono program sterujący oraz funkcję `step`, która jest używana przez program `hilbert`. Programowi przekazuje się informację odnośnie stopnia konstruowanej krzywej Hilberta, który wyświetla listę segmentów linii, podając dla każdej kierunek ruchu, długość wzdłuż krzywej do końca segmentu oraz współrzędne końca segmentu. Przykładowo, dla krzywej Hilberta stopnia 2. wyświetlane informacje to:

```
0   0000   00 00
0   0001   01 00
1   0010   01 01
2   0011   00 01
1   0100   00 10
1   0101   00 11
0   0110   01 11
-1  0111   01 10
0   1000   10 10
1   1001   10 11
0   1010   11 11
-1  1011   11 10
-1  1100   11 01
-2  1101   10 01
-1  1110   10 00
0   1111   11 00
```

Listing 14.2. *Program sterujący dla generatora krzywej Hilberta*

```
#include <stdio.h>
#include <stdlib.h>

int x = -1, y = 0;                // Zmienne globalne.
int i = 0;                        // Odległość wzdłuż krzywej.
int blen;                         // Długość do wydrukowania.

void hilbert(int dir, int rot, int order);

void binary(unsigned k, int len, char *s) {
/* Konwertuje liczbę całkowitą bez znaku k do postaci binarnego
ciągu znaków. Rezultatem jest ciąg s o długości len. */
   int i;

   s[len] = 0;
   for (i = len - 1; i >= 0; i--) {
      if (k & 1) s[i] = '1';
      else       s[i] = '0';
      k = k >> 1;
   }
}
void step(int dir) {
   char ii[33], xx[17], yy[17];

   switch(dir & 3) {
      case 0: x = x + 1; break;
      case 1: y = y + 1; break;
      case 2: x = x - 1; break;
      case 3: y = y - 1; break;
   }
   binary(i, 2*blen, ii);
   binary(x, blen, xx);
   binary(y, blen, yy);
   printf("%5d   %s   %s %s\n", dir, ii, xx, yy);
   i = i + 1;                     // Zwiększenie odległości.
}
int main(int argc, char *argv[]) {
   int order;

   order = atoi(argv[1]);
   blen = order;
   step(0);                       // Wydrukowanie punktu początkowego.
   hilbert(0, 1, order);
   return 0;
}
```

14.2. Określanie współrzędnych na podstawie odległości wzdłuż krzywej Hilberta

W celu znalezienia współrzędnych (x, y) punktu znajdującego się w odległości s wzdłuż krzywej Hilberta stopnia n wystarczy zauważyć, że dwa najbardziej znaczące bity $2n$-bito-

wej liczby całkowitej s określają, w którym z czterech głównych kwadratów znajduje się szukany punkt. Dzieje się tak dlatego, że krzywa Hilberta dowolnego stopnia zachowuje się zgodnie z ogólnym wzorcem krzywej 1. stopnia. Jeżeli dwa najbardziej znaczące bity liczby s to 00, punkt znajduje się gdzieś w lewym dolnym kwadracie; wartość 01 określa, że punkt znajduje się gdzieś w lewym górnym kwadracie, 10 — w prawym górnym, 11 — w prawym dolnym. Zatem dwa najbardziej znaczące bity s określają najbardziej znaczące bity n-bitowych liczb całkowitych x i y w sposób następujący:

Dwa najbardziej znaczące bity liczby *s*	**Najbardziej znaczące bity liczb (*x*, *y*)**
00	(0, 0)
01	(0, 1)
10	(1, 1)
11	(1, 0)

W każdej krzywej Hilberta występują tylko cztery spośród ośmiu możliwych kształtów litery U. Przedstawiono je w tabeli 14.1 jako obiekty graficzne oraz jako odwzorowania dwóch bitów liczby s na pojedynczy bit każdej z liczb x i y.

Tabela 14.1. *Cztery możliwe odwzorowania*

A	**B**	**C**	**D**
00 → (0,0)	00 → (0,0)	00 → (1,1)	00 → (1,1)
01 → (0,1)	01 → (1,0)	01 → (1,0)	01 → (0,1)
10 → (1,1)	10 → (1,1)	10 → (0,0)	10 → (0,0)
11 → (1,0)	11 → (0,1)	11 → (0,1)	11 → (1,0)

Na podstawie rysunku 14.2 można stwierdzić, że we wszystkich przypadkach krzywe w kształcie litery U, reprezentowane przez odwzorowanie A (⊓↓), na kolejnym poziomie uszczegółowienia stają się krzywymi w kształcie litery U, reprezentowanymi przez odwzorowanie B, A, A lub D, w zależności od tego, czy długość przebyta we wspomnianym jako pierwsze odwzorowaniu A wyniosła 0, 1, 2 lub 3. Podobnie krzywe w kształcie litery U reprezentowane przez odwzorowanie B (⊔) na kolejnym poziomie uszczegółowienia stają się krzywymi w kształcie litery U, reprezentowanymi przez odwzorowanie A, B, B lub C, w zależności od tego, czy długość przebyta we wspomnianym jako pierwsze odwzorowaniu B wyniosła 0, 1, 2 lub 3.

Obserwacje te pozwalają na utworzenie tabeli przejść stanów — tabela 14.2 — w której stany odpowiadają odwzorowaniom przedstawionym w tabeli 14.1.

Aby skorzystać z tabeli, należy rozpocząć od stanu A. Liczba całkowita s powinna zostać uzupełniona zerami wiodącymi, tak aby jej długość wynosiła $2n$, gdzie n jest stopniem krzywej Hilberta. Badamy bity liczby s parami od strony lewej do prawej. Pierwszy wiersz tabeli 14.2 oznacza, że jeżeli bieżącym stanem jest A oraz właśnie

Tabela 14.2. *Tabela przejść stanów służąca do obliczania (x, y) na podstawie s*

Jeżeli bieżącym stanem jest	a kolejne dwa bity s (po prawej) to	do (x, y) należy dopisać	i przejść do stanu
A	00	(0, 0)	B
A	01	(0, 1)	A
A	10	(1, 1)	A
A	11	(1, 0)	D
B	00	(0, 0)	A
B	01	(1, 0)	B
B	10	(1, 1)	B
B	11	(0, 1)	C
C	00	(1, 1)	D
C	01	(1, 0)	C
C	10	(0, 0)	C
C	11	(0, 1)	B
D	00	(1, 1)	C
D	01	(0, 1)	D
D	10	(0, 0)	D
D	11	(1, 0)	A

zbadanymi bitami są (0, 0), to wartością wyjściową jest (0, 0) a kolejnym stanem — stan B. Następnie przechodzimy do kolejnych dwóch bitów liczby *s*. Podobnie, drugi wiersz określa, że jeżeli bieżącym stanem jest A, zaś odczytane bity to (0, 1), to wartością wyjściową jest (0, 1) i należy pozostać w stanie A.

Bity wyjściowe są grupowane w kolejności od lewej do prawej. W momencie osiągnięcia końca liczby *s* zostają zdefiniowane *n*-bitowe wielkości określające *x* i *y*.

Jako przykład załóżmy, że $n = 3$ oraz

$$s = 110100.$$

Proces rozpoczyna się w stanie A i pierwsze odczytane bity to 11, na wyjściu pojawia się (1, 0) i przechodzimy do stanu D (czwarty wiersz). Następnie, w stanie D, badamy bity 01 i przekazujemy na wyjście (0, 1), pozostając w stanie D. W końcu na wyjście przekazujemy (1, 1) i przechodzimy do stanu C, choć ten końcowy stan nie ma już znaczenia.

Zatem na wyjście przekazano wartości (101, 011), co oznacza, że $x = 5$ oraz $y = 3$.

Na listingu 14.3 przedstawiono program napisany w języku C, implementujący opisane działania. W programie tym stan bieżący jest reprezentowany przez liczbę całkowitą od 0 do 3, odpowiednio dla stanów od A do D. W przypisaniu do zmiennej `row` bieżący stan jest składany z dwoma kolejnymi bitami `s`, co daje liczbę całkowitą z przedziału

od 0 do 15, która stanowi numer odpowiedniego wiersza w tabeli 14.2. Zmienna row jest używana w celu uzyskania dostępu do liczb całkowitych (wyrażonych jako wartości szesnastkowe), które są używane jako ciągi bitów w celu reprezentowania dwóch prawych kolumn tabeli 14.2. Oznacza to, że takie operacje dostępu stanowią przeglądy tabeli. Interpretacja wartości szesnastkowych od strony lewej do prawej odpowiada interpretacji tabeli 14.2 w kierunku od dołu do góry.

Listing 14.3. *Program obliczający (x, y) na podstawie s*

```
void hil_xy_from_s(unsigned s, int n, unsigned *xp, unsigned *yp) {

   int i;
   unsigned state, x, y, row;

   state = 0;                          // Inicjalizacja.
   x = y = 0;

   for (i = 2*n - 2; i >= 0; i -= 2) {  // Powtórzenie n razy.
      row = 4*state | (s >> i) & 3;     // Wiersz w tabeli.
      x = (x << 1) | (0x936C >> row) & 1;
      y = (y << 1) | (0x39C6 >> row) & 1;
      state = (0x3E6B94C1 >> 2*row) & 3; // Nowy stan.
   }
   *xp = x;                            // Przekazanie
   *yp = y;                            // wyników.
}
```

W pracy [L&S] podano całkowicie odmienny algorytm. W przeciwieństwie do algorytmu z listingu 14.3 bada on bity liczby s od strony prawej do lewej. Jego podstawą jest poczynienie obserwacji, że istnieje możliwość odwzorowania dwóch najmniej znaczących bitów liczby s na wartości (x, y) na podstawie krzywej Hilberta stopnia 1., a następnie sprawdzenie dwóch kolejnych bitów s po lewej stronie. Jeżeli mają one wartości 00, właśnie obliczone wartości x i y powinny zostać zamienione, co odpowiada odbiciu krzywej Hilberta stopnia 1. wzdłuż prostej $x = y$ (patrz krzywe stopnia 1. i 2. na rysunku 14.2 na stronie 270). Jeżeli bity mają wartości 01 lub 10, wartości x i y zostają niezmienione. Jeżeli ich wartości to 11, wartości x i y zostają zamienione i uzupełnione. Te same reguły mają zastosowanie w trakcie posuwania się w lewo wzdłuż bitów wartości s. Przedstawiono je w tabeli 14.3, zaś odpowiedni kod zawarto na listingu 14.4. Interesującą kwestią jest fakt, że bity można najpierw dostawić na początku wartości x i y, a następnie wykonać operacje zamian i uzupełnień, uwzględniając nowo dodane bity — wynik będzie ten sam.

Tabela 14.3. *Metoda Lama i Shapiro obliczania wartości (x, y) na podstawie s*

Jeżeli kolejne (po lewej stronie) dwa bity *s* mają wartość	to	i dopisz do (*x*, *y*)
00	zamień x z y	(0, 0)
01	nie rób nic	(0, 1)
10	nie rób nic	(1, 1)
11	zamień x z y i uzupełnij	(1, 0)

Listing 14.4. *Metoda Lama i Shapiro obliczania (x, y) na podstawie s*

```
void hil_xy_from_s(unsigned s, int n, unsigned *xp, unsigned *yp) {

   int i, sa, sb;
   unsigned x, y, temp;

   for (i = 0; i < 2*n; i += 2) {
      sa = (s >> (i+1)) & 1;        // Pobranie bitu i+1 liczby s.
      sb = (s >> i) & 1;            // Pobranie bitu i liczby s.

      if ((sa ^ sb) == 0) {         // Jeżeli sa,sb = 00 lub 11,
         temp = x;                  // zamieniamy x z y
         x = y^(-sa);               // i jeżeli sa = 1,
         y = temp^(-sa);            // uzupełniamy je.
      }
      x = (x >> 1) | (sa << 31);    // Dodajemy sa na początek x
      y = (y >> 1) | ((sa ^ sb) << 31); // i (sa^sb) na początek y.
   }
   *xp = x >> (32 - n);             // Wyrównujemy do prawej strony
   *yp = y >> (32 - n);             // wartości x i y oraz zwracamy
}                                   // je do programu wywołującego.
```

Na listingu 14.4 zmienne `x` i `y` nie są inicjalizowane, co może spowodować wyświetlenie komunikatu o błędzie w przypadku niektórych kompilatorów. Jednak kod funkcjonuje poprawnie bez względu na początkowe wartości `x` i `y`.

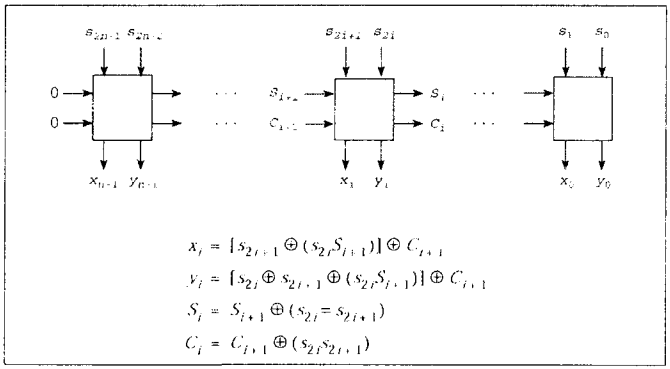
Rozgałęzienia w pętli z listingu 14.4 można uniknąć poprzez wykonanie operacji zamiany bitów za pomocą sztuczki „trzech instrukcji *exclusive or*", opisanej w podrozdziale 2.19 na stronie 58.

Blok instrukcji `if` można zastąpić następującym fragmentem kodu, w którym `swap` i `cmpl` są liczbami całkowitymi bez znaku:

```
swap = (sa ^ sb) - 1;  // -1, jeżeli należy zamieniać, w przeciwnym razie 0.
cmpl = -(sa & sb);     // -1, jeżeli należy uzupełnić, w przeciwnym razie 0.
x = x ^ y;
y = y ^ (x & swap) ^ cmpl;
x = x ^ y;
```

Jednakże oznacza to konieczność wykonania dziewięciu instrukcji w porównaniu z sześcioma w przypadku bloku `if`, tak więc koszt wykonania rozgałęzienia musiałby być dość wysoki, aby mogła to być przydatna modyfikacja.

Pomysł „zamiany i uzupełnienia" pochodzący z pozycji [L&S] sugeruje sposób wykonania układu logicznego, generującego krzywą Hilberta. Podstawą układu opisanego poniżej jest poczynienie obserwacji, że w miarę przechodzenia wzdłuż ścieżki krzywej stopnia n w gruncie rzeczy odwzorowujemy pary bitów wartości s na parę (x, y) zgodnie z odwzorowanie A tabeli 14.1. Jednak wraz z przechodzeniem do różnych obszarów wyniki odwzorowania zostają zamienione, uzupełnione lub zostają wykonane obie te operacje. Układ z rysunku 14.3 śledzi wymagania odnośnie zamiany i uzupełnienia w każdym etapie, wykorzystuje odpowiednie odwzorowania w celu zmapowania dwóch bitów wartości s na parę (x_i, y_i) i generuje sygnały zamiany i uzupełnienia dla następnego etapu.



Rysunek 14.3. *Układ logiczny obliczający (x, y) na podstawie s*

Zakładamy, że istnieje rejestr zawierający długość ścieżki s oraz układy służące do zwiększania jej wartości. W celu znalezienia kolejnego punktu na krzywej Hilberta najpierw zwiększamy wartość s a następnie przekształcamy ją zgodnie z tabelą 14.4. Jest to proces przebiegający od strony lewej do prawej, co stanowi pewien problem, gdyż zwiększanie wartości s jest procesem wykonywanym od strony prawej do lewej. Zatem czas wymagany do wygenerowania nowego punktu na krzywej Hilberta stopnia n jest proporcjonalny do $2n$ (zwiększenie wartości s) plus n (przekształcenie s na (x, y)).

Tabela 14.4. *Sposób obliczania (x, y) na podstawie s*

Jeżeli kolejne (po prawej stronie) dwa bity *s* mają wartość	dopisz do (*x*, *y*)	i ustaw
00	(0, 0)*	swap = $\overline{\text{swap}}$
01	(0, 1)*	bez mian
10	(1, 1)*	bez zmian
11	(1, 0)*	swap = $\overline{\text{swap}}$, cmpl = $\overline{\text{cmpl}}$

* Być może zamienione i (lub) uzupełnione

Na rysunku 14.3 przedstawiono powyższe obliczenia w formie układu logicznego. Symbol S oznacza sygnał zamiany, zaś C — sygnał uzupełnienia.

Układ z rysunku 14.3 sugeruje inny sposób obliczenia wartości (x, y) na podstawie s. Należy zwrócić uwagę na sposób propagacji sygnałów zamiany i uzupełnienia od strony lewej do prawej przez wszystkie n segmentów układu. Sugeruje to, że istnieje możliwość użycia operacji równoległego dodania przedrostka w celu zapewnienia szybkiej (wykonywanej w $\log_2 n$ zamiast $n - 1$ krokach) propagacji sygnałów zamiany i uzupełnienia do każdego segmentu a następnie wykonania pewnych równoległych operacji logicznych na słowach w celu obliczenia x i y, korzystając z równań przedstawionych na rysunku 14.3. Wartości x i y występują na przemian na parzystych i nieparzystych pozycjach słowa, tak więc muszą zostać rozdzielone za pomocą operacji odtasowania (patrz strona 127). Procedura taka może wydawać się nieco skomplikowana i przynosi korzyści raczej dla dużych wartości n, ale poniżej przeanalizujemy jej działanie.

Procedurę działania tej operacji przedstawiono na listingu 14.5 [GLS1]. Działania są wykonywane na wielkościach o rozmiarze słowa, tak więc w pierwszej kolejności następuje dopełnienie s z lewej strony bitami 01. Taka kombinacja bitów nie wpływa na wielkości zamieniane i uzupełniane. Następnie zostaje wyliczona wartość wielkości cs (uzupełnionej i zmienionej). Obliczone słowo ma postać cscs...cs, gdzie każdy bit c (pojedynczy bit) o wartości 1 oznacza, że odpowiednia para bitów ma zostać uzupełniona, zaś każdy bit s o wartości 1 oznacza, że odpowiednia para bitów ma zostać zamieniona — zgodnie z tabelą 14.3. Innymi słowy, te dwie instrukcje odwzorowują każdą parę bitów s w sposób następujący:

s_{2i+1}	s_{2i}	cs
0	0	01
0	1	00
1	0	00
1	1	11

Listing 14.5. *Metoda równoległego dodawania przedrostka w celu obliczenia (x, y) na podstawie s*

```
void hil_xy_from_s(unsigned s, int n, unsigned *xp, unsigned *yp) {

   unsigned comp, swap, cs, t, sr;

   s = s | (0x55555555 << 2*n);  // Dopełnienie s z lewej strony
   sr = (s >> 1) & 0x55555555;   // grupami bitów 01 (brak zmiany).
   cs = ((s & 0x55555555) + sr)  // Określenie informacji o uzupełnieniach
        ^ 0x55555555;            // i zamianach w grupach dwubitowych.
   // Operacja xor równoległego dodania przedrostka propaguje zarówno informację
   // o uzupełnieniu, jak i o zamianie od strony lewej do prawej (nie występuje
   // operacja "cs ^= cs >> 1", tak więc w efekcie są wykonywane dwie niezależne
   // operacje równoległego dodania przedrostka na dwóch przeplatających się
   // zbiorach szesnastu bitów.

   cs = cs ^ (cs >> 2);
   cs = cs ^ (cs >> 4);
   cs = cs ^ (cs >> 8);
   cs = cs ^ (cs >> 16);
   swap = cs & 0x55555555;       // Oddzielenie bitów zamiany
   comp = (cs >> 1) & 0x55555555; // i uzupełnienia.

   t = (s & swap) ^ comp;        // Obliczenie x i y na
   s = s ^ sr ^ t ^ (t << 1);    // nieparzystych i parzystych
                                 // (odpowiednio) pozycjach.
   s = s & ((1 << 2*n) - 1);     // Wyzerowanie wszelkich niepotrzebnych
                                 // po lewej stronie.

   // Teraz należy wykonać operację "odmieszania" w celu oddzielenia bitów x i y.

   t = (s ^ (s >> 1)) & 0x22222222; s = s ^ t ^ (t << 1);
   t = (s ^ (s >> 2)) & 0x0C0C0C0C; s = s ^ t ^ (t << 2);
   t = (s ^ (s >> 4)) & 0x00F000F0; s = s ^ t ^ (t << 4);
   t = (s ^ (s >> 8)) & 0x0000FF00; s = s ^ t ^ (t << 8);

   *xp = s >> 16;                // Przypisanie dwóch połówek
   *yp = s & 0xFFFF;             // t do x i y.
```

Jest to wielkość, względem której chcemy zastosować operację równoległego dodawania przedrostka. Używamy instrukcji PP-XOR, przechodząc od strony lewej do prawej, ponieważ kolejne bity o wartości 1, oznaczające uzupełnienie lub zamianę, mają takie same własności logiczne, jak instrukcja różnicy wykluczającej *exclusive or*: dwa kolejne bity o wartości 1 anulują się nawzajem.

