

# O sierotce, co chciała się mózgiem elektronowym wyręczyć, czyli kolejność znowu ma znaczenie

*Tomasz IDZIASZEK*

Wyobraźmy sobie biedną sierotkę, której macocha nakazała oddzielić groch od fasoli. Chcąc nie chcąc, dziewczę siada w kącie izby przed pokaznym kopcem pomieszanego grochu z fasolą i zaczyna pracę. Praca jest niezwykle monotonna: sierotka bierze nasiono z górki i jeśli to fasola odrzuca je na lewą stronę, a jeśli groch – na prawą; i tak w kółko. Sierotka jest całkiem porządna, więc aby nie rzucać nasionkami po całej izbie ustaliła, że nasionka fasoli odkłada na lewą stronę lewą ręką, a nasionka grochu odkłada prawą ręką. Ponieważ na wybór ręki, którą sięgnie po kolejne nasionko musi się zdecydować zanim rozpozna jego rodzaj (w izbie jest dość ciemno), więc nierzadko będzie zmuszona do przełożenia nasionka z jednej ręki do drugiej, co oczywiście będzie spowalniać jej pracę.

Powiedzmy, że sierotka w większości przypadków wyciąga z górki nasionko fasoli (takie już ma szczęście). Zauważywszy to, może postąpić praktycznie i na początku zawsze wyciągać nasionka lewą ręką, tak by zminimalizować liczbę przełożeń nasionka z ręki do ręki, a gdy w kopczyku zostanie już w większości groch, to będzie wyciągać nasionka prawą ręką. Sierotka może mieć szczęście innego rodzaju, np. prawie zawsze co trzecie wyciągnięte nasionko to groch. W tej sytuacji mądre dziewczę stwierdza, że najbardziej opłacalny jest schemat: lewa ręka, lewa ręka, prawa ręka itd. Niestety, znając bajkowe realia, sierotka najpewniej będzie wyciągała nasionka w zupełnie losowej kolejności, zatem żadne wymyślne strategie nie pomogą jej w uniknięciu marnego losu i średnio co drugie nasionko będzie musiało być przełożone z ręki do ręki.

Czytelnik Postępowy od razu zauważy, że wszystkie kłopoty skończyłyby się, gdyby na miejscu sierotki postawić robota z chwytnym ramieniem, fotokomórką i mózgiem elektronowym. Nie dość, że wykonałby on zadanie sprawniej i na pewno bezbłędnie, to nie rozwodziłby się długo nad wyborem strategii postępowania, a praca zajęłaby mu tyle samo czasu, niezależnie od kolejności w jakiej wyciągałby ziarenka z górki.

I mogliśmy zakończyć ten artykuł powyższym morałem gdyby nie to, że nie jest on do końca prawdziwy. Okazuje się, że w świecie maszyn nie wszystko jest takie jasne i poukładane, a w elektronowym mózgu może kryć się coś ze sprytniej sierotki. