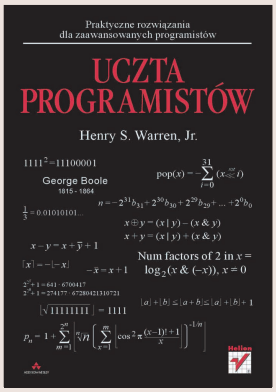
Uczta programistów



Autorzy: Henry S. Warren, Jr.
Tłumaczenie: Marek Pętlicki (rozdz. 1 – 9),
Bartłomiej Garbacz (rozdz. 10 – 16, dod. A, B)
ISBN: 83-7361-220-3
Tytuł oryginału: Hacker's Delight
Format: B5, stron: 336

Do tworzenia wydajnych programów nie wystarczy teoretyczna wiedza o algorytmach, strukturach danych i inżynierii oprogramowania. Istnieje pokaźna liczba sztuczek, sprytnych technik i praktycznych rozwiązań, których znajomość jest niezbędna każdemu programiście.

Niniejsza książka zawiera pokaźny zestaw technik, które pomogą zaoszczędzić sporo czasu. Techniki te zostały opracowane przez twórców kodu poszukujących eleganckich i wydajnych sposobów tworzenia lepszego oprogramowania. W „Uczcie programistów" doświadczony programista Hank Warren dzieli się z Czytelnikami znanymi sobie sztuczkami, które zgromadził wraz z imponującym doświadczeniem w dziedzinie programowania aplikacji i systemów operacyjnych. Większość z tych sztuczek jest niezwykle praktyczna, niektóre zostały przedstawione jako ciekawostki lub zaskakujące rozwiązania. Ich zestawienie stanowi niesamowitą kolekcję, która jest w będzie pomocna nawet dla najbardziej doświadczonych programistów w rozszerzeniu ich umiejętności.

W książce opisano następujące zagadnienia:

- Obszerna kolekcja użytecznych sztuczek programistycznych
- Drobne algorytmy rozwiązujące często spotykane problemy
- Algorytmy kontroli przekroczenia ograniczeń
- Zmiana kolejności bitów i bajtów
- Dzielenie całkowite i dzielenie przez stałe
- Elementarne operacje na liczbach całkowitych
- Kod Gray'a
- Krzywa Hilberta
- Formuły wyznaczania liczb pierwszych

Niniejsza książka jest doskonałą pozycją dla wszystkich programistów, którzy mają zamiar tworzyć wydajny kod. „Uczta programistów" nauczy Cię tworzenia aplikacji wysokiej jakości – wyższej niż wymagana a uczelniach i kursach programowania.

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

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

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:

$$x = ((x \;\&\; \mathbf{0x00FF00FF}) \overset{rot}{>>} 8) \mid ((x \overset{rot}{<<} 8) \;\&\; \mathbf{0x00FF00FF}) \cdot$$

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: remu((***x*** ∗ **0x00082082**) & **0x01122408**, **255**)

7 bitów: remu((***x*** ∗ **0x40100401**) & **0x442211008**, **255**)

8 bitów: remu((***x*** ∗ **0x202020202**) & **0x10884422010**, **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:

$$s \leftarrow (x * \mathbf{0x00020202}) \,\&\, \mathbf{0x01044010}$$

$$t \leftarrow (x * \mathbf{0x00080808}) \,\&\, \mathbf{0x02088020}$$

$$\text{remu}(s + t, \mathbf{4095})$$

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:

$$s \leftarrow (x * \mathbf{0x01001001}) \,\&\, \mathbf{0x84108010}$$

$$t \leftarrow (x * \mathbf{0x00040040}) \,\&\, \mathbf{0x00841080}$$

$$\text{remu}(s + t, \mathbf{1023})$$

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:

$$
\begin{aligned}
\text{Najpierw wyliczamy}\quad & \boldsymbol{s} \leftarrow \mathrm{nlz}(\neg\boldsymbol{x})\\
\text{następnie:}\quad & \boldsymbol{x} \leftarrow \boldsymbol{x} \oplus (\mathbf{0x80000000} \overset{s}{>>} \boldsymbol{s})\\
\text{lub:}\quad & \boldsymbol{x} \leftarrow ((\boldsymbol{x} << \boldsymbol{s}) + \mathbf{0x80000000}) \overset{u}{>>} \boldsymbol{s}
\end{aligned}
$$

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 8`m`×8`n` bitów. Podobnie parametr `B` stanowi adres pierwszego bajtu podmacierzy 8×8 macierzy docelowej o wymiarach 8`n`×8`m` bitów. Oznacza to, że cała macierz źródłowa ma wymiary 8`m`×`n` bajtów, natomiast macierz wyjściowa ma wymiary 8`n`×`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) mod *n* bitów;
- przesunięcie cykliczne wiersza *i* w prawo o (*i* + 1) mod *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 ***m*** i słowa ***x***, bity w ***x***, dla których odpowiednie bity w masce ***m*** mają wartość 1 zostają skopiowane do wyniku i przesunięte (skompresowane) do prawej. Załóżmy na przykład, że operujemy na słowie ***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 $\text{pop}(\overline{\boldsymbol{m}})$ 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