Oba sygnały (uzupełnienia i zamiany) są propagowane w tej samej operacji PP-XOR dla każdego bitu `cs`.

Kolejne cztery instrukcje przypisania dają w efekcie zamianę każdej pary bitów `s` na wartości (`x`, `y`), gdzie wartości `x` znajdują się na nieparzystych (pierwszych od lewej) pozycjach, zaś wartości `y` na parzystych pozycjach bitów. Choć całość może wydawać się nieco niejasna, nie jest rzeczą trudną sprawdzenie, że każda para bitów `s` jest przekształcana zgodnie z pierwszymi dwoma równaniami logicznymi z rysunku 14.3 (sugestia: rozważ oddzielnie sposób przekształcania nieparzystych i parzystych bitów, korzystając z faktu, że `t` i `sr` są równe 0 na pozycjach nieparzystych).

Pozostała część procedury jest bardzo prosta. Wykonuje ona 66 podstawowych instrukcji RISC (na stałych, bez rozgałęzień) w porównaniu z $19n + 10$ (średnio) instrukcjami w przypadku kodu z listingu 14.4 (wartości określone w oparciu o skompilowany kod, wliczając części wstępną i końcową, które zasadniczo są puste). Zatem metoda równoległego dodawania przedrostków jest szybsza dla $n \geq 3$.

14.3. Określanie odległości na podstawie współrzędnych na krzywej Hilberta

Mając dane współrzędne punktu na krzywej Hilberta, odległość od początku krzywej do tego punktu można obliczyć przy użyciu tabeli przejść stanów, podobnej do tabeli 14.2. Przedstawiono ją jako tabelę 14.5.

Jej interpretacja jest podobna do przedstawionej w poprzednim podrozdziale. Najpierw wartości x i y muszą zostać dopełnione zerami wiodącymi, tak aby miały długość n bitów, gdzie n oznacza stopień krzywej Hilberta. Następnie bity wartości x i y są przeglądane od strony lewej do prawej a wartość s jest tworzona w tym samym kierunku.

Na listingu 14.6 przedstawiono program w języku C implementujący opisane działania.

W pracy [L&S] przedstawiono algorytm obliczania s na podstawie (x, y), który jest podobny do analogicznego algorytmu wykonującego obliczenia w przeciwną stronę (tabela 14.3). Jest to algorytm wykonujący działania od strony lewej do prawej i przedstawiono go w tabeli 14.6 oraz na listingu 14.7.

Tabela 14.5. *Tabela przejść stanów w celu obliczenia s na podstawie (x, y)*

Jeżeli bieżącym stanem jest	a kolejne dwa bity (*x*, *y*) (po prawej) to	do *s* należy dopisać	i przejść do stanu
A	(0, 0)	00	B
A	(0, 1)	01	A
A	(1, 0)	11	D
A	(1, 1)	10	A
B	(0, 0)	00	A
B	(0, 1)	11	C
B	(1, 0)	01	B
B	(1, 1)	10	B
C	(0, 0)	10	C
C	(0, 1)	11	B
C	(1, 0)	01	C
C	(1, 1)	00	D
D	(0, 0)	10	D
D	(0, 1)	01	D
D	(1, 0)	11	A
D	(1, 1)	00	C

Listing 14.6. *Program obliczający s na podstawie (x, y)*

```
unsigned hil_s_from_xy(unsigned x, unsigned y, int n) {

   int i;
   unsigned state, s, row;

   state = 0;                          // Inicjalizacja.
   s = 0;

   for (i = n - 1; i >= 0; i--) {
      row = 4*state | 2*((x >> i) & 1) | (y >> i) & 1;
      s = (s << 2) | (0x361E9CB4 >> 2*row) & 3;
      state = (0x8FE65831 >> 2*row) & 3;
   }
   return s;
}
```

Tabela 14.6. *Metoda Lama i Shapiro obliczania wartości s na podstawie (x, y)*

Jeżeli kolejne (po prawej stronie) dwa bity (*x*, *y*) mają wartość	to	i dopisz do *s*
(0, 0)	zamień *x* z *y*	00
(0, 1)	nie rób nic	01
(1, 0)	zamień *x* z *y* i uzupełnij	11
(1, 1)	nie rób nic	10

Listing 14.7. *Metoda Lama i Shapiro obliczania s na podstawie (x, y)*

```
unsigned hil_s_from_xy(unsigned x, unsigned y, int n) {

   int i, xi, yi;
   unsigned s, temp;

   s = 0;                          // Inicjalizacja.
   for (i = n - 1; i >= 0; i--) {
      xi = (x >> i) & 1;           // Pobranie bitu i wartości x.
      yi = (y >> i) & 1;           // Pobranie bitu i wartości y.

      if (yi == 0) {
         temp = x;                 // Zamiana x z y oraz,
         x = y^(-xi);              // jeżeli xi = 1,
         y = temp^(-xi);           // uzupełnienie ich.
      }
      s = 4*s + 2*xi + (xi^yi);    // Dodanie dwóch bitów do s.
   }
   return s;
}
```

14.4. Zwiększanie wartości współrzędnych na krzywej Hilberta

Mając dane współrzędne (x, y) punktu na krzywej Hilberta stopnia n, chcemy znaleźć współrzędne następnego punktu. Jednym ze sposobów jest przekonwertowanie wartości (x, y) na s, dodanie 1 do s i w końcu przekonwertowanie nowej wartości s z powrotem na (x, y), korzystając z algorytmów przedstawionych powyżej.

Nieco lepsza (choć nieznacznie) metoda opiera się na fakcie, że wraz z przechodzeniem wzdłuż krzywej Hilberta, w każdy kroku albo x, albo y, ale nie obydwie te wartości, jest zwiększane lub zmniejszane o 1. Opisywany algorytm przegląda współrzędne od strony lewej do prawej w celu określenia typu krzywej w kształcie litery U, na której znajdują się dwa skrajnie prawe bity. Następnie, na podstawie typu krzywej U oraz wartości obu bitów zostaje zwiększona lub zmniejszona albo wartość x, albo y.

W ogólnym zarysie tak wygląda całość wykonywanych działań, ale powstaje pewna komplikacja w przypadku, gdy ścieżka znajduje się na końcu krzywej U (co zdarza się raz na każde cztery etapy). W takim punkcie kierunek kolejnego przejścia jest określany na podstawie *poprzednich* bitów x i y oraz na podstawie krzywej U wyższego stopnia, z którą są powiązane te bity. Jeżeli taki punkt znajduje się dodatkowo na końcu krzywej U, to poprzednie bity oraz występująca tam krzywa U pozwalają na określenie kierunku kolejnego przejścia i tak dalej.

Tabela 14.7 ilustruje działanie takiego algorytmu. W tabeli tej symbole A, B, C i D określają typy krzywych U, zgodnie z opisem w tabeli 14.1 na stronie 274. Aby użyć tej tabeli, najpierw należy dopełnić x i y zerami wiodącymi, tak aby miały długość n bitów,

gdzie n oznacza stopień krzywej Hilberta. Zaczynamy w stanie A i badamy bity x i y od strony lewej do prawej. Pierwszy wiersz tabeli 14.7 określa, że jeżeli bieżącym stanem jest A oraz właśnie odczytane bity to (0, 0), należy ustawić zmienną, tak aby wskazywała na konieczność zwiększenia wartości y i przejść do stanu B. Inne wiersze należy interpretować w podobny sposób, gdzie znak minusa określa konieczność zmniejszenia wartości odpowiedniej współrzędnej. Pauza w trzeciej kolumnie oznacza, że nie należy zmieniać wartości zmiennej śledzącej wystąpienie konieczności zmiany współrzędnych.

Tabela 14.7. *Przechodzenie o jedną pozycję na krzywej Hilberta*

Jeżeli bieżącym stanem jest	a kolejne dwa bity (x, y) (po prawej) to	należy przygotować się do zwiększenie lub zmniejszenia	i przejść do stanu
A	(0, 0)	$y+$	B
A	(0, 1)	$x+$	A
A	(1, 0)	–	D
A	(1, 1)	$y-$	A
B	(0, 0)	$x+$	A
B	(0, 1)	–	C
B	(1, 0)	$y+$	B
B	(1, 1)	$x-$	B
C	(0, 0)	$y+$	C
C	(0, 1)	–	B
C	(1, 0)	$x-$	C
C	(1, 1)	$y-$	D
D	(0, 0)	$x+$	D
D	(0, 1)	$y-$	D
D	(1, 0)	–	A
D	(1, 1)	$x-$	C

Po przejrzeniu ostatnich (skrajnie prawych) bitów x i y zwiększamy lub zmniejszamy odpowiednią współrzędną, zgodnie z końcowym wskazaniem wartości zmiennej.

Program w języku C, implementujący omówione działania przedstawiono na listingu 14.8. Zmienna dx jest inicjalizowana w ten sposób, że w przypadku wielokrotnego wywołania algorytm jest wykonywany cyklicznie i za każdym razem generuje tę samą krzywą Hilberta (jednakże etap łączący jeden cykl z następnym nie jest krokiem jednostkowym).

Listing 14.8. *Program wykonujący przejście o jedną pozycję na krzywej Hilberta*

```
void hil_inc_xy(unsigned *xp, unsigned *yp, int n) {

  int i;
  unsigned x, y, state, dx, dy, row, dochange;
```

```
x = *xp;
   y = *yp;
   state = 0;                    // Inicjalizacja.
   dx = -((1 << n) - 1);         // Inicjalizacja -(2**n - 1).
   dy = 0;

   for (i = n-1; i >= 0; i--) {          // Wykonanie n razy.
      row = 4*state | 2*((x >> i) & 1) | (y >> i) & 1;
      dochange = (0xBDDB >> row) & 1;
      if (dochange) {
         dx = ((0x16451659 >> 2*row) & 3) - 1;
         dy = ((0x51166516 >> 2*row) & 3) - 1;
      }
      state = (0x8FE65831 >> 2*row) & 3;
   }
   *xp = *xp + dx;
   *yp = *yp + dy;
}
```

Tabelę 14.7 można z łatwością zaimplementować w postaci układu logicznego, co przedstawiono na rysunku 14.4. Pokazane na nim zmienne mają następujące znaczenie:

x_i: Bit i wartości wejściowej x.

y_i: Bit i wartości wejściowej y.

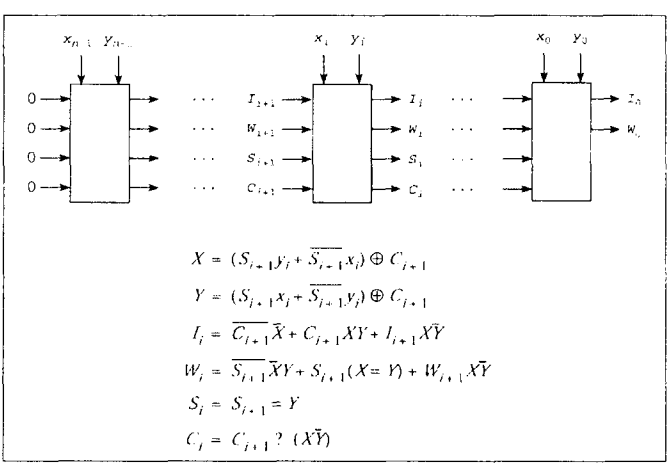
X, Y: x_i i y_i zamienione i uzupełnione, zgodnie z wartościami S_{i+1} i C_{i+1}.

I: Jeżeli równe 1, zwiększenie; jeżeli równe 0, zmniejszenie (o 1).

W: Jeżeli równe 1, zwiększenie lub zmniejszenie x; jeżeli równe 0, zwiększenie lub zmniejszenie y.

S: Jeżeli równe 1, zamiana x_i z y_i.

C: Jeżeli równe 1, uzupełnienie x_i i y_i.



Rysunek 14.4. *Układ logiczny służący do zwiększania (x, y) o jeden krok wzdłuż krzywej Hilberta*

Sygnały S i C wspólnie identyfikują „stan" z tabeli 14.7, gdzie wartości $(C, S) = (0, 0)$, $(0, 1)$, $(1, 0)$ i $(1, 1)$ oznaczają, odpowiednio, stany A, B, C i D. Sygnały wyjściowe to I_0 i W_0, które określają, czy należy wykonać operację zmniejszenia, czy zwiększenia wartości, oraz którą zmienną należy zmienić. Oprócz układu przedstawionego powyżej potrzebny jest układ zwiększania-zmniejszania wartości wraz z multiplekserem kierującym do tego układu wartość x lub y, jak również układ kierujący zmienioną wartość z powrotem do rejestru, który przechowuje x lub y. Alternatywnym rozwiązaniem byłoby wykorzystanie dwóch układów zmniejszania-zwiększania wartości.

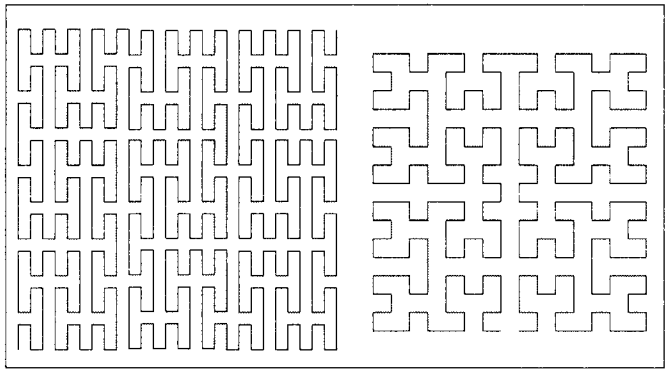
14.5. Nierekurencyjny algorytm generowania

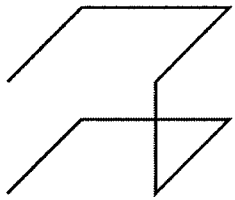
Algorytmy z tabel 14.2 i 14.7 oferują dwa nierekurencyjne algorytmy generowania krzywej Hilberta dowolnego stopnia. Każdy z algorytmów można zaimplementować sprzętowo bez większych trudności. Rozwiązanie sprzętowe oparte na informacjach z tabeli 14.2 wiąże się z użyciem rejestru przechowującego wartość s, która w każdym kroku jest zwiększana a następnie konwertowana do postaci współrzędnych (x, y). Rozwiązanie sprzętowe oparte na informacjach z tabeli 14.7 nie wiązałoby się z użyciem rejestru dla s, ale algorytm jest bardziej skomplikowany.

14.6. Inne krzywe wypełniające przestrzeń

Jak wcześniej wspomniano, w 1890 roku Peano jako pierwszy odkrył krzywą wypełniającą przestrzeń. Wiele odmian tej krzywej odkrytych od tego czasu często nazywa się krzywymi Peano. Jedną z interesujących odmian krzywej Hilberta odkrył w 1900 roku Eliakim Hastings Moore. Jest ona cykliczna w tym sensie, że punkt końcowy znajduje się o jeden krok od punktu początkowego. Krzywą Peano stopnia 3. oraz krzywą Moore'a stopnia 4. przedstawiono na rysunku 14.5. Krzywa Moore'a charakteryzuje się nieregularnością polegającą na tym, że krzywa stopnia 1. jest odwrócona do góry nogami (⊓↓), ale kształt ten nie pojawia się wśród krzywych wyższych stopni. Oprócz tego drobnego odstępstwa algorytmy związane z krzywą Moore'a są bardzo podobne do algorytmów związanych z krzywą Hilberta.

Krzywa Hilberta została uogólniona do postaci dowolnych prostokątów a także do trzech i większej liczby wymiarów. Podstawowy blok tworzący w przypadku 3-wymiarowej krzywej Hilberta przedstawiono poniżej. Przechodzi on przez wszystkie wierzchołki sześcianu o wymiarach 2×2×2. Te i wiele innych krzywych wypełniających przestrzeń omówiono w pracy [Sagan].





Rysunek 14.5.
Krzywa Peano (po lewej) i Moore'a (po prawej)

14.7. Zastosowania

Krzywe wypełniające przestrzeń znajdują zastosowanie w przetwarzaniu obrazów: kompresji, stosowaniu półtonów (rastrowaniu) i analizie strukturalnej [L&S]. Innym zastosowaniem jest zwiększenie wydajności działania komputera w przypadku metody śledzenia promieni (ang. *ray tracing*) — jednej z technik generowania obrazu. Tradycyjnie, określona scena jest skanowana rzutowanymi na nią promieniami w zwykłej kolejności linii skanujących (od strony lewej do prawej oraz od góry na dół). Kiedy promień dociera do obiektu w bazie danych symulowanej sceny, zostaje określony kolor oraz inne właściwości obiektu znajdującego się w tym punkcie, a otrzymane wartości wynikowe są używane do podświetlenia pikseli znajdujących się na drodze promienia. Taki opis stanowi oczywiście znaczne uproszczenie, ale dla naszych celów jest odpowiedni. Jednym z występujących tu problemów jest fakt, że baza danych jest często duża i dane dla każdego obiektu muszą być stronicowane i pobierane z powrotem w miarę jak promień skanujący dociera do różnych obiektów. Kiedy promień przechodzi przez linię, często dociera do wielu obiektów, do których dotarł już w poprzednim przebiegu skanowania, co wymaga wykonania ich ponownego stronicowania. Liczbę takich operacji można by zredukować, gdyby skanowanie posiadało pewną właściwość lokalności. Przykładowo, przydatnym działaniem mogłoby być pełne przeskanowanie ćwiartki ekranu przed przejściem do innej ćwiartki.

Krzywa Hilberta wydaje się posiadać właściwość lokalności, której szukamy. Skanuje ona całą ćwiartkę obszaru przed przejściem do kolejnej w sposób rekurencyjny i nie wykonuje długich skoków w momencie przechodzenia z jednej ćwiartki do drugiej.

Douglas Voorhies [Voor] przeprowadził symulację operacji stronicowania w przypadku konwencjonalnego jednokierunkowego przechodzenia linii skanującej, krzywej Peano oraz krzywej Hilberta. Jego metoda polegała na losowym rozrzuceniu na ekranie okręgów o danym rozmiarze. Ścieżka skanowania docierająca do okręgu reprezentowała dotarcie do nowego obiektu oraz jego stronicowanie. Kiedy jednak ścieżka opuszczała okręg, zakładał, że dane związane z obiektem pozostawały w pamięci do momentu opuszczenia okręgu o promieniu dwa razy większym od okręgu „obiektu". Zatem, jeżeli ścieżka skanująca opuszczała obiekt tylko na krótki okres a potem powracała do niego, zakładał, że nie miała miejsca operacja stronicowania. Swoje doświadczenie powtórzył dla wielu rozmiarów okręgów na ekranie o rozdzielczości 1024×1024.

Załóżmy, że dotarcie do obiektu i opuszczenie otaczającego go okręgu reprezentuje jedną operację stronicowania. Wówczas oczywiście zwykła linia skanująca powoduje D operacji stronicowania w przypadku niezbyt dużego okręgu o promieniu D pikseli, ponieważ każda linia skanująca wchodząca do niego wychodzi z zewnętrznego okręgu. Interesującym wynikiem symulacji Voorhiesa jest fakt, że w przypadku krzywej Peano liczba operacji stronicowania wymaganych w celu przeskanowania okręgu wynosi około 2,7 i, co zaskakujące, wartość ta jest niezależna od promienia okręgu. W przypadku krzywej Hilberta wartość ta wynosi około 1,4 i również jest niezależna od rozmiaru promienia okręgu. Zatem doświadczenie sugeruje, że krzywa Hilberta jest lepsza od krzywej Peano i o wiele lepsza od zwykłej ścieżki skanującej, jeśli chodzi o redukcję operacji stronicowania. Niezależność liczby operacji stronicowania od rozmiaru promienia okręgu wynika prawdopodobnie z faktu, że rozmiar okręgu zewnętrznego jest proporcjonalny do rozmiaru okręgu obiektu.

Rozdział 15.
Liczby zmiennopozycyjne

Bóg stworzył liczby całkowite,
wszystkie inne to dzieło człowieka.

Leopold Kronecker

Działanie na liczbach zmiennopozycyjnych przy użyciu arytmetyki całkowitoliczbowej oraz instrukcji logicznych stanowi często karkołomne zadanie, w szczególności, gdy chodzi o reguły i formaty określone w standardzie *IEEE Standard for Binary Floating--Point Arithmetic*, IEEE Std. 754-1985, znanym pod potoczną nazwą „arytmetyki IEEE". Zawiera on definicje wartości NaN (ang. *not a number*, nie liczba) oraz nieskończoności, które stanowią przypadki szczególne dla niemal wszystkich operacji. Zawiera plus oraz minus zero, które to wartości muszą być sobie równe. Definiuje czwarty wynik porównania — *nieuporządkowany* (ang. *unordered*). Najbardziej znaczący bit ułamka nie jest jawnie obecny w „normalnych" liczbach, ale jest tak w przypadku liczb „zdenormalizowanych" lub „podnormalnych". Ułamek znajduje się w *postaci ze znakiem* (ang. *signed-true form*) a wykładnik w *postaci przesuniętej* (ang. *biased form*), gdy liczby całkowite obecnie niemal zawsze są reprezentowane w postaci uzupełnienia dwójkowego. Oczywiście wszystkie wymienione elementy standardu mają swoje uzasadnienie, ale w efekcie powstające programy są pełne instrukcji *porównania* i *skoków*, co znacznie utrudnia ich wydajną implementację.

Poniżej założono, że Czytelnik zna w pewnym stopniu standard IEEE i przedstawiono tylko bardzo ogólnikowy opis.

15.1. Standard IEEE

Naszą uwagę skupimy na formatach o pojedynczej i podwójnej precyzji (32- oraz 64-bitowych), opisanych w standardzie IEEE 754. Standard opisuje również formaty „pojedynczy rozszerzony" oraz „podwójny rozszerzony", ale poniżej zostaną opisane tylko

skrótowo, gdyż związane z nimi szczegóły są zależne od implementacji (na przykład standard nie określa szerokości wykładnika). Format pojedynczej i podwójnej precyzji przedstawiono poniżej.

Format pojedynczej precyzji

s	e	f
1	8	23

Format podwójnej precyzji

s	e	f
1	11	52

Bit znaku s jest kodowany jako 0 dla wartości dodatnich i 1 dla wartości ujemnych. Przesunięty wykładnik e oraz ułamek f to wielkości o bitach bardziej znaczących po lewej stronie. Reprezentowaną wartość zmiennopozycyjną przedstawiono poniżej:

Format pojedynczej precyzji

e	f	*wartość*
0	0	± 0
0	$\neq 0$	$\pm 2^{-126}(0.f)$
od 1 do 254	–	$\pm 2^{e-127}(1.f)$
255	0	$\pm\infty$
255	$\neq 0$	NaN

Format podwójnej precyzji

e	f	*wartość*
0	0	± 0
0	$\neq 0$	$\pm 2^{-1022}(0.f)$
od 1 do 2046	–	$\pm 2^{e-1023}(1.f)$
2047	0	$\pm\infty$
2047	$\neq 0$	NaN

Jako przykład rozważmy sposób zakodowania liczby π w formacie o pojedynczej precyzji. W postaci binarnej [Knu1]

π = 11.0010 0100 0011 1111 0110 1010 1000 1000 1000 0101 1010 0011 0000 10...

Jest to zakres liczb „znormalizowanych", przedstawionych w trzecim wierszu powyższej tabeli. Najbardziej znacząca jedynka w π zostaje odrzucona, gdyż jedynka wiodąca nie jest przechowywana w przypadku kodowania liczb znormalizowanych. Wykładnik $e - 127$ powinien być równy 1, aby otrzymano punkt dziesiętny w odpowiednim miejscu, a stąd $e = 128$. Zatem reprezentacja ma postać

```
0 10000000 10010010001111111011011
```

lub w postaci szesnastkowej

```
40490FDB.
```

gdzie zaokrąglono ułamek do najbliższej reprezentowalnej wartości.

Liczby, w których $1 \leq e \leq 254$ są nazywane *liczbami znormalizowanymi* (ang. *normalized numbers*). Znajdują się w one w postaci normalnej w tym sensie, że ich najbardziej znaczący bit nie jest bezpośrednio przechowywany. Niezerowe liczby, w których $e = 0$ są nazywane *liczbami zdenormalizowanymi* (ang. *denormalized numbers*) lub po prostu *denormami*. Ich najbardziej znaczący bit jest przechowywany bezpośrednio. Taki schemat określa się czasem mianem *stopniowego niedomiaru* (ang. *gradual underflow*). Niektóre wartości graniczne w różnych zakresach liczb zmiennopozycyjnych przedstawiono w tabeli 15.1. W tabeli tej „Maksymalna l. całkowita" oznacza największą liczbę cał-

Tabela 15.1. *Wartości graniczne*

	Szesnastkowo	Wartość dokładna	Wartość przybliżona
Pojedyncza precyzja			
Najmniejszy denorm	0000 0001	2^{-149}	$1{,}401 * 10^{-45}$
Największy denorm	007F FFFF	$2^{-126}(1-2^{-23})$	$1{,}175 * 10^{-38}$
Najmniejsza l. znormalizowana	0080 0000	2^{-126}	$1{,}175 * 10^{-38}$
1,0	3F80 0000	1	1
Maksymalna l. całkowita	4B80 0000	2^{24}	$1{,}677 * 10^{7}$
Największa l. znormalizowana	7F7F FFFF	$2^{128}(1-2^{-24})$	$3{,}403 * 10^{38}$
∞	7F80 0000	∞	∞
Podwójna precyzja			
Najmniejszy denorm	0...0001	2^{-1074}	$4{,}941 * 10^{-324}$
Największy denorm	000F...F	$2^{-1022}(1-2^{-52})$	$2{,}225 * 10^{-308}$
Najmniejsza l. znormalizowana	0010...0	2^{-1022}	$2{,}225 * 10^{-308}$
1,0	3FF0...0	1	1
Maks. l. całkowita	4340...0	2^{53}	$9{,}007 * 10^{15}$
Największa l. znormalizowana	7FEF...F	$2^{1024}(1-2^{-53})$	$1{,}798 * 10^{308}$
∞	7FF0...0	∞	∞

kowitą, taką że wszystkie liczby całkowite mniejsze od niej lub jej równe, pod względem wartości bezwzględnej, można reprezentować dokładnie; kolejna liczba całkowita zostaje zaokrąglona.

W przypadku liczb znormalizowanych jedna *jednostka na ostatniej pozycji* („ulp", ang. *unit in the last position*) ma wartość względną w zakresie od $1 / 2^{24}$ do $1 / 2^{23}$ (od około $5{,}96 * 10^{-8}$ do $1{,}19 * 10^{-7}$) w formacie o pojedynczej precyzji oraz od $1 / 2^{53}$ do $1 / 2^{52}$ (od około $1{,}11 * 10^{-16}$ do $2{,}22 * 10^{-16}$) w formacie o podwójnej precyzji. Maksymalny „błąd względny", związany z zaokrągleniem do najbliższego modułu, to połowa określonych wartości.

Zakres liczb całkowitych, które są reprezentowane dokładnie, to od -2^{24} do $+2^{24}$ (–16 777 216 do +16 777 216) w przypadku formatu o pojedynczej precyzji i od -2^{53} do $+2^{53}$ (–9 007 199 254 740 992 do +9 007 199 254 740 992) w przypadku formatu o podwójnej precyzji. Oczywiście pewne liczby całkowite znajdujące się poza tymi przedziałami, na przykład duże potęgi liczby 2, mogą być reprezentowane dokładnie. Wymienione przedziały to przedziały maksymalne, w których *wszystkie* liczby całkowite są reprezentowane dokładnie.

Można by czasem zechcieć zamienić dzielenie przez stałą na mnożenie przez jej odwrotność. Można tego dokonać w przypadku pełnej (IEEE) dokładności jedynie w przypadku liczb, których odwrotności są reprezentowane dokładnie. Są to potęgi 2 od 2^{-127}

do 2^{127} w przypadku formatu o pojedynczej precyzji oraz potęgi 2 od 2^{-1023} do 2^{1023} w przypadku formatu o podwójnej precyzji. Liczby 2^{-127} i 2^{-1023} są liczbami zdenormalizowanymi, których używania należy unikać na maszynach implementujących operacje na liczbach zdenormalizowanych w sposób niewydajny.

15.2. Porównywanie liczb zmiennopozycyjnych za pomocą operacji całkowitoliczbowych

Jedną z cech kodowania IEEE jest to, że wartości różne od NaN są prawidłowo uporządkowane, jeśli traktować je jako liczby całkowite ze znakiem.

Zaprogramowanie porównywania liczb zmiennopozycyjnych za pomocą operacji całkowitoliczbowych wymaga zapobieżenia występowaniu wyniku „nieuporządkowanego". W przypadku standardu IEEE 754 wynik nieuporządkowany występuje wówczas, gdy jedna lub obie porównywane wartości to wartości NaN. Przedstawiona poniżej metoda traktuje wartości NaN jako większe od nieskończoności.

Porównania są również o wiele prostsze, gdy wartość –0,0 można traktować jako mniejszą od +0,0 (co stoi w sprzeczności ze standardem IEEE 754). Zakładając, że jest to akceptowalne, porównania można wykonywać w sposób przedstawiony poniżej, gdzie symbole $\overset{f}{<}$, $\overset{f}{\leq}$ oraz $\overset{f}{=}$ oznaczają porównania zmiennopozycyjne, zaś symbol ≈ jest używany jako forma przypomnienia, że przedstawione wzory nie traktują wartości ±0,0 do końca poprawnie.

$$a \overset{f}{=} b \approx (a = b)$$

$$a \overset{f}{<} b \approx (a \geq 0 \,\&\, a < b) \mid (a < 0 \,\&\, a \overset{u}{>} b)$$

$$a \overset{f}{\leq} b \approx (a \geq 0 \,\&\, a \leq b) \mid (a < 0 \,\&\, a \overset{u}{\geq} b)$$

Jeżeli wartość –0,0 musi być traktowana jako równa +0,0, wydaje się, że nie istnieje żaden elegancki sposób zapewnienia wykonania odpowiednich operacji, ale poniższe wzory, wynikające w mniej lub bardziej jasny sposób z powyższych, są jedną z możliwości.

$$\begin{aligned} a \overset{f}{=} b &\equiv (a = b) \mid (-a = a \,\&\, -b = b) \\ &\equiv (a = b) \mid ((a \mid b) = \mathbf{0x80000000}) \\ &\equiv (a = b) \mid (((a \mid b) \,\&\, \mathbf{0x7FFFFFFF}) = 0) \end{aligned}$$

$$a \overset{f}{<} b \equiv ((a \geq 0 \,\&\, a < b) \mid (a < 0 \,\&\, a \overset{u}{>} b)) \,\&\, ((a \mid b) \neq \mathbf{0x80000000})$$

$$a \overset{f}{\leq} b \equiv (a \geq 0 \,\&\, a \leq b) \mid (a < 0 \,\&\, a \overset{u}{\geq} b) \mid ((a \mid b) = \mathbf{0x80000000})$$

W przypadku pewnych zastosowań bardziej wydajnym rozwiązaniem może być najpierw przekształcenie liczb w pewien sposób a następnie wykonanie porównania zmiennopozycyjnego za pomocą pojedynczej instrukcji *porównania stałopozycyjnego*. Przykładowo w trakcie sortowania n liczb przekształcenie byłoby wykonywane tylko raz dla każdej liczby, zaś porównanie musiałoby być wykonane przynajmniej $\lceil n\log_2 n \rceil$ razy (w sensie zasady minimaks).

W tabeli 15.2 przedstawiono cztery takie przekształcenia. W lewej kolumnie wartość –0,0 jest przy porównaniu równa +0,0, zaś w prawej kolumnie wartość –0,0 jest przy porównaniu mniejsza od +0,0. We wszystkich przypadkach sens porównania nie zostaje zmieniony przez przekształcenie. Zmienna `n` jest wartością ze znakiem, `t` jest wartością bez znaku, zaś `c` może być wartością ze znakiem lub bez.

Tabela 15.2. *Wstępne przygotowanie liczb zmiennopozycyjnych do porównań całkowitoliczbowych*

–0,0 = +0,0 (IEEE)	**–0,0 < +0,0 (nie IEEE)**

`if (n >= 0) n = n+0x80000000;`
`else n = -n;`
Wykorzystanie porównania bez znaku.
	`if (n >= 0) n = n+0x80000000;`
	`else n = -n;`
	Wykorzystanie porównania bez znaku.

`c = 0x7FFFFFFF;`
`if (n < 0) n = (n ^ c) + 1;`
Wykorzystanie porównania ze znakiem.
	`c = 0x7FFFFFFF;`
	`if (n < 0) n = n ^ c;`
	Wykorzystanie porównania ze znakiem.

`c = 0x80000000;`
`if (n < 0) n = c - n;`
Wykorzystanie porównania ze znakiem.
	`c = 0x7FFFFFFF;`
	`if (n < 0) n = c - n;`
	Wykorzystanie porównania ze znakuem.

`t = n >> 31;`
`n = (n ^ (t >> 1)) -t;`
Wykorzystanie porównania ze znakiem.
	`t = (unsigned) (n >> 30) >> 1;`
	`n = n ^ t;`
	Wykorzystanie porównania ze znakiem.

Ostatni wiersz przedstawia kod pozbawiony rozgałęzień, który można zaimplementować na podstawowym procesorze RISC za pomocą czterech instrukcji w lewej kolumnie i trzech instrukcji w prawej kolumnie (owe cztery i trzy instrukcje muszą zostać wykonane dla każdej wartości porównywanej).

15.3. Rozkład cyfr wiodących

Kiedy firma IBM wprowadzała komputer System/360 w 1964 roku, specjaliści od analizy numerycznej byli przerażeni utratą precyzji związanej z arytmetyką precyzji pojedynczej. Wcześniejsza linia produktów IBM, rodzina 704 - 709 - 7090, posiadała słowo 36-bitowe. W przypadku arytmetyki zmiennopozycyjnej pojedynczej precyzji format określał 9-bitowe pole znaku i wykładnika oraz 27-bitowe pole ułamkowe. Najbardziej znaczący bit ułamka był bezpośrednio uwzględniany (w liczbach „normalnych"), tak więc wielkości były reprezentowane z precyzją 27 bitów.

Komputer S/360 posiadał słowo 32-bitowe. Dla arytmetyki pojedynczej precyzji IBM wybrał 8-bitowe pole znaku i wykładnika oraz 24-bitowe pole ułamka. Ów spadek z 27 do 24 bitów był wystarczająco złym posunięciem, ale to nie wszystko. W celu zachowania dużego przedziału wartości wykładnika jednostka w 7-bitowym wykładniku komputera S/360 reprezentuje współczynnik 16. Stąd ułamek jest o podstawie 16, a ten format został nazwany „heksadecymalnym" zmiennopozycyjnym. Cyfrą wiodącą może być dowolna z przedziału od 1 do 15 (binarnie — od 0001 do 1111). Liczby z wiodącą cyfrą 1 mają tylko 21 bitów precyzji (ze względu na 3 zera wiodące), ale powinny stanowić tylko 1/15 (6,7%) wszystkich liczb.

Okazuje się, że sytuacja wygląda jednak gorzej. Swego czasu wiele osób starało się zarówno analitycznie, jak i doświadczalnie dowieść, że cyfry wiodące nie mają rozkładu równomiernego. W przypadku heksadecymalnego formatu zmiennopozycyjnego około 25% liczb posiada jako wiodącą cyfrę 1 a stąd tylko 21 bitów precyzji.

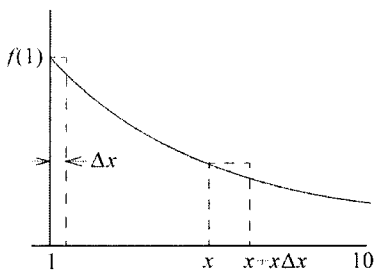
Rozważmy rozkład cyfr wiodących w formacie dziesiętnym. Załóżmy, że posiadamy duży zbiór liczb z jednostkami, takich jak długości, objętości, masy, prędkości i tak dalej, wyrażonych w notacji naukowej (np. $6{,}022 * 10^{23}$). Jeżeli cyfra wiodąca dużej ilości takich liczb posiada dobrze zdefiniowaną funkcję rozkładu, to musi być ona niezależna od jednostek — czy chodziłoby o cale lub centymetry, czy o funty lub kilogramy i tak dalej. Zatem, jeżeli pomnożymy wszystkie liczby w zbiorze przez stałą, rozkład cyfr wiodących powinien zostać niezmieniony. Przykładowo, rozważmy pomnożenie przez 2 — wnioskujemy, że ilość liczb z wiodącą cyfrą 1 (czyli należących do przedziału od 1.0 do 1,999... razy 10 do pewnej potęgi) musi być równa ilości liczb z wiodącą cyfrą 2 lub 3 (czyli należących do przedziału 2.0 do 3,999... razy 10 do pewnej potęgi), ponieważ nie powinno mieć znaczenia, czy daną jednostką długości jest cal lub pół cala albo daną jednostką masy jest kilogram lub pół kilograma i tak dalej.

Niech $f(x)$, gdzie $1 \le x < 10$, będzie funkcją gęstości prawdopodobieństwa dla cyfr wiodących zbioru liczb z jednostkami. $f(x)$ posiada tę właściwość, że

$$\int_a^b f(x)dx$$

jest proporcją liczb, które posiadają cyfry wiodące należące do przedziału od a do b. Odwołując się do poniższego wykresu, dla małego przyrostu Δx wartości x, f musi spełniać zależność

$$f(1) \cdot \Delta x = f(x) \cdot x\Delta x$$



ponieważ $f(1) \cdot \Delta x$ jest, w przybliżeniu, proporcją liczb należących do przedziału od 1 do $1 + \Delta x$ (ignorując wielokrotność potęgi 10) oraz $f(x) \cdot x\Delta x$ jest, w przybliżeniu, proporcją liczb należących do przedziału od x do $x + x\Delta x$. Ponieważ ten drugi zbiór to zbiór pierwszy pomnożony przez x, ich proporcje muszą być równe. Zatem funkcja gęstości prawdopodobieństwa jest zwykłą relacją odwrotności

$$f(x) = f(1) / x.$$

Obszar pod krzywą z zakresu od $x = 1$ do $x = 10$ musi być równy 1 (wszystkie liczby posiadają cyfry wiodące z zakresu 1,000... do 9,999...), zatem z łatwością można wykazać, że

$$f(1) = 1 / \ln 10.$$

Proporcja liczb z cyframi wiodącymi należącymi do przedziału od a do b, gdzie $1 \le a \le b < 10$, wynosi

$$\int_a^b \frac{dx}{x \ln 10} = \frac{\ln x}{\ln 10}\bigg|_a^b = \frac{\ln b / a}{\ln 10} = \log_{10} \frac{b}{a}.$$

Zatem w formacie dziesiętnym proporcja liczb z wiodącą cyfrą 1 wynosi $\log_{10}(2/1) \approx 0{,}30103$, zaś proporcja liczb z wiodącą cyfrą 9 wynosi $\log_{10}(10/9) \approx 0{,}0458$.

W przypadku systemu o podstawie 16 proporcja liczb z cyframi wiodącymi z zakresu od a do b, gdzie $1 \le a \le b < 16$, może zostać określona podobnie i wynosi $\log_{16}(b/a)$. Stąd proporcja liczb z wiodącą cyfrą 1 wynosi $\log_{16}(2/1) = 1 / \log_2 16 = 0{,}25$.

15.4. Tabela różnych wartości

W tabeli 15.3 przedstawiono reprezentację IEEE różnych wartości, które mogą okazać się interesujące. Wartości, które nie są dokładne, są zaokrąglane do najbliższej reprezentowalnej wartości.

Tabela 15.3. *Różne wartości*

Dziesiętnie	Format pojedynczej precyzji (heks.)	Format podwójnej precyzji (heks.)
$-\infty$	FF80 0000	FFF0 0000 0000 0000
−2,0	C000 0000	C000 0000 0000 0000
−1,0	BF80 0000	BFF0 0000 0000 0000
−0,5	BF00 0000	BFE0 0000 0000 0000
−0,0	8000 0000	8000 0000 0000 0000
+0,0	0000 0000	0000 0000 0000 0000
Najmniejszy dodatni denorm	0000 0001	0000 0000 0000 0001

Tabela 15.3. *Różne wartości (ciąg dalszy)*

Dziesiętnie	Format pojedynczej precyzji (heks.)	Format podwójnej precyzji (heks.)
Największy denorm	007F FFFF	000F FFFF FFFF FFFF
Najmniejszy dodatnia liczba znormalizowana	0080 0000	0010 0000 0000 0000
$\pi/180$ (0,01745...)	3C8E FA35	3F91 DF46 A252 9D39
0,1	3DCC CCCD	3FB9 9999 9999 999A
$\log_{10}2$ (0,3010...)	3E9A 209B	3FD3 4413 509F 79FF
$1/e$ (0,3678...)	3EBC 5AB2	3FD7 8B56 362C EE38
1/ln 10 (0,4342...)	3EDE 5BD9	3FDB CB7B 1526 E50E
0,5	3F00 0000	3FE0 0000 0000 0000
ln 2 (0,6931...)	3F31 7218	3FE6 2E42 FEFA 39EF
$1/\sqrt{2}$ (0,7071...)	3F35 04F3	3FE6 A09E 667F 3BCD
1/ln 3 (0,9102...)	3F69 0570	3FED 20AE 03BC C153
1,0	3F80 0000	3FF0 0000 0000 0000
ln 3 (1,0986...)	3F8C 9F54	3FF1 93EA 7AAD 030B
$\sqrt{2}$ (1,414...)	3FB5 04F3	3FF6 A09E 667F 3BCD
1/ln 2 (1,442...)	3FB8 AA3B	3FF7 1547 652B 82FE
$\sqrt{3}$ (1, 732...)	3FDD B3D7	3FFB B67A E858 4CAA
2,0	4000 0000	4000 0000 0000 0000
ln 10 (2,302...)	4013 5D8E	4002 6BB1 BBB5 5516
e (2,718...)	402D F854	4005 BF0A 8B14 5769
3,0	4040 0000	4008 0000 0000 0000
π (3,141...)	4049 0FDB	4009 21FB 5444 2D18
$\sqrt{10}$ (3,162...)	404A 62C2	4009 4C58 3ADA 5B53
$\log_2 10$ (3,321...)	4054 9A78	400A 934F 0979 A371
4,0	4080 0000	4010 0000 0000 0000
5,0	40A0 0000	4014 0000 0000 0000
6,0	40C0 0000	4018 0000 0000 0000
2π (6,283...)	40C9 0FDB	4019 21FB 5444 2D18
7,0	40E0 0000	401C 0000 0000 0000
8,0	4100 0000	4020 0000 0000 0000
9,0	4110 0000	4022 0000 0000 0000
10,0	4120 0000	4024 0000 0000 0000
11,0	4130 0000	4026 0000 0000 0000
12,0	4140 0000	4028 0000 0000 0000
13,0	4150 0000	402A 0000 0000 0000

Tabela 15.3. *Różne wartości (ciąg dalszy)*

Dziesiętnie	Format pojedynczej precyzji (heks.)	Format podwójnej precyzji (heks.)
14,0	4160 0000	402C 0000 0000 0000
15,0	4170 0000	402E 0000 0000 0000
16,0	4180 0000	4030 0000 0000 0000
180/π (57,295...)	4265 2EE1	404C A5DC 1A63 C1F8
$2^{23} - 1$	4AFF FFFE	415F FFFF C000 0000
2^{23}	4B00 0000	4160 0000 0000 0000
$2^{24} - 1$	4B7F FFFF	416F FFFF E000 0000
2^{24}	4B80 0000	4170 0000 0000 0000
$2^{31} - 1$	4F00 0000	41DF FFFF FFC0 0000
2^{31}	4F00 0000	41E0 0000 0000 0000
$2^{32} - 1$	4F80 0000	41EF FFFF FFE0 0000
2^{32}	4F80 0000	41F0 0000 0000 0000
2^{52}	5980 0000	4330 0000 0000 0000
2^{63}	5F00 0000	43E0 0000 0000 0000
2^{64}	5F80 0000	43F0 0000 0000 0000
Największa liczba znormalizowana	7F7F FFFF	7FEF FFFF FFFF FFFF
∞	7F80 0000	7FF0 0000 0000 0000
„Najmniejsza" wartość SNaN	7F80 0001	7FF0 0000 0000 0001
„Największa" wartość SNaN	7FBF FFFF	7FF7 FFFF FFFF FFFF
„Najmniejsza" wartość QNaN	7FC0 0000	7FF8 0000 0000 0000
„Największa" wartość QNaN	7FFF FFFF	7FFF FFFF FFFF FFFF

Standard IEEE 754 nie określa, jak należy odróżnić sygnalizujące (SNaN, signaling NaN) i ciche (QNaN, quiet NaN) wartości NaN. W tabeli 15.3 wykorzystano konwencję zastosowaną w procesorach PowerPC, AMD 29050, Intel x86 oraz I860, a także Fairchild Clipper: najbardziej znaczący bit ułamka ma wartość 0 w przypadku sygnalizującej oraz 1 w przypadku cichej wartości NaN. W komputerach Compaq Alpha, HP PA-RISC oraz MIPS wykorzystano ten sam bit, ale w odwrotny sposób (0 = cicha, 1 = sygnalizująca).

Rozdział 16.

Wzory na liczby pierwsze

16.1. Wprowadzenie

Jak wielu młodych studentów zafascynowałem się liczbami pierwszymi i próbowałem odnaleźć wzór, który umożliwiłby ich łatwe wyszukiwanie. Właściwie nie wiedziałem, jakie operacje miałyby być traktowane jako poprawne we wzorze, bądź, jakiej funkcji szukałem — wzór na n-tą liczbę pierwszą jako funkcję zmiennej n lub jako funkcję zależną od poprzedniej(nich) liczb(y) pierwszej(szych) czy też wzór, który dawałby liczby pierwsze ale nie wszystkie itd. Niemniej jednak, pomimo tych niejasności, chciałbym zaprezentować Czytelnikowi pewne rozważania o tym, co wiadomo o problemie. Jak się okaże, po pierwsze, *istnieją* wzory na liczby pierwsze, ale — po drugie — żaden z nich nie jest wystarczająco satysfakcjonujący.

W dużej mierze tematyka ta jest związana z zagadnieniami poruszanymi w niniejszej książce, przynajmniej o tyle, że stosowane wzory są podobne do niektórych używanych jako sztuczki programistyczne, aczkolwiek raczej w dziedzinie arytmetyki liczb rzeczywistych niż arytmetyki komputerowej. Najpierw jednak należy wspomnieć o kilku najważniejszych wydarzeniach z historii tematu.

W roku 1640 Fermat wysunął hipotezę, że wzór

$$F_n = 2^{2^n} + 1$$

zawsze daje w wyniku liczby pierwsze i liczby tej postaci otrzymały nazwę liczb Fermata. Co prawda F_n jest liczbą pierwszą dla n z przedziału od 0 do 4, jednak w 1732 Euler odkrył, że

$$F_5 = 2^{2^5} + 1 = 641 \cdot 6700417.$$

Spotkaliśmy się już z tymi współczynnikami w związku z dzieleniem przez stałą na maszynie 32-bitowej. Następnie, w 1880 roku, F. Landry wykazał, że

$$F_6 = 2^{2^6} + 1 = 274177 \cdot 67280421310721.$$

Wiadomo, że F_n jest liczbą złożoną dla wielu większych wartości n, na przykład wszystkich wartości n od 7 do 16 włącznie. Nie wiadomo o żadnej innej wartości $n > 4$, która byłaby liczbą pierwszą [H&W]. Tyle o pochopnych hipotezach[1].

Nawiasem mówiąc, można by się zastanowić, dlaczego Fermat użył podwójnego wykładnika. Wiedział, że jeżeli m posiada czynnik nieparzysty różny od 1, to liczba 2^{m+1} jest złożona. Wynika to faktu, że jeżeli $m = ab$ dla b nieparzystego i różnego od 1, to

$$2^{ab} + 1 = (2^a + 1)(2^{a(b-1)} - 2^{a(b-2)} + 2^{a(b-3)} - \ldots + 1).$$

Wiedząc to, musiał zastanawiać się nad wyrażeniem $2^m + 1$ z m, niezawierającym żadnych czynników nieparzystych (różnych od 1) — to znaczy $m = 2^n$. Wypróbował kilka wartości n i odkrył, że $2^{2^n} + 1$ wydaje się być liczbą pierwszą.

Oczywiście każdy zgodzi się, że wielomian kwalifikuje się jako „wzór". Jeden zadziwiający wielomian został odkryty przez Leonarda Eulera w 1772 roku. Odkrył on , że

$$f(n) = n^2 + n + 41$$

daje jako wartość liczbę pierwszą dla każdego n od 0 do 39. Jego wynik można rozszerzyć. Ze względu na fakt, że

$$f(-n) = n^2 - n + 41 = f(n-1),$$

$f(-n)$ daje jako wartość liczbę pierwszą dla każdego n od 1 do 40. Oznacza to, że $f(n)$ daje wartość będącą liczbą pierwszą dla każdego n z przedziału od -1 do -40. Stąd wielomian

$$f(n-40) = (n-40)^2 + (n-40) + 41 = n^2 - 79n + 1601$$

daje jako wartość liczbę pierwszą dla każdego n od 0 do 79. To, co można mu zarzucić, to brak estetycznego wyglądu, ponieważ wzór ten jest niemonotoniczny i powtarza wartości; to znaczy, dla $n = 0, 1, \ldots, 79$, $n^2 - 79n + 1601 = 1601, 1523, 1447, \ldots, 43, 41, 41, 43, \ldots, 1447, 1523, 1601$.

Pomimo tego sukcesu wiadomo obecnie, że nie istnieje żaden wielomian $f(n)$, który dawałby liczbę pierwszą dla każdego n (oprócz wielomianów stałych, takich jak $f(n) = 5$). W rzeczywistości każdy niebanalny „wielomian w wykładnikach" daje wartości liczb złożonych nieskończoną liczbę razy. Ściślej rzecz ujmując, zgodnie z twierdzeniem z pracy [H&W],

twierdzenie Jeżeli $f(n) = P(n, 2^n, 3^n, \ldots, k^n)$ jest wielomianem swoich argumentów, z całkowitymi współczynnikami, oraz $f(n) \to \infty$, kiedy $n \to \infty$, to $f(n)$ daje wartość złożoną dla nieskończenie wielu wartości n.

Zatem wzór w rodzaju $n^2 \cdot 2n + 2n^3 + 2n + 5$ musi zwracać nieskończoną liczbę wartości złożonych. Z drugiej strony, twierdzenie nie mówi nic o wzorach zawierających współczynniki, takie jak 2^{2^n}, n^n czy $n!$.

[1] Jakkolwiek jest to jedyna hipoteza Fermata, która okazała się błędna [Wells].

Wzór na n-tą liczbę pierwszą wyrażoną w funkcji n można otrzymać używając funkcji podłoga oraz liczby magicznej

$$a = 0{,}2030050007000110000 13\ldots$$

Liczba a stanowi, w zapisie dziesiętnym, pierwszą liczbę pierwszą zapisaną na pierwszym miejscu po przecinku dziesiętnym, drugą liczbą pierwszą zapisaną na kolejnych dwóch miejscach, trzecią liczbą pierwszą zapisaną na kolejnych trzech miejscach i tak dalej. Zawsze istnieje miejsce na n-tą liczbą pierwszą, ponieważ $p_n < 10^n$. Nie będziemy tego dowodzić, wskażemy jedynie, że wiadomo, iż zawsze istnieje liczba pierwsza między n a $2n$ (dla $n \geq 2$), a stąd z pewnością co najmniej jedna między n a $10n$, z czego z kolei wynika, że $p_n < 10^n$. Wzór na n-tą liczbę pierwszą ma postać

$$p_n = \left\lfloor 10^{\frac{n^2+n}{2}} a \right\rfloor - 10^n \left\lfloor 10^{\frac{n^2-n}{2}} a \right\rfloor,$$

gdzie skorzystaliśmy z relacji $1 + 2 + 3 + \ldots + n = (n^2 + n)/2$. Przykładowo

$$\begin{aligned} p_3 &= \lfloor 10^6 a \rfloor - 10^3 \lfloor 10^3 a \rfloor \\ &= 203005 - 203000 \\ &= 5. \end{aligned}$$

Jest to dość kosztowna sztuczka, gdyż wymaga znajomości rezultatu definiującego liczbę a. Wzór ten byłby interesujący, gdyby istniał sposób zdefiniowania a niezależnie od liczb pierwszych, jednak nikt nie zna takiej definicji.

Rzecz jasna, technika ta może zostać użyta w celu otrzymania wzoru na wiele ciągów, ale to jest spostrzeżenie oczywiste.

16.2. Wzory Willansa

C. P. Willans podał następujący wzór na n-tą liczbę pierwszą [Will]:

$$p_n = 1 + \sum_{m=1}^{2^n} \left\lfloor \sqrt[n]{n} \left(\sum_{x=1}^{m} \left\lfloor \cos^2 \pi \frac{(x-1)!+1}{x} \right\rfloor \right)^{-1/n} \right\rfloor.$$

Rozwikłanie wzoru należy rozpocząć się od twierdzenia Wilsona, które określa, że p jest liczbą pierwszą lub jest równe 1 wtedy i tylko wtedy, gdy $(p-1)! \equiv -1 (\mathrm{mod}\ p)$. Stąd

$$\frac{(x-1)!+1}{x}$$

jest liczbą całkowitą dla x będącego liczbą pierwszą lub $x = 1$ oraz jest ułamkiem dla wszystkich złożonych x. Zatem

$$F(x) = \left\lfloor \cos^2 \pi \frac{(x-1)!+1}{x} \right\rfloor = \begin{cases} 1, & \text{dla } x \text{ bedacego liczba pierwsza lub rownego 1} \\ 0, & \text{dla } x \text{ bedacego liczba zlozona} \end{cases} \quad (1)$$

Zatem, jeżeli $\pi(m)$ oznacza[2] liczbę liczb pierwszych $\leq m$, to

$$\pi(m) = -1 + \sum_{x=1}^{m} F(x) \quad (2)$$

Należy zauważyć, że $\pi(p_n) = n$, a ponadto

$$\pi(m) < n, \text{ dla } m < p_n$$

$$\pi(m) \geq n, \text{ dla } m \geq p_n.$$

Stąd liczba wartości m od 1 do ∞, dla których $\pi(m) < n$ wynosi $p_n - 1$. To znaczy

$$p_n = 1 + \sum_{m=1}^{\infty} (\pi(m) < n), \quad (3)$$

gdzie wyrażenie pod sumą jest predykatem (o wartości 0 lub 1).

Wzór na $\pi(m)$ jest znany, zatem równanie (3) stanowi wzór na n-tą liczbę pierwszą w funkcji n. Jednak posiada on dwie cechy, które mogą być potraktowane jako niemożliwe do zaakceptowania: sumowanie do nieskończoności oraz użycie predykatu, co nie jest przykładem standardowej operacji matematycznej.

Udowodniono, że dla $n \geq 1$ istnieje co najmniej jedna liczba pierwsza między n a $2n$. Zatem liczba liczb pierwszych $\leq 2^n$ wynosi co najmniej n — to znaczy, $\pi(2^n) \geq n$. Stąd predykat $\pi(m) < n$ wynosi 0 dla $m \geq 2^n$, tak więc jako górne ograniczenie sumowania można określić wartość 2^n.

Willans podał dość interesujące wyrażenie zastępcze dla predykatu. Niech

$$\mathrm{LT}(x, y) = \left\lfloor \sqrt[y]{\frac{y}{1+x}} \right\rfloor, \text{ dla } x = 0, 1, 2, \ldots; y = 1, 2, \ldots.$$

Wówczas, jeżeli $x < y$, to $1 \leq y / (1 + x) \leq y$, tak więc $1 \leq \sqrt[y]{y/(1+x)} \leq \sqrt[y]{y} < 2$. Ponadto, jeżeli $x \geq y$, to $0 < y / (1 + x) < 1$, tak więc $0 \leq \sqrt[y]{y/(1+x)} < 1$. Stosując funkcję podłoga, otrzymujemy

$$\mathrm{LT}(x, y) = \begin{cases} 1, & \text{dla } x < y, \\ 0, & \text{dla } x \geq y. \end{cases}$$

Oznacza to, że $\mathrm{LT}(x, y)$ jest predykatem $x < y$ (dla x i y należących do podanych przedziałów).

[2] Należałoby przeprosić za dwa różne użycia symbolu π w bliskim sąsiedztwie, ale jest to standardowa notacja i nie powinna powodować żadnych problemów.

Podstawiając, równanie (3) można zapisać jako

$$p_n = 1 + \sum_{m=1}^{2^n} \mathrm{LT}(\pi(m), n)$$
$$= 1 + \sum_{m=1}^{2^n} \left\lfloor \sqrt[n]{\frac{n}{1+\pi(m)}} \right\rfloor.$$

Dalsze podstawienie równania (2) za $\pi(m)$ w funkcji $F(x)$ oraz równania (1) za $F(x)$ daje wzór przedstawiony na początku niniejszego podrozdziału.

Drugi wzór

Willans podał też inny wzór:

$$p_n = \sum_{m=1}^{2^n} mF(m)\left\lfloor 2^{-|\pi(m)-n|} \right\rfloor$$

W tym przypadku F i π są funkcjami używanymi w jego pierwszym wzorze. Zatem $mF(m) = m$, jeżeli m jest liczbą pierwszą lub jest równe 1, a w przeciwnym razie wynosi 0. Trzeci współczynnik pod sumą to predykat $\pi(m) = n$. Składniki sumy mają wartość 0, poza jednym składnikiem, który jest n-tą liczbą pierwszą. Przykładowo,

$$p_4 = 1\cdot1\cdot0 + 2\cdot1\cdot0 + 3\cdot1\cdot0 + 4\cdot0\cdot0 + 5\cdot1\cdot0 + 6\cdot0\cdot0 + 7\cdot1\cdot1$$
$$+ 8\cdot0\cdot1 + 9\cdot0\cdot1 + 10\cdot0\cdot1 + 11\cdot1\cdot0 + \ldots + 16\cdot0\cdot0$$
$$= 7.$$

Trzeci wzór

Willans poszedł dalej, prezentując kolejny wzór na n-tą liczbę pierwszą, który nie zawiera żadnych „nieanalitycznych”[3] funkcji, takich jak podłoga i wartość bezwzględna. Rozpoczyna od zauważenia, że dla $x = 2, 3, \ldots$, funkcja

$$\frac{((x-1)!)^2}{x} = \begin{cases} \text{liczba calkowita} + \dfrac{1}{x}, \text{kiedy } x \text{ jest liczba pierwsza,} \\ \text{liczba calkowita, kiedy } x \text{ jest liczba zlozona lub rowne } 1. \end{cases}$$

Pierwsza część wynika ze wzoru

$$\frac{((x-1)!)^2}{x} = \frac{((x-1)!+1)\cdot((x-1)!-1)}{x} + \frac{1}{x},$$

a na podstawie twierdzenia Wilsona, x dzieli $(x-1)! + 1$. Stąd predykat „x jest liczbą pierwszą” dla $x \geq 2$ można określić jako

[3] Jest to terminologia Autora, nie Willansa.

$$H(x) = \frac{\sin^2 \pi \frac{((x-1)!)^2}{x}}{\sin^2 \frac{\pi}{x}}.$$

Stąd wynika, że

$$\pi(m) = \sum_{x=2}^{m} H(x), \text{ dla } m = 2,3,\ldots.$$

Nie można go przekształcić na wzór na p_n za pomocą metod użytych w pierwszych dwóch wzorach, ponieważ wykorzystują one funkcję *podłoga*. Zamiast tego Willans zasugerował następujący wzór[4] na predykat $x < y$, dla $x, y \geq 1$:

$$\mathrm{LT}(x, y) = \sin\left(\frac{\pi}{2} \cdot 2^e\right), \text{ gdzie}$$

$$e = \prod_{i=0}^{y-1} (x - i).$$

Zatem, jeżeli $x < y$, to $e = x(x-1)\ldots(0)(-1)\ldots(x-(y-1)) = 0$, tak więc $\mathrm{LT}(x, y) = \sin(\pi / 2) = 1$. Jeżeli $x \geq y$, iloczyn nie zawiera 0, tak więc $e \geq 1$, czyli $\mathrm{LT}(x, y) = \sin((\pi / 2) \cdot (\text{liczba parzysta})) = 0$.

W końcu, podobnie jak w pierwszym wzorze Willansa,

$$p_n = 2 + \sum_{m=2}^{2^n} \mathrm{LT}(\pi(m), n).$$

Pełny zapis wzoru trudno uznać za mało skomplikowany

$$p_n = 2 + \sum_{m=2}^{2^n} \sin\left(\frac{\pi}{2} \cdot 2^{\prod_{i=0}^{n-1}\left(\sum_{x=2}^{m} \frac{\sin^2 \pi \frac{((x-1)!)^2}{x}}{\sin^2 \frac{\pi}{x}} - i\right)}\right).$$

Czwarty wzór

Następnie Willans podał wzór na p_{n+1} w stosunku do p_n:

$$p_{n+1} = 1 + p_n + \sum_{i=1}^{2p_n} \prod_{j=1}^{i} f(p_n + j),$$

[4] Nieco upraszczamy jego wzór.

gdzie $f(x)$ jest predykatem „x jest liczbą złożoną" dla $x \geq 2$; to znaczy

$$f(x) = \left\lfloor \cos^2 \pi \frac{((x-1)!)^2}{x} \right\rfloor.$$

Alternatywnie można użyć zapisu $f(x) = 1 - H(x)$ w celu uniknięcia stosowania we wzorze funkcji *podłoga*.

Jako przykład użycia tego wzoru, niech $p_n = 7$. Wówczas

$$\begin{aligned} p_{n+1} &= 1 + 7 + f(8) + f(8)f(9) + f(8)f(9)f(10) \\ &\quad + f(8)f(9)f(10)f(11) + \ldots + f(8)f(9)\ldots f(14) \\ &= 1 + 7 + 1 + 1\cdot 1 + 1\cdot 1\cdot 1 + 1\cdot 1\cdot 1\cdot 0 + \ldots + 1\cdot 1\cdot 1\cdot 0\cdot 1\cdot 0\cdot 1 \\ &= 11. \end{aligned}$$

16.3. Wzory Wormella

C. P. Wormell [Wor] ulepszył wzory Willansa unikając zarówno funkcji trygonometrycznych, jak i funkcji podłoga. Ze wzoru Wormella można w ogólności obliczać za pomocą prostszego programu komputerowego, który korzysta tylko z arytmetyki całkowitoliczbowej. Nie korzystamy w tym przypadku z twierdzenia Wilsona. Wormell rozpoczął od zależności, dla $x \geq 2$,

$$B(x) = \prod_{a=2}^{x} \prod_{b=2}^{x} (x - ab)^2 = \begin{cases} \text{dodatnia liczba calkowita, jezeli x jest liczba pierwsza,} \\ 0, \text{jezeli x jest liczba zlozona.} \end{cases}$$

Zatem liczbę liczb pierwszych $\leq m$ określa wzór

$$\pi(m) = \sum_{x=2}^{m} \frac{1 + (-1)^{2^{B(x)}}}{2},$$

ponieważ sumowany jest predykat „x jest liczbą pierwszą".

Należy zauważyć, że dla $n \geq 1$ i $a \geq 0$

$$\prod_{r=1}^{x} (1 - r + a)^2 = \begin{cases} 0, \text{kiedy } a < n, \\ \text{dodatnia liczba calkowita, kiedy } a \geq n. \end{cases}$$

Powtarzając powyższą sztuczkę można wykazać, że predykat $a < n$ ma postać

$$(a < n) = \frac{1 - (-1)^{2^{\prod_{r=1}^{n} (1-r+a)^2}}}{2}.$$

Ponieważ

$$p_n = 2 + \sum_{m=2}^{2^n} (\pi(m) < n),$$

po wyciągnięciu przed nawias stałych współczynników otrzymujemy

$$p_n = \frac{3}{2} + 2^{n-1} - \frac{1}{2} \sum_{m=2}^{2^n} (-1)^{2^{\prod_{r=1}^{n} \left(1 - r + \frac{(m-1)}{2} + \frac{1}{2} \sum_{x=2}^{m} (-1)^{2^{\prod_{a=2}^{x} \prod_{b=2}^{x} (x-ab)^2}} \right)}}.$$

Zgodnie z obietnicą wzór Wormella nie zawiera funkcji trygonometrycznych. Jednakże, jak sam stwierdził, jeżeli potęgi –1 są rozszerzane za pomocą zależności $(-1)^n = \cos \pi n$, to znów się pojawiają.

16.4. Wzory na inne trudne funkcje

Przyjrzyjmy się bliżej dokonaniom Willansa i Wormella. Stwierdzamy, że poniższe reguły definiują klasę funkcji, które można reprezentować przez „wzory", które będziemy nazywać *funkcjami wzorowymi* (ang. *formula functions*). W takim przypadku symbol $\overline{x}$ stanowi skrótowy zapis dla $x_1, x_2, \ldots x_n$ dla dowolnego $n \geq 1$. Dziedziną wartości są liczby całkowite ... –2, –1, 0, 1, 2, ...

1. Stałe ... –1, 0, 1, ... są funkcjami wzorowymi.
2. Funkcje rzutowania (ang. *projection functions*) $f(\overline{x}) = x_i$, dla $1 \leq i \leq n$, są funkcjami wzorowymi.
3. Wyrażenia $x + y$, $x - y$ i xy są funkcjami wzorowymi, jeżeli są nimi x i y.
4. Klasa funkcji wzorowych jest zamknięta w ramach złożenia (podstawienia). To znaczy, $f(g_1(\overline{x}), g_2(\overline{x}), \ldots g_m(\overline{x}))$ jest funkcją wzorową, jeżeli są nią f i g_i, dla $i = 1, \ldots, m$.
5. Ograniczone sumy i iloczyny, zapisane jako

$$\sum_{i=a(\overline{x})}^{b(\overline{x})} f(i, \overline{x}) \qquad \prod_{i=a(\overline{x})}^{b(\overline{x})} f(i, \overline{x}),$$

są funkcjami wzorowymi, jeżeli są nimi a, b i f oraz $a(\overline{x}) \leq b(\overline{x})$.

Co do sum i iloczynów, wymagane jest, aby były one ograniczone w celu zachowania obliczeniowego charakteru wzorów. Oznacza to, że wzory można obliczać poprzez podstawianie wartości argumentów i wykonywanie skończonej liczby obliczeń. Powód używania symboli prim wobec $\sum$ i $\prod$ wyjaśniono w dalszej części niniejszego rozdziału.

Tworząc nowe funkcje wzorowe przy użyciu składania dodajemy nawiasy tam, gdzie to konieczne, zgodnie z przyjętymi konwencjami.

Należy zauważyć, że dzielenie nie zostało zawarte w powyższej liście: jest ono zbyt skomplikowane, aby je bezkrytycznie zaakceptować jako „funkcję wzorową". Jednak i tak powyższa lista nie jest minimalna. Ciekawym zadaniem mogłoby być poszukanie minimalnego punktu wyjścia, ale my nie będziemy zajmować się tym zagadnieniem.

Definicja „funkcji wzorowej" jest bliska definicji „funkcji elementarnej" podanej w pracy [Cut]. Jednakże dziedzina wartości użytych w pracy [Cut] to nieujemne liczby całkowite (jak to przyjęte w teorii funkcji rekurencyjnych). Ponadto w pracy znalazło się wymaganie, aby ograniczenia na iterowane sumy i iloczyny wynosiły 0 oraz $x - 1$ (gdzie x jest zmienną) i dozwolone jest, aby zakres był pusty (w tym przypadku suma jest definiowana jako 0, zaś iloczyn jako 1).

Poniżej wykażemy, że klasa funkcji wzorowych jest dość rozległa i należy do niej większość funkcji zwykle spotykanych w matematyce. Ale nie zawiera ona każdej funkcji, którą łatwo zdefiniować i która ma podstawowy charakter.

Nasze rozważania są nieco utrudnione w porównaniu z podobnymi rozważaniami w teorii funkcji rekurencyjnych, ponieważ tutaj zmienne mogą przyjmować wartości ujemne. Jednakże możliwości występowania wartości ujemnych często można uniknąć po prostu podnosząc do kwadratu pewne wyrażenie, które w przeciwnym razie wystąpiłoby w pierwszej potędze. Nasze wymaganie, aby sumy i iloczyny nie były puste jest kolejnym drobnym utrudnieniem.

U nas predykat oznacza po prostu funkcję o wartościach 0 lub 1, zaś w teorii funkcji rekurencyjnych predykat stanowi funkcję o wartościach *prawda-fałsz* i każdy predykat posiada związaną z nim *funkcję charakterystyczną* (ang. *characteristic function*) o wartościach 0 lub 1. Kojarzymy wartość 1 z prawdą, zaś wartość 0 z fałszem, jak to jest powszechnie przyjęte w językach programowania i informatyce (zgodnie z zachowaniem instrukcji *and* i *or*). W logice i teorii funkcji rekurencyjnych powiązanie to jest często odwrotne.

Poniżej wymieniono funkcje wzorowe.

1. $a^2 = aa$, $a^3 = aaa$ i tak dalej.

2. Predykat $a = b$:

$$(a = b) = \prod_{j=0}^{(a-b)^2} (1 - j).$$

3. $(a \neq b) = 1 - (a = b)$.

4. Predykat $a \geq b$:

$$(a \geq b) = \sum_{i=0}^{(a-b)^2} ((a-b) = i)$$
$$= \sum_{i=0}^{(a-b)^2} \prod_{j=0}^{((a-b)-i)^2} (1-j).$$

Teraz możemy wyjaśnić, dlaczego nie korzystamy z konwencji, że pusta suma ma wartość 0 a pusty iloczyn ma wartość 1. Jeżeli miałoby tak być, mielibyśmy do czynienia z takimi przypadkami, jak

$$(a = b) = \sum_{i=0}^{-(a-b)^2} 1 \qquad \text{oraz} \qquad (a \geq b) = \prod_{i=a}^{b-1} 0.$$

Porównywanie predykatów stanowi klucz do wszystkich innych działań i nie chcemy, aby bazowały one na czymkolwiek tak sztucznym.

1. $(a > b) = (a \geq b + 1)$.

2. $(a \leq b) = (b \geq a)$.

3. $(a < b) = (b > a)$.

4. $|a| = (2(a \geq 0) - 1)a$.

5. $\max(a, b) = (a \geq b)\,(a - b) + b$.

6. $\min(a, b) = (a \geq b)\,(b - a) + a$.

Teraz możemy poprawić iterowane sumy i iloczyny, tak aby dawały konwencjonalne i przydatne wyniki w przypadku, gdy zakresy są puste.

1. $$\sum_{i=a(\bar{x})}^{b(\bar{x})} f(i,\bar{x}) = (b(\bar{x}) \geq a(\bar{x})) \sum_{i=a(\bar{x})}^{\max(a(\bar{x}),b(\bar{x}))} f(i,\bar{x}).$$

2. $$\prod_{i=a(\bar{x})}^{b(\bar{x})} f(i,\bar{x}) = 1 + (b(\bar{x}) \geq a(\bar{x}))(-1 + \prod_{i=a(\bar{x})}^{\max(a(\bar{x}),b(\bar{x}))} f(i,\bar{x})).$$

Od tej pory będziemy używać symboli $\sum$ i $\prod$ bez znaków prim. Wszystkie funkcje dotąd zdefiniowane są zupełne (zdefiniowane dla wszystkich wartości argumentów).

3. $$n! = \prod_{i=1}^{n} i.$$

To daje $n! = 1$ dla $n \leq 0$.

Poniżej P i Q oznaczają predykaty.

1. $\neg P(\overline{x}) = 1 - P(\overline{x})$.

2. $P(\overline{x}) \,\&\, Q(\overline{x}) = P(\overline{x})Q(\overline{x})$.

3. $P(\overline{x}) \mid Q(\overline{x}) = 1 - (1 - P(\overline{x}))(1 - Q(\overline{x}))$.

4. $P(\overline{x}) \oplus Q(\overline{x}) = (P(\overline{x}) - Q(\overline{x}))^2$.

5. Jeżeli $P(\overline{x})$, to $f(\overline{y})$, w przeciwnym razie $g(\overline{z}) = P(\overline{x})f(\overline{y}) + (1 - P(\overline{x}))g(\overline{z})$.

6. a^n = jeżeli $n \geq 0$, to $\prod_{i=1}^{n} a$, w przeciwnym razie 0.

   To daje dowolnie i prawdopodobnie niepoprawnie w kilku przypadkach wyniki 0 dla $n < 0$, oraz wynik 1 dla 0^0.

7. $(m \leq \forall x \leq n)P(x, \overline{y}) = \prod_{x=m}^{n} P(x, \overline{y})$.

8. $(m \leq \exists x \leq n)P(x, \overline{y}) = 1 - \prod_{x=m}^{n} (1 - P(x, \overline{y}))$.

   W przypadku zakresu pustego $\forall$ jest prawdą, zaś $\exists$ jest fałszem.

9. $(m \leq \min x \leq n)P(x, \overline{y}) = m + \sum_{i=m}^{n} \prod_{j=m}^{i} (1 - P(x, \overline{y}))$.

   Wartość tego wyrażenia jest najmniejszym x w zakresie od m do n, takim że predykat jest prawdą lub wynosi m, jeśli przedział jest pusty, lub wynosi $n + 1$, jeżeli predykat jest fałszywy na (niepustym) przedziale. Operacja ta nosi nazwę *ograniczonej minimalizacji* (ang. *bounded minimalization*) i jest potężnym narzędziem służącym do opracowywania nowych funkcji wzorowych. Jest to rodzaj inwersji funkcjonalnej, co ilustruje kolejny wzór. Fakt, że taka minimalizacja może być wykonana za pomocą sumy iloczynów wynika z pracy Goodsteina [Good].

10. $\lfloor \sqrt{n} \rfloor = (0 \leq \min k \leq |n|)((k + 1)^2 > n)$.

    Jest to funkcja „całkowitoliczbowego pierwiastka kwadratowego", którą definiujemy jako zwracającą 0 dla $n < 0$, tak aby była funkcją zupełną.

11. $d \mid n = (-|n| \leq \exists q \leq |n|)(n = qd)$.

    Jest to predykat „d dzieli n", zgodnie z którym $0 \mid 0$ oprócz $\neg(0 \mid n)$ dla $n \neq 0$.

    $n \div d$ = jeżeli $n \geq 0$, to $(-n \leq \min q \leq n)(0 \leq \exists r \leq |d| - 1)(n = qd + r)$

    w przeciwnymrazie $(n \leq \min q \leq -n)(-|d| + 1 \leq \exists r \leq 0)(n = qd + r)$

    Jest to konwencjonalna obcinająca forma dzielenia całkowitoliczbowego. Dla $d = 0$ daje ona wynik $|n| + 1$, dowolnie.

12. $\mathrm{rem}(n, d) = n - (n \div d)d$.

Jest to konwencjonalna funkcja reszty. Jeżeli $\mathrm{rem}(n, d)$ ma wartość niezerową, posiada znak numeratora n. Jeżeli $d = 0$, reszta wynosi n.

13. $\mathrm{isprime}(n) = n \geq 2 \,\&\, \neg(2 \leq \exists d \leq |n| - 1)(d \mid n)$.

14. $\pi(n) = \sum_{i=1}^{n} \mathrm{isprime}(i)$.

Liczba liczb pierwszych $\leq n$.

15. $p_n = (1 \leq \min k \leq 2^n)(\pi(k) = n)$.

16. $\mathrm{exponent}(p, n) = (0 \leq \min x \leq |n|)\neg(p^{x+1} \mid n)$.

Jest to eksponent danego czynnika pierwszego p z n, dla $n \geq 1$.

17. Dla $n \geq 0$:

$$2^n = \prod_{i=1}^{n} 2, \qquad 2^{2^n} = \prod_{i=1}^{2^n} 2, \qquad 2^{2^{2^n}} = \prod_{i=1}^{2^{2^n}} 2 \text{ itd.}$$

18. n-ta cyfra po przecinku dziesiętnym w dziesiętnym rozwinięciu $\sqrt{2}$:

$$\mathrm{rem}(\lfloor \sqrt{2 \cdot 10^{2n}} \rfloor, 10).$$

A zatem klasa funkcji wzorowych jest dość obszerna. Jest jednak ograniczona następującym twierdzeniem (co najmniej):

twierdzenie

Jeżeli f jest funkcją wzorową, to istnieje stała k, taka że

$$f(\overline{x}) \leq 2^{2^{\cdot^{\cdot^{2^{\max(|x_1|,\ldots,|x_n|)}}}}}$$

gdzie występuje k dwójek.

Można je udowodnić, wykazując, że każde zastosowanie jednej z reguł od 1. do 5. (ze strony 306) zachowuje prawdziwość twierdzenia. Przykładowo, jeżeli $f(\overline{x}) = c$ (reguła 1.), to dla pewnego h

$$f(\overline{x}) \leq 2^{2^{\cdot^{\cdot^{2}}}}\}h,$$

gdzie występuje h dwójek. Stąd

$$f(\overline{x}) \leq 2^{2^{\cdot^{\cdot^{2^{\max(|x_1|,\ldots,|x_n|)}}}}}\Big\}h+2,$$

ponieważ $\max(|x_1|, \ldots, |x_n|) \geq 0$.

Dla $f(\overline{x}) = x_i$ (reguła 2.), $f(\overline{x}) \leq \max(|x_1|, \ldots, |x_n|)$, tak więc twierdzenie jest zachowane dla $k = 0$.

W przypadku reguły 3., niech

$$f(\overline{x}) \le 2^{2^{\cdot^{\cdot^{2^{\max(|x_1|,\ldots,|x_n|)}}}}}\Big\}k_1 \qquad \text{oraz} \qquad g(\overline{x}) \le 2^{2^{\cdot^{\cdot^{2^{\max(|x_1|,\ldots,|x_n|)}}}}}\Big\}k_2.$$

Wówczas oczywiście

$$\begin{aligned} f(\overline{x}) \pm g(\overline{x}) &\le 2 \cdot 2^{2^{\cdot^{\cdot^{2^{\max(|x_1|,\ldots,|x_n|)}}}}}\Big\}\max(k_1, k_2) \\ &\le 2^{2^{\cdot^{\cdot^{2^{\max(|x_1|,\ldots,|x_n|)}}}}}\Big\}\max(k_1, k_2) + 1. \end{aligned}$$

Podobnie można wykazać, że twierdzenie jest zachowane dla $f(x, y) = xy$.

Dowody, że reguły 4. oraz 5. zachowują prawdziwość twierdzenia są nieco nużące ale nietrudne, więc pominiemy je.

Z twierdzenia wynika, że funkcja

$$\mathrm{f}(\overline{\mathrm{x}}) = 2^{2^{\cdot^{\cdot^{2^{\mathrm{x}}}}}}\Big\}\mathrm{x} \tag{4}$$

nie jest funkcją wzorową, ponieważ dla dostatecznie dużego x równanie (4) przekracza wartość tego samego wyrażenia dla dowolnej stałej liczby k dwójek.

Czytelnik zainteresowany teorią funkcji rekurencyjnych może być zaciekawiony faktem, że równanie (4) jest *pierwotnie rekurencyjne* (ang. *primitive recursive*). Ponadto, łatwo wykazać bezpośrednio z definicji rekurencji pierwotnej, że funkcje wzorowe są pierwotnie rekurencyjne. Stąd klasa funkcji wzorowych stanowi podzbiór właściwy klasy funkcji pierwotnie rekurencyjnych. Zainteresowanego Czytelnika można odesłać do pozycji [Cut].

Podsumowując, w niniejszym podrozdziale wykazaliśmy, że wzór w postaci elementarnych funkcji istnieje nie tylko na n-tą liczbę pierwszą ale również dla wielu innych znanych funkcji występujących w matematyce. Ponadto nasze „funkcje wzorowe" nie są oparte na funkcjach trygonometrycznych, funkcji podłoga, wartości bezwzględnej, potęgach -1 ani dzieleniu bez reszty. Jedyny wybieg polega na wykorzystaniu faktu, że iloczyn wielu liczb wynosi 0, jeżeli dowolna z nich ma wartość 0, z czego skorzystano we wzorze na predykat $a = b$.

Jest jednak prawdą, że kiedy się je zobaczy, przestają być interesujące. Poszukiwania „interesujących" wzorów na liczby pierwsze należy kontynuować. Przykładowo w pracy [Rib] zacytowano zdumiewające twierdzenie W. H. Millsa (1947) o tym, że istnieje takie θ, że wyrażenie

$$\left\lfloor \theta^{3^n} \right\rfloor$$

daje wartość liczby pierwszej dla każdego $n \ge 1$. W rzeczywistości istnieje nieskończona liczba takich wartości (np. 1,3063778838+ oraz 1,4537508625483+). Ponadto nie ma nic specjalnego w wartości „3"; twierdzenie jest prawdziwe, jeżeli 3 zastąpi

się dowolną liczbą całkowitą ≥ 3 (dla różnych wartości θ). Co więcej, 3 można zastąpić 2, jeżeli jest prawdą, że między n^2 a $(n+1)^2$ zawsze występuje liczba pierwsza, co jest niemal pewne, ale nie zostało jak dotąd udowodnione. A poza tym..., cóż, zainteresowany Czytelnik powinien sięgnąć po pracę [Rib] oraz [Dud], gdzie znajdzie jeszcze bardziej fascynujące wzory tego typu.

Dodatek A

Tablice arytmetyczne dla maszyny 4-bitowej

W tabelach dodatku A podkreślenie oznacza przepełnienie ze znakiem.

Tabela A.1. *Dodawanie*

										-8	-7	-6	-5	-4	-3	-2	-1
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
	0	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
	1	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10
	2	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11
	3	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12
	4	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13
	5	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14
	6	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15
	7	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16
-8	8	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17
-7	9	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18
-6	A	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19
-5	B	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A
-4	C	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B
-3	D	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C
-2	E	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D
-1	F	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E

W tabeli dla odejmowania (tabela A.2) przyjęto, że bit przeniesienia dla operacji $\boldsymbol{a} - \boldsymbol{b}$ jest ustawiany w taki sam sposób, w jaki byłby ustawiany dla operacji $\boldsymbol{a} + \overline{\boldsymbol{b}} + \mathbf{1}$, tak więc przeniesienie jest równoważne „brakowi pożyczki".

Tabela A.2. *Odejmowanie (wiersz – kolumna)*

										-8	-7	-6	-5	-4	-3	-2	-1
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
	0	10	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1
	1	11	10	F	E	D	C	B	A	9	8	7	6	5	4	3	2
	2	12	11	10	F	E	D	C	B	A	9	8	7	6	5	4	3
	3	13	12	11	10	F	E	D	C	B	A	9	8	7	6	5	4
	4	14	13	12	11	10	F	E	D	C	B	A	9	8	7	6	5
	5	15	14	13	12	11	10	F	E	D	C	B	A	9	8	7	6
	6	16	15	14	13	12	11	10	F	E	D	C	B	A	9	8	7
	7	17	16	15	14	13	12	11	10	F	E	D	C	B	A	9	8
-8	8	18	17	16	15	14	13	12	11	10	F	E	D	C	B	A	9
-7	9	19	18	17	16	15	14	13	12	11	10	F	E	D	C	B	A
-6	A	1A	19	18	17	16	15	14	13	12	11	10	F	E	D	C	B
-5	B	1B	1A	19	18	17	16	15	14	13	12	11	10	F	E	D	C
-4	C	1C	1B	1A	19	18	17	16	15	14	13	12	11	10	F	E	D
-3	D	1D	1C	1B	1A	19	18	17	16	15	14	13	12	11	10	F	E
-2	E	1E	1D	1C	1B	1A	19	18	17	16	15	14	13	12	11	10	F
-1	F	1F	1E	1D	1C	1B	1A	19	18	17	16	15	14	13	12	11	10

W przypadku mnożenia (tabele A.3 i A.4) przepełnienie oznacza, że wynik nie może zostać wyrażony jako wartość 4-bitowa. W przypadku mnożenia ze znakiem (tabela A.3) jest to równoważne określeniu, że pięć pierwszych bitów 8-bitowego wyniku nie może wszystkie być równe 1 lub wszystkie równe 0.

Tabela A.3. *Mnożenie ze znakiem*

										-8	-7	-6	-5	-4	-3	-2	-1
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1	0	1	2	3	4	5	6	7	F8	F9	FA	FB	FC	FD	FE	FF
	2	0	2	4	6	8	A	C	E	F0	F2	F4	F6	F8	FA	FC	FE
	3	0	3	6	9	C	F	12	15	E8	EB	EE	F1	F4	F7	FA	FD
	4	0	4	8	C	10	14	18	1C	E0	E4	E8	EC	F0	F4	F8	FC
	5	0	5	A	F	14	19	1E	23	D8	DD	E2	E7	EC	F1	F6	FB
	6	0	6	C	12	18	1E	24	2A	D0	D6	DC	E2	E8	EE	F4	FA
	7	0	7	F	15	1C	23	2A	31	C8	CF	D6	DD	E4	EB	F2	F9
-8	8	0	F8	F0	E8	E0	D8	D0	C8	40	38	30	28	20	18	10	8
-7	9	0	F9	F2	EB	E4	DD	D6	CF	38	31	2A	23	1C	15	E	7
-6	A	0	FA	F4	EE	E8	E2	DC	D6	30	2A	24	1E	18	12	C	6
-5	B	0	FB	F6	F1	EC	E7	E2	DD	28	23	1E	19	14	F	A	5
-4	C	0	FC	F8	F4	F0	EC	E8	E4	20	1C	18	14	10	C	8	4
-3	D	0	FD	FA	F7	F4	F1	EE	EB	18	15	12	F	C	9	6	3
-2	E	0	FE	FC	FA	F8	F6	F4	F2	10	E	C	A	8	6	4	2
-1	F	0	FF	FE	FD	FC	FB	FA	F9	8	7	6	5	4	3	2	1

Tabela A.4. *Mnożenie bez znaku*

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
2	0	2	4	6	8	A	C	E	10	12	14	16	18	1A	1C	1E
3	0	3	6	9	C	F	12	15	18	1B	1E	21	24	27	2A	2D
4	0	4	8	C	10	14	18	1C	20	24	28	2C	30	34	38	3C
5	0	5	A	F	14	19	1E	23	28	2D	32	37	3C	41	46	4B
6	0	6	C	12	18	1E	24	2A	30	36	3C	42	48	4E	54	5A
7	0	7	E	15	1C	23	2A	31	38	3F	46	4D	54	5B	62	69
8	0	8	10	18	20	28	30	38	40	48	50	58	60	68	70	78
9	0	9	12	1B	24	2D	36	3F	48	51	5A	63	6C	75	7E	87
A	0	A	14	1E	28	32	3C	46	50	5A	64	6E	78	82	8C	96
B	0	B	16	21	2C	37	42	4D	58	63	6E	79	84	8F	9A	A5
C	0	C	18	24	30	3C	48	54	60	6C	78	84	90	9C	A8	B4
D	0	D	1A	27	34	41	4E	5B	68	75	82	8F	9C	A9	B6	C3
E	0	E	1C	2A	38	46	54	62	70	7E	8C	9A	A8	B6	C4	D2
F	0	F	1E	2D	3C	4B	5A	69	78	87	96	A5	B4	C3	D2	E1

Tabele A.5 i A.6 dotyczą zwykłego dzielenia z obcinaniem. Tabela A.5 zawiera wynik wynoszący 8 z przepełnieniem w przypadku maksymalnej liczby ujemnej dzielonej przez –1, ale na większości maszyn wynik jest w takim przypadku nieokreślony lub operacja jest anulowana.

Tabela A.5. *Dzielenie krótkie ze znakiem (wiersz / kolumna)*

										-8	-7	-6	-5	-4	-3	-2	-1
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
	0	-	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1	-	1	0	0	0	0	0	0	0	0	0	0	0	0	0	F
	2	-	2	1	0	0	0	0	0	0	0	0	0	0	0	F	E
	3	-	3	1	1	0	0	0	0	0	0	0	0	0	F	F	D
	4	-	4	2	1	1	0	0	0	0	0	0	0	F	F	E	C
	5	-	5	2	1	1	1	0	0	0	0	0	0	F	F	E	B
	6	-	6	3	2	1	1	1	0	0	0	F	F	F	E	D	A
	7	-	7	3	2	1	1	1	1	0	F	F	F	F	E	D	9
-8	8	-	8	C	E	E	F	F	F	1	1	1	1	2	2	4	8
-7	9	-	9	D	E	F	F	F	F	0	1	1	1	1	2	3	7
-6	A	-	A	D	E	F	F	F	0	0	0	1	1	1	2	3	6
-5	B	-	B	E	F	F	F	0	0	0	0	0	1	1	1	2	5
-4	C	-	C	E	F	F	0	0	0	0	0	0	0	1	1	2	4
-3	D	-	D	F	F	0	0	0	0	0	0	0	0	0	1	1	3
-2	E	-	E	F	0	0	0	0	0	0	0	0	0	0	0	1	2
-1	F	-	F	0	0	0	0	0	0	0	0	0	0	0	0	0	1

Tabela A.6. *Dzielenie krótkie bez znaku (wiersz / kolumna)*

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	-	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	-	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	-	2	1	0	0	0	0	0	0	0	0	0	0	0	0	0
3	-	3	1	1	0	0	0	0	0	0	0	0	0	0	0	0
4	-	4	2	1	1	0	0	0	0	0	0	0	0	0	0	0
5	-	5	2	1	1	1	0	0	0	0	0	0	0	0	0	0
6	-	6	3	2	1	1	1	0	0	0	0	0	0	0	0	0
7	-	7	3	2	1	1	1	1	0	0	0	0	0	0	0	0
8	-	8	4	2	2	1	1	1	1	0	0	0	0	0	0	0
9	-	9	4	3	2	1	1	1	1	1	0	0	0	0	0	0
A	-	A	5	3	2	2	1	1	1	1	1	0	0	0	0	0
B	-	B	5	3	2	2	1	1	1	1	1	1	0	0	0	0
C	-	C	6	4	3	2	2	1	1	1	1	1	1	0	0	0
D	-	D	6	4	3	2	2	1	1	1	1	1	1	1	0	0
E	-	E	7	4	3	2	2	2	1	1	1	1	1	1	1	0
F	-	F	7	5	3	3	2	2	1	1	1	1	1	1	1	1

Tabele A.7 i A.8 dotyczą zwykłego dzielenia z obcinaniem. Tabela A.7 zawiera wynik wynoszący 0 w przypadku maksymalnej liczby ujemnej dzielonej przez –1, ale na większości maszyn wynik jest w takim przypadku nieokreślony lub operacja jest anulowana.

Tabela A.7. *Reszta z dzielenia krótkiego ze znakiem (wiersz / kolumna)*

										-8	-7	-6	-5	-4	-3	-2	-1
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
	0	-	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1	-	0	1	1	1	1	1	1	1	1	1	1	1	1	1	0
	2	-	0	0	2	2	2	2	2	2	2	2	2	2	2	0	0
	3	-	0	1	0	3	3	3	3	3	3	3	3	3	0	1	0
	4	-	0	0	1	0	4	4	4	4	4	4	4	0	1	0	0
	5	-	0	1	2	1	0	5	5	5	5	5	0	1	2	1	0
	6	-	0	0	0	2	1	0	6	6	6	0	1	2	0	0	0
	7	-	0	1	1	3	2	1	0	7	0	1	2	3	1	1	0
-8	8	-	0	0	E	0	D	E	F	0	F	E	D	0	E	0	0
-7	9	-	0	F	F	D	E	F	0	9	0	F	E	D	F	F	0
-6	A	-	0	0	0	E	F	0	A	A	A	0	F	E	0	0	0
-5	B	-	0	F	E	F	0	B	B	B	B	B	0	F	E	F	0
-4	C	-	0	0	F	0	C	C	C	C	C	C	C	0	F	0	0
-3	D	-	0	F	0	D	D	D	D	D	D	D	D	D	0	F	0
-2	E	-	0	0	E	E	E	E	E	E	E	E	E	E	E	0	0
-1	F	-	0	F	F	F	F	F	F	F	F	F	F	F	F	F	0

Tabela A.8. *Reszta z dzielenia krótkiego bez znaku (wiersz / kolumna)*

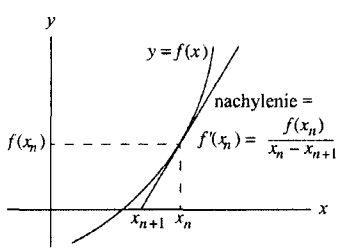
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	-	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	-	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	-	0	0	2	2	2	2	2	2	2	2	2	2	2	2	2
3	-	0	1	0	3	3	3	3	3	3	3	3	3	3	3	3
4	-	0	0	1	0	4	4	4	4	4	4	4	4	4	4	4
5	-	0	1	2	1	0	5	5	5	5	5	5	5	5	5	5
6	-	0	0	0	2	1	0	6	6	6	6	6	6	6	6	6
7	-	0	1	1	3	2	1	0	7	7	7	7	7	7	7	7
8	-	0	0	2	0	3	2	1	0	8	8	8	8	8	8	8
9	-	0	1	0	1	4	3	2	1	0	9	9	9	9	9	9
A	-	0	0	1	2	0	4	3	2	1	0	A	A	A	A	A
B	-	0	1	2	3	1	5	4	3	2	1	0	B	B	B	B
C	-	0	0	0	0	2	0	5	4	3	2	1	0	C	C	C
D	-	0	1	1	1	3	1	6	5	4	3	2	1	0	D	D
E	-	0	0	2	2	4	2	0	6	5	4	3	2	1	0	E
F	-	0	1	0	3	0	3	1	7	6	5	4	3	2	1	0

Dodatek B
Metoda Newtona

W celu skrótowego przypomnienia metody Newtona załóżmy że posiadamy różniczkowalną funkcję f zmiennej rzeczywistej x i chcemy rozwiązać równanie $f(x) = 0$ względem x. Jeżeli posiadamy bieżące oszacowanie x_n pierwiastka funkcji f, metoda Newtona pozwala na otrzymanie lepszego oszacowania x_{n+1} przy sprzyjających okolicznościach, zgodnie ze wzorem:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

Wyrażenie $f'(x_n)$ oznacza pochodną z f w punkcie $x = x_n$. Rozwiązanie przedstawionego wzoru można odczytać z poniższego rysunku (rozwiązanie dla x_{n+1}).



Metoda ta sprawdza się bardzo dobrze w przypadku prostych, charakteryzujących się w miarę jednostajnym przebiegiem funkcji, takich jak wielomiany, zakładając, że pierwsze oszacowanie jest dość bliskie rzeczywistej wartości. Kiedy oszacowanie jest dostatecznie bliskie, metoda ulega konwergencji kwadratowej. Oznacza to, że jeżeli r jest dokładną wartością pierwiastka funkcji a x_n jest dostatecznie bliskimi oszacowaniem, to

$$|x_{n+1} - r| \le (x_n - r)^2.$$

Zatem liczba cyfr znaczących podwaja się po każdej iteracji (na przykład, jeżeli $|x_n - r| \le 0{,}001$, to $|x_{n+1} - r| \le 0{,}000001$).

Jeżeli pierwsze oszacowanie jest bardzo odległe, to kolejne iteracje mogą podlegać bardzo powolnej konwergencji, mogą rozbiegać się do nieskończoności, mogą ulegać

konwergencji do pierwiastka innego niż najbliższy do początkowo wybranego oszacowania lub mogą ulec zapętleniu w nieskończoność wśród określonych wartości.

Dyskusja powyższa miała nieduże znaczenie teoretyczne, z uwagi na użycie określeń, takich jak „sprzyjające okoliczności", „w miarę jednostajne" czy „dostatecznie bliskie". W celu zapoznania się bardziej precyzyjnym omówieniem powyższego zagadnienia należy sięgnąć po niemal dowolny podręcznik z pierwszego roku studiów matematycznych.

Pomimo pułapek kryjących się w tej metodzie, jest ona niekiedy przydatna w dziedzinie liczb całkowitych. Aby sprawdzić, czy metoda nadaje się do wykorzystania w przypadku określonej funkcji, należy ją wypróbować, podobnie jak uczyniono to w podrozdziale 11.1 *Całkowitoliczbowy pierwiastek kwadratowy* na stronie 231.

W tabeli B.1 przedstawiono kilka wzorów iteracyjnych otrzymanych na podstawie metody Newtona w celu obliczania określonych liczb. Pierwsza kolumna zawiera liczbę, którą należy obliczyć. W drugiej podano funkcję, która posiada tę liczbę jako pierwiastek. W trzeciej kolumnie przedstawiono prawą stronę wzoru Newtona, odpowiadającego danej funkcji.

Tabela B.1. *Metoda Newtona służąca do obliczania określonych wartości*

Obliczana wielkość	Funkcja	Wzór iteracyjny
$\sqrt{a}$	$x^2 - a$	$\frac{1}{2}\left(x_n + \frac{a}{x_n}\right)$
$\sqrt[3]{a}$	$x^3 - a$	$\frac{1}{3}\left(2x_n + \frac{a}{x_n^2}\right)$
$\frac{1}{\sqrt{a}}$	$x^{-2} - a$	$\frac{x_n}{2}\left(3 - ax_n^2\right)$
$\frac{1}{a}$	$x^{-1} - a$	$x_n(2 - ax_n)$
$\log_2 a$	$2^x - a$	$x_n + \frac{1}{\ln 2}\left(\frac{a}{2^{x_n}} - 1\right)$

Nie zawsze jest rzeczą łatwą znalezienie odpowiedniej funkcji. Oczywiście istnieje wiele funkcji, które posiadają daną wielkość jako pierwiastek i tylko kilka z nich pozwala na otrzymanie przydatnego wzoru iteracyjnego. Zazwyczaj używana funkcja stanowi swego rodzaju odwrotność poszukiwanej wartości. Przykładowo, w celu znalezienia $\sqrt{a}$ używamy funkcji $f(x) = x^2 - a$; w celu znalezienia $\log_2 a$ używamy funkcji $f(x) = 2^x - a$ i tak dalej[1].

Wzór iteracyjny na wartość $\log_2 a$ ulega konwergencji (do $\log_2 a$), nawet jeżeli mnożnik 1 / ln2 zostanie nieco zmieniony (na przykład na 1 lub 2). Jednakże wówczas konwergencja odbywa się wolniej. Wartość 3/2 lub 23/16 może być przydatna w niektórych zastosowaniach (1 / ln2 ≈ 1,4427).

[1] Metoda Newtona dla szczególnego przypadku funkcji pierwiastka kwadratowego była znana Babilończykom około 4000 lat temu.

Bibliografia

[AES] *Advanced Encryption Standard* (*AES*), National Institute of Standards and Technology, FIPS PUB 197 (November 2001). Materiały dostępne pod adresem *http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf*.

[Alv] Alverson, Robert, „Integer Division Using Reciprocals", In Proceedings IEEE 10th Symposium on Computer Arithmetic, June 26 – 28, 1991, Grenoble, France, 186 – 190.

[Aus1] Znalezione w podprogramie interpretera języka REXX napisanego przez Marca A. Auslandera.

[Aus2] Auslander, Marc A. Korespondencja prywatna.

[Bern] Bernstein, Robert, „Multiplication by Integer Constants", *Software — Practice and Experience 16*, 7 (July 1986), 641 – 652.

[BGN] Burks, Arthur W., Goldstine, Herman H., and von Neumann, John, „Preliminary Discussion of the Logical Design of an Electronic Computing Instrument, Second Edition" (1947), In *Papers of John von Neumann on Computing and Computing Theory*, Volume 12 in the Charles Babbage Institute Reprint Series for the History of Computing, MIT Press, 1987.

[CJS] Stephenson, Christopher J., Korespondencja prywatna.

[Cohen] Reguły te wskazał Norman H. Cohen.

[Cut] Cutland, Nigel J., *Computability: An Introduction to Recursive Function Theory*, Cambridge University Press, 1980.

[CWG] Hoxey, Karim, Hay, and Warren (Editors), *The PowerPC Compiler Writer's Guide*, Warthman Associates, 1996.

[DES] *Data Encryption Standard* (*DES*), National Institute of Standards and Technology, FIPS PUB 46 – 2 (December 1993). Materiały dostępne pod adresem *http://www.itl.nist.gov/fipspubs/fip46-2.htm*.

[Dewd] Dewdney, A. K., *The Turing Omnibus*, Computer Science Press, 1989.

[Dud] Dudley, Underwood, „History of a Formula for Primes", *American Mathematics Monthly 76* (1969), 23 – 28.

[EL] Ercegovac, Miloš D. and Lang, Tomás, *Division and Square Root: Digit-Recurrence Algorithms and Implementations*, Kluwer Academic Publishers, 1994.

[Gard] Gardner, Martin, „Mathematical Games" artykuł w *Scientific American 227*, 2 (August 1972), 106 – 109.

[GGS] Gregoire, Dennis G., Groves, Randall D., and Schmookler, Martin S., *Single Cycle Merge/Logic Unit*, US Patent No. 4,903,228, February 20, 1990.

[GK] Granlund, Torbjörn and Kenner, Richard, „Eliminating Branches Using a Superoptimizer and the GNU C Compiler", In *Proceedings of the 5th ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI), July 1992, 341 – 352.

[GKP] Graham, Ronald L., Knuth, Donald E., Patashnik, Oren, *Matematyka konkretna*, PWN, Warszawa 1998.

[GLS1] Steele, Guy L., Jr., Korespondencja prywatna.

[GLS2] Steele, Guy L., Jr., „Arithmetic Shifting Considered Harmful", AI Memo 378, MIT Artificial Intelligence Laboratory (September 1976); także w *SIGPLAN Notices 12*, 11 (November 1977), 61 – 69.

[GM] Granlund, Torbjörn and Montgomery, Peter L., „Division by Invariant Integers Using Multiplication", In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation* (PLDI), August 1994, 61 – 72.

[Gold] Drugie wyrażenie jest dziełem Richarda Goldberga.

[Good] Goodstein, Prof. R. L., „Formulae for Primes", *The Mathematical Gazette 51* (1967), 35 – 36.

[GSO] Znalezione przez GNU Superoptimizer.

[HAK] Beeler, M., Gosper, R. W., and Schroeppel, R., *HAKMEM*, MIT Artificial Intelligence Laboratory AIM 239, February 1972.

[Hay1] Hay, R. W. Korespondencja prywatna.

[Hay2] Pierwsze wyrażenie znaleziono w podprogramie kompilatora napisanego przez R. W. Haya.

[Hil] Hubert, David, „Ueber die stetige Abbildung einer Linie auf ein Flächenstück", *Mathematischen Annalen 38* (1891), 459 – 460.

[Hop] Hopkins, Martin E., Korespondencja prywatna.

[HS] Hillis, W. Daniel and Steele, Guy L., Jr., „Data Parallel Algorithms", *Comm. ACM 29*, 12 (December 1986) 1170 – 1183.

[H&P] Hennessy, John L. and Patterson, David A. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1990.

[H&S] Harbison, Samuel P and Steele, Guy L., Jr., *C: A Reference Manual*, Fourth Edition. Prentice-Hall, 1995.

[H&W] Hardy, G. H. and Wright, E. M., *An Introduction to the Theory of Numbers*, Fourth Edition. Oxford University Press, 1960.

[IBM] Z kursu programowania firmy IBM, 1961.

[Irvine] Irvine, M. M., „Early Digital Computers at Bell Telephone Laboratories", *IEEE Annals of the History of Computing 23*, 3 (July-September 2001), 22 – 42.

[JVN] von Neumann, John, „First Draft of a Report on the EDVAC", In *Papers of John von Neumann on Computing and Computing Theory*, Volume 12 in the Charles Babbage Institute Reprint Series for the History of Computing, MIT Press, 1987.

[Ken] Znalezione w kompilatorze GNU języka C dla komputera IBM RS/6000, który został przeniesiony przez Richarda Kennera. Przypisuje to artykułowi z konferencji PLDI z 1992 roku autorstwa swojego oraz Torbjörna Granlunda.

[Knu1] Knuth, Donald E., *Sztuka programowania. Tom 1. Algorytmy podstawowe*, WNT, Warszawa 2002.

[Knu2] Knuth, Donald E., *Sztuka programowania. Tom 2. Algorytmy seminumeryczne*, WNT, Warszawa 2002.

[Knu3] Pomysł użycia ujemnej liczby całkowitej jako podstawy systemu liczbowego w arytmetyce został niezależnie odkryty przez wiele osób. Najwcześniejsze wzmianki, o jakich mówi Knuth dotyczą Vittorio Grünwalda w 1885. Sam Knuth wysłał artykuł na ten temat w 1955 r. na konkurs „poszukiwania talentów naukowych" dla absolwentów studiów wyższych. Inne odwołania do wczesnych źródeł można znaleźć w pozycji Knutha, Tom 2.

[KRS] Kruskal, Clyde P., Rudolph, Larry, and Snir, Marc, „The Power of Parallel Prefix", *IEEE Transactions on Computers C-34*, 10 (October 1985), 965 – 968.

[Lamp] Lamport, Leslie, „Multiple Byte Processing with Full-Word Instructions", *Communications of the ACM 18*, 8 (August 1975), 471 – 475.

[LSY] Lee, Ruby B., Shi, Zhijie, and Yang, Xiao, „Efficient Permutation Instructions for Fast Software Cryptography", *IEEE Micro 21*, 6 (November/December 2001), 56 – 69.

[L&S] Lam, Warren M. and Shapiro, Jerome M., „A Class of Fast Algorithms for the Peano-Hilbert Space-Filling Curve", In *Proceedings ICIP 94*, 1 (1994), 638 – 641.

[MD] Denneau, Monty, Korespondencja prywatna.

[MIPS] Kane, Gerry and Heinrich, Joe, *MIPS RISC Architecture*, Prentice-Hall, 1992.

[MM] Morton, Mike, „Quibbles & Bits", *Computer Language 7*, 12 (December 1990), 45 – 55.

[MMIX] Część mającego wkrótce się ukazać wydania *Sztuki programowania*. Dostępna pod adresem *http://www-cs-faculty.stanford.edu/~knuth/taocp.html*.

[NZM] Niven, Ivan, Zuckerman, Herbert S., and Montgomery, Hugh L., *An Introduction to the Theory of Numbers*, Fifth Edition. John Wiley & Sons, Inc., 1991.

[PB] Purdom, Paul Walton Jr., and Brown, Cynthia A., *The Analysis of Algorithms*, Holt, Rinehart and Winston, 1985.

[PHO] Oden, Peter H., Korespondencja prywatna.

[PL8] Sztuczki tej Autor nauczył się dzięki kompilatorowi PL.8.

[Rib] Ribenboim, Paulo, *Mała księga wielkich liczb pierwszych*, WNT, Warszawa 1997.

[RND] Reingold, Edward M., Nievergelt, Jurg, Deo, Narsingh, *Algorytmy kombinatoryczne*, PWN, Warszawa 1985.

[Sagan] Sagan, Hans, *Space-Filling Curves*, Springer-Verlag, 1994, Doskonała książka, gorąco polecana każdemu, choć trochę zainteresowanemu tematem.

[Shep] Shepherd, Arvin D., Korespondencja prywatna.

[Stall] Stallman, Richard M., *Using and Porting GNU CC*, Free Software Foundation, 1998.

[Voor] Voorhies, Douglas, „Space-Filling Curves and a Measure of Coherence", *Graphics Gems II*, AP Professional (1991).

[War] Warren, H. S., Jr., „Functions Realizable with Word-Parallel Logical and Two's-Complement Addition Instructions", *Communications of the ACM 20*, 6 (June 1977), 439 – 441.

[Weg] Najwcześniejsze wzmianki, o jakich wiadomo Autorowi, to: Wegner, P. A., „A Technique for Counting Ones in a Binary Computer", *Communications of the ACM 3*, 5 (May 1960), 322.

[Wells] Wells, David, *The Penguin Dictionary of Curious and Interesting Numbers*, Penguin Books, 1997.

[Will] Willans, C. P., „On Formulae for the nth Prime Number", *The Mathematical Gazette 48* (1964), 413 – 415.

[Wor] Wormell, C. P., „Formulae for Primes", *The Mathematical Gazette 51* (1967), 36 – 38.

Skorowidz

#define, 59

A

abs(), 39
absolute value, 183
add, 202
addis, 20
Advanced Encription Standard, 147
AES, 147
algorytm
  AES, 147
  DES, 147
  dzielenia bez znaku, 208, 213
  Euklidesa, 220
  FFT, 125
  generowania krzywej Hilberta, 271
  Gospera, 107, 108
  magic, 204, 210, 215
  mnożenia, 151
  przesuwania i odejmowania, 173
  Rijndael, 147
  szybkie przekształcenie Fouriera, 125
  Triple DES, 147
  wykrywania cykli, 108
  zewnętrznego tasowania zupełnego, 128
  znajdowania liczby magicznej, 189
alignment amount, 64
alloca(), 167
alternatywa, 24
  bitowa, 25
  logiczna, 29
ALU, 55
analiza
  strukturalna, 286
  zakresu, 74
and immediate, 64, 205
and not, 57
anonimowe unie, 101
anonymous union, 101
APL, 137
architektura RISC, 12
argumenty, 13, 14
arytmetyka
  IEEE, 289
  komputerowa, 13
asembler, 12

B

B/V, 137
bajty, 14
big-endian, 101, 121, 129
bit, 13, 14, 23
  najmniej znaczący, 25
  odwracanie kolejności, 121
  parzystość, 94
  podzbiory, 26
  przeniesienia, 39
  przeniesienia wejściowego, 41
  przepełnienia, 44
  reversal, 143
  rozszerzenie o znak, 31
  stanu, 41
  tasowanie, 126
  transponowanie macierzy, 128
  zmiana kolejności, 149
bitgather, 146, 147
bitsize, 103
borrow, 44
borrow-in, 44
bounded minimalization, 309
branch, 235
  if negative, 42
  on carry, 68
  true, 59

C

C, 14
  #define, 59
  do, 16
  for, 16
  kontrola przebiegu pętli, 16
  while, 16
  wyrażenia, 15, 16
C++, 14
całkowitoliczbowe potęgowanie, 241
  rozkład binarny n, 241
całkowitoliczbowy pierwiastek kwadratowy, 231, 320
  algorytm sprzętowy, 237, 238, 240
  metoda Newtona, 231
  przeszukiwanie binarne, 234, 235, 236
całkowitoliczbowy pierwiastek sześcienny, 239
carry, 39, 44
carry-in, 41, 44, 50
ceiling, 64
characteristic function, 307
clp2(), 65
CMPULE, 172
commoning, 12
compare trap, 71
compare-branch, 226, 228
comparison predicate, 35
complement, 118
compress, 137, 138, 140, 141
condition
  code, 49
  register, 40
conditional
  and, 46
  move, 20, 52, 56
CROSSES, 68
cube root, 239
cykl zegara, 21
cykliczny binarny kod Graya, 264
czas wykonania, 21
czujnik położenia, 268
czynniki wieloelementowe, 151

D

Data Encryption Standard, 146
denormalized numbers, 290
DES, 146, 147
deszyfrowanie, 146, 147
difference or zero, 54
divide
  long signed, 178
  long unsigned, 177
  step, 176
divide and conquer, 86
długie dzielenie
  bez znaku, 173
  ze znakiem, 170
do, 16
dodawanie, 24, 156
  bez znaku, 43, 44
  kod warunkowy, 49
  liczb binarnych, 28
  liczb podwójnej długości, 51
  łączenie z operacjami logicznymi, 27
  ze znakiem, 41, 43, 44
dopełnienie, 118
  bitowe, 81
doubleword, 14, 147
doz(), 36, 39, 54, 55
drzewo przeszukiwania binarnego, 106
dwusłowa, 14
dziel i zwyciężaj, 86
dzielenie, 27, 48
  algorytm D. Knutha, 165
  algorytm przesuwania i odejmowania, 173
  bez znaku, 20, 46, 170, 205, 209, 212, 213, 226
  bez znaku przez dzielniki większe od 1, 207
  bez znaku przez liczbę 3, 205
  bez znaku przez liczbę 7, 206
  długie, 198
  długie bez znaku, 173
  długie ze znakiem, 170
  dokładne przez stałe, 217
  dzielnik, 201, 202, 212
  kompilator, 197
  krok, 176
  krótkie, 170, 176
  krótkie ze znakiem, 171
  liczb całkowitych, 161
  licznik, 184
  magiczna liczba, 189, 200
  moduł, 162, 163, 215
  multiplikatywna odwrotność, 218
  nieprzywracające, 173
  nonrestoring, 173
  normalizacja dzielnika, 167
  podłoga, 162, 215
  przez 3, 184
  przez 5, 186
  przez 7, 186
  przez stałą, 181, 225
  przyrostowe z resztą, 210
  przywracające, 173
  restoring, 173
  reszta ze znakiem, 182, 184
  sprawdzanie zerowej reszty, 225
  system o podstawie –2, 254

twierdzenie ZRS, 228
wartości long bez znaku, 177
wartości long ze znakiem, 178
wartości wieloelementowych, 165
wartości wieloelementowych ze znakiem, 169
ze znakiem, 170, 172, 227
ze znakiem przez dzielniki mniejsze od –2, 195
ze znakiem przez dzielniki większe od 2, 187
ze znakiem przez potęgę liczby 2, 181
dzielnik, 202
dzielenie bez znaku, 212
liczba magiczna, 201

E

ekstrakcja, 137, 142
uogólniona, 138
equivalence, 57
error-correcting codes, 93
exact division, 217
exclusive or, 256, 266
extract, 143

F

Fast Fourier Transform, 125
FFT, 125
flip, 122
floor, 64
flp2(), 64, 65
for, 16
formula functions, 306
Fortran, 34, 242
funkcje
charakterystyczne, 307
clp2(), 65
doz(), 55
elementarne, 307
flp2(), 65
linijki, 107, 109
max(), 55
maxOR(), 83
min(), 55
minOR(), 83
mulhs(), 154
mulhu(), 154
osadzone, 102
pierwotnie rekurencyjne, 311
porównania trójwartościowego, 33
rekurencyjne, 311
remu(), 123
rzutowania, 306
signum(), 32, 33
strlen(), 111
wzorowe, 306

G

Galois field, 97
GCC, 93, 102
generalized
extract, 138
unshuffle, 144
GF(2), 97
gradual underflow, 290

H

haker, 12
halfwords, 14
Hamming distance, 93
hiperkostka, 268
HLL, 114

I

IDIM(), 55
IEEE, 101, 289
IEEE 754, 289
I-fetching, 135
iloczyn logiczny z wartością natychmiastową, 205
inner perfect shuffle, 127
insert, 142
instruction scheduling, 12
instrukcje, 17
bitgather, 147
dodawania, 22
dzielenia ze znakiem, 172
dzielenie bez znaku, 20
ekstrakcji, 142, 143
li, 200
lswi, 35
ładowania, 21
ładowania wartości bezpośredniej, 21
mulhs, 200
nadzorcze, 21
obliczania reszty z dzielenia, 20
obliczeniowe, 17
porównania, 21, 22
przesunięcia w prawo ze znakiem, 20
pułapek, 21
RISC, 18
równoważności, 57
shift right logical, 205
shrxi, 206
skoków, 181
warunkowego przeniesienia wartości, 20
wstawiania, 142
zapisywania, 21
zapisywania wartości bezpośrednich, 21
zaplanowane, 22

integer
logarithm, 243
square root, 231
ISIGN, 34
ISO C, 14

J

jednostka
arytmetyczno-logiczna, 55
logiczna, 22
na ostatniej pozycji, 291
przesuwająca, 22
sumująca, 22
język
APL, 137
C, 11, 14
C++, 14
Fortran, 34
programowania, 11

K

kod
korekcyjny, 93
maszynowy, 12
operacji, 59
warunkowy, 49
wynikowy, 12
kod Graya, 109, 263
cykliczny binarny, 264
konwersja, 264
odbiciowy binarny, 263
ujemnobinarny, 267
zwiększanie liczb całkowitych, 266, 267
kompilator, 12, 59
dzielenie, 197
dzielenie bez znaku, 210
GCC, 102
XLC, 102
kompresja, 128, 137, 143, 144
B/V, 137
do lewej strony, 143
prefiks równoległy, 140
kongruencja, 162
koniunkcja, 24, 31
bitowa, 25
warunkowa, 46
z wartością bezpośrednią, 64
z zaprzeczeniem, 57
kontrola ograniczeń, *Patrz* ograniczenia
krok dzielenia, 176
krótkie dzielenie, 170, 176
ze znakiem, 171
kryptografia, 126
permutacje, 146
krzywa
Moore'a, 285
Peano, 269, 285
krzywa Hilberta, 269
algorytm nierekurencyjny generowania, 285
algorytm rekurencyjny generowania, 271
generator, 272
metoda Lama i Shapiro, 276, 281
metoda równoległego dodawania przedrostka, 279
odległość od krzywej na podstawie współrzędnych, 280
stopnia n, 269
współrzędne punktu w odległości s od krzywej, 273
zwiększanie wartości współrzędnych, 282

L

li, 212
liczby
całkowite, 13
Fermata, 299
końcowych zer, 27
magiczne, 189, 201, 216
podnormalne, 289
podwójnej długości, 51
wiodących zer, 27, 66, 80, 103
zaokrąglanie, 63
zdenormalizowane, 289, 290
zespolone, 259, 261
znormalizowane, 290
liczby pierwsze, 299
funkcje wzorowe, 306, 307
ograniczona minimalizacja, 309
wyszukiwanie, 299
wzory Willansa, 301
wzory Wormella, 305
liczby zmiennopozycyjne, 101, 289
IEEE, 289
NaN, 297
podwójna precyzja, 290
pojedyncza precyzja, 290
porównywanie, 292
porównywanie stałopozycyjne, 293
QNaN, 297
rozkład cyfr wiodących, 293
SNaN, 297
stopniowy niedomiar, 290
wartości, 295
licznik, 184
little-endian, 101, 121, 144

load
double, 135
multiple, 135
string word immediate, 35
word byte reversed, 113
logarytm, 103
logarytm całkowitoliczbowy, 243
podstawa 10, 244
podstawa 2, 244
przegląd dwóch tablic, 247
przegląd dwóch tablic bez rozgałęzień, 247
przegląd jednej tablicy, 249
przegląd jednej tablicy bez rozgałęzień, 249
przeszukiwanie tablicy, 245
wielokrotne mnożenie przez 10, 245
zmodyfikowane przeszukiwanie binarne, 246
logical units, 22
logiczne przesunięcie
w lewo, 31
w prawo, 205
long
division, 198
long, 91
lookup table, 12, 90
loop detection, 107
lswi, 35
lswx, 35
lwbrx, 113

Ł

ładowanie, 21
bajtów, 129
danych, 135
słowa z odwróceniem bajtów, 113
wartości z pamięci, 129
wielu wartości, 135
łączenie dodawania z operacjami logicznymi, 27

M

macierz transponowana, 129
magic, 210
magiczna liczba, 189, 200, 216
dzielenie bez znaku, 210
unikatowość, 201
maszyna 4-bitowa, 313
dodawanie, 313
dzielenie krótkie bez znaku, 316
dzielenie krótkie ze znakiem, 315
mnożenie bez znaku, 315
mnożenie ze znakiem, 314
odejmowanie, 314
reszta z dzielenia krótkiego bez znaku, 317
reszta z dzielenia krótkiego ze znakiem, 316
maszyna MMIX, 163
max(), 55
maxOR(), 83
maxXOR(), 83
metody
Newtona, 319
śledzenia promieni, 286
min(), 55
minOR(), 83
minXOR(), 83
MIPS, 35, 41
MMIX, 163
mnemonik, 17, 19
mnożenie, 24, 45
bardziej znacząca połowa 64-bitowego iloczynu, 154
bez znaku, 47, 153, 213
kod warunkowy, 50
konwersja bardziej znaczącej połowy 64-bitowego lioczynu, 155
liczb wieloelementowych, 151
liczb wieloelementowych całkowitych ze znakiem, 152
macierzy bitowych, 97
przepełnienie, 45
przez 3, 87
przez stałą, 156
przez wartość natychmiastową, 185
z dodawaniem, 60
ze znakiem, 46, 213
ze znakiem dla bardziej znaczącej części wyniku, 184
MOD(), 162
model wykonawczy, 17
moduł, 162
move register, 89
moveq, 20
mulhs, 154, 156, 184, 186, 200, 202, 213
mulhu, 154, 212
multiplekser, 146
multiplikatywna odwrotność, 218, 220, 225
algorytm Euklidesa, 220
metoda Newtona, 222
multiply
add, 60
high signed, 154, 156, 184
high unsigned, 154
immediate, 185, 205
MUX, 122, 146

N

nabs(), 36, 39
NaN, 289
napisy, 111
neg(), 182
negacja, 24
nibbles, 14
nierówności, 29
nieskończoność, 289
nlz(), 27, 36, 37, 65, 113, 234, 243
normalized numbers, 290
notacja arytmetyki komputerowej, 13
ntz(), 27, 105
number of leading zeros, 27, 66, 234
number of trailing zeros, 27
numerator, 184

O

obliczanie reszty z dzielenia, 20
obróć z rozszerzeniem, 206
odbicie bitowe, 143
odbiciowy binarny kod Graya, 263, 264
odczyt podwójnej wartości, 135
odejmowanie, 29, 43
  bez znaku, 44
  kod warunkowy, 50
  liczb podwójnej długości, 51
  pierwszego stopnia, 55
  wartości bezpośredniej, 69
  ze znakiem, 41, 44
odległość Hamminga, 93
odrzucanie dziewiątek, 123
odtasowanie, 127
  zewnętrzne, 127
odwracanie kolejności bitów, 121, 122
  wariacje, 122
odwrotność multiplikatywna,
  Patrz multiplikatywna odwrotność
odwrócone odejmowanie, 56
ograniczenia, 71, 72
  liczby całkowite, 71
  liczby ze znakiem, 76, 83
  operacje logiczne, 78
  operacje ze znakiem, 78
  potęgi dwójki, 63
  przekazywanie, 76
  różnica, 74
  suma, 74
  transformacje, 73
ograniczona minimalizacja, 309
opcode, 59
operacje
  kompresji, 144
  logiczne, 30
  owce i kozły, 143
operatory, 14
optymalizacja, 12
ori, 20
outer perfect unshuffle, 148
outer unshuffle, 127
overflow, 17, 40
owce i kozły, 143

P

parallel prefix, 95
parity, 94
parzystość, 94
  wartości 7-bitowe, 96
perfect shuffle, 126
permutacje, 143
pętle, 16
pierwiastek
  kwadratowy, 231
  sześcienny, 239
pobieranie instrukcji, 135
podłoga, 64, 162, 163, 182
podstawy systemów liczbowych, 251
podzbiory, 26
  przeglądanie, 26
  suma, 26
pola rejestrów, 21
pop(), 27
population count, 27, 85
porównanie, 21, 22, 35
  bit przeniesienia, 39
  liczb zmiennopozycyjnych, 292
  nieuporządkowane, 289
  stałopozycyjne, 293
  trójwartościowe, 33
  ze znakiem, 37
porządek bitów, 101
potęgowanie
  całkowitoliczbowe, 241
  liczba 2, 242
  ograniczenia liczby 2, 63
pow2(), 243
pożyczka, 44
  wejściowa, 44
półbajty, 14
półsłowa, 14
PP-XOR, 138, 139, 280
prawa DeMorgana, 81
prawostronne bity, 23

predykat porównania, 35
rozstrzygania, 40
prefiks równoległy, 138
preprocesor, 59
primitive recursive, 311
procesor
high-end, 21
MIPS, 35
PowerPC, 35
RISC, 17, 21, 35
programowanie obiektowe, 14
projection functions, 306
przekroczenie
ograniczeń potęgi dwójki, 67
zakresu, 17
przekształcenia poprzeczne, 136
przeniesienie, 34, 39, 44
wartości rejestru, 89
wejściowe, 44
znaku, 34
przepełnienie, 40, 43
dodawanie ze znakiem, 41
dzielenie, 48
mnożenia, 45
mnożenia bez znaku, 47
odejmowanie ze znakiem, 41
ustawianie bitu, 44
ze znakiem w kierunku dodatnim, 43
ze znakiem w kierunku ujemnym, 43
przesłanie warunkowe, 52
przesunięcie
bez znaku, 32
cykliczne, 26, 50, 122
cykliczne w lewo, 122, 130
cykliczne w prawo, 90
dwusłowne w lewo, 175
liczb o podwójnej długości, 52
logiczne, 32
logiczne w lewo, 31
w lewo, 24, 26, 156
w lewo liczb o podwójnej długości, 52
w prawo, 26
w prawo liczby podwójnej długości bez znaku, 52
w prawo liczby podwójnej długości
ze znakiem, 52
w prawo ze znakiem, 20, 31, 32, 33, 53, 181
z obrotem, 226
przeszukiwanie, 111
binarne, 86, 97
przetwarzanie
obrazów, 286
równoległe, 24
przydział rejestrów, 12
pułapki, 21, 48
porównania, 71

Q

QNaN, 297
quicksort, 86
quiet NaN, 297

R

range analysis, 74
rastrowanie, 286
ray tracing, 286
reflective binary Gray code, 263
register
allocation, 12
fields, 21
reguła Hornera, 60
rejestry
ogólnego przeznaczenia, 40
warunkowe, 40
wymiana dwóch pól, 57
wymiana wartości, 56
wymiana warunkowa, 58
rem(), 123, 162
remu, 123
remu(), 123
reszta z dzielenia, 88, 123
bez znaku, 88
przez potęgę liczby 2, 182
przez wartość bezpośrednią, 96
ze znakiem, 184
rev(), 125
reverse subtract, 56
reversing bits, 121
Rijndael, 147
RISC, 12, 17, 21, 35
instrukcje, 18, 19
mnemoniki, 19
rlwimi, 122
rotate
left word immediate then mask insert, 122
with extend, 206
roxr, 206
rozdzielanie owiec i kozłów, 144
rozgałęzienie, 71
w przypadku przeniesienia, 68
w przypadku wartości ujemnej, 42
w przypadku wartości zero, 113
w przypadku wyniku fałsz, 59
w przypadku wyniku prawda, 59
rozkład binarny, 241
rozmiar bitowy, 103
rozszerzenie o znak, 31
rozszerzone przesunięcia w prawo o wartość
natychmiastową, 206

rozszerzony algorytm Euklidesa, 220
równanie pierwotnie rekurencyjne, 311
równoległe przetwarzanie instrukcji, 21
równoległość, 12
równoległy prefiks, 95
równoważność, 55, 57
różnica
  symetryczna, 26, 58, 61, 94
  wykluczająca, 256
ruler function, 107

S

SAG, 144, 145, 146, 147
scan, 95
schemat Hornera, 60
set on less than unsigned, 51, 247
shearing transformation, 136
sheep and goats, 144
shift
  and add, 157
  and subtract, 173
  left double, 52, 175
  left logical, 31
  right, 212
  right double signed, 52
  right double unsigned, 52
  right extended immediate, 206
  right logical, 205
  right signed, 31, 32, 181
  right signed 31, 238
short division, 170
shrsi, 182, 202
shrxi, 206
sign extension, 31
signaling NaN, 297
signed
  long division, 170
  overflow in the negative direction, 43
  overflow in the positive direction, 43
signum(), 32, 33
skanowanie, 95
skok, 181
SLTU, 51, 172
słowa, 13, 14
SNaN, 297
sortowanie
  stabilne, 145
  szybkie, 86
space filling, 269
SPEC, 20
stabilne sortowanie o podstawie 2, 145
stable binary radix sort, 145
standard IEEE, 289
status bits, 41
stopniowy niedomiar, 290
store
  double, 135
  multiple, 135
strategia dziel i zwyciężaj, 86
strlen(), 111
subtract from immediate, 69, 246
sufit, 64
Superoptimiser, 39
supervisor services, 17
switch, 17, 74
system liczbowy
  dziesiętny, 252
  operacje jednorodne, 257
  poczwórnie urojony, 261
  podstawa, 262
  podstawa –1 – i, 261
  podstawa –1 + i, 258
  podstawa –2, 251
  ujemnobinarny, 251
szeregowanie instrukcji, 12
szybkie przekształcenie Fouriera, 125
szyfrowanie, 146

Ś

śledzenie promieni, 286

T

tablice
  dynamiczne, 72
  kontrola zakresów, 72
  przeglądowe, 12, 90
  zliczanie jedynek, 92
tasowanie, 137
  bitów, 126
  wewnętrzne, 126
  zewnętrzne, 126
  zupełne, 126
testy SPEC, 20
three-valued compare, 33
transfer of sign, 34
transformacje oparte na indeksach, 148
transponowanie macierzy bitów, 128
  32x32 bity, 132, 135
  8x8 bity, 131
  przekształcenia poprzeczne, 136
transpozycja, 137
trap, 21, 48
Triple DES, 147
twierdzenie Wilsona, 305
tworzenie krzywej Hilberta, 271

U

ujemnobinarny
kod Graya, 267
system liczbowy, 251
ułamki, 289
postać przesunięta, 289
postać ze znakiem, 289
unit in the last position, 291
uogólnione
ekstrakcje, 138
odtasowanie, 144
permutacje, 143
urządzenie osadzone, 22
usługi zarządcze, 17
usuwanie podwyrażeń wspólnych, 12
uzupełnienie dwójkowe, 13

V

VLIW, 99

W

wartości, 13
bezwzględne, 30, 183
bezwzględne wielobajtowe, 53, 54
NaN, 289, 297
wymiana, 59
wymiana między rejestrami, 56
warunkowe przeniesienie wartości, 20, 56
wektor, 13
wewnętrzne tasowanie zupełne, 127
while, 16
wielkości
populacji, 27, 85
wyrównania, 64
wieże Hanoi, 109
words, 14
wstawianie, 142
wykrywanie cykli, 107, 108
wymiana wartości, 59
między rejestrami, 56
wypełnianie przestrzeni, 269
wyrażenia
arytmetyki komputerowej, 13
arytmetyki matematycznej, 13
C, 15
nierówności, 29
wyszukiwanie
bajtu o wartości 0, 111
ciągu n jedynek, 117, 119
lewostronnego bajtu zerowego, 112, 113
pierwszego bajtu o wartości 0, 120
prawostronnego bajtu zerowego, 115, 116
słów, 111
wartości w przedziale, 116
wzory
Willansa, 301
Wormella, 305

X

XLC, 102, 170
XORI, 172

Z

zaokrąglanie
liczb całkowitych, 63
w dół, 64, 66
w górę, 63, 64, 66
w kierunku 0, 63
zapis
bajtu w pamięci, 129
danych, 135
podwójnej wartości, 135
wielu wartości, 135
zero
oznacza 2^n, 34
wiodące, 97
zewnętrzne odtasowanie, 127
zupełne, 148
zliczanie
bitów, 85
jedynek, 85, 89
jedynek w macierzy, 92
jedynek w tablicy, 92
parzystości słowa, 94
populacji, 92, 93, 100, 104, 143
rzadka populacja, 89
zliczanie zer końcowych, 104
drzewo przeszukiwania binarnego, 106
mniejsze wartości bezpośrednie, 105
pętle zliczające, 106
przeszukiwanie binarne, 104
zliczanie zer wiodących, 97
jednoczesne zliczanie z obu stron, 99
metody zmiennoprzecinkowe, 101
propagacja w prawo i zliczanie jedynek, 100
przeszukiwanie binarne, 97
zliczanie zer wiodących
przeszukiwanie binarne bez rozgałęzień, 100
przeszukiwanie binarne, odliczanie w dół, 98
przeszukiwanie binarne, odliczanie w dół w formie pętli, 99
zmiana kolejności bitów, 149
znak liczby, 32
ZRU, 228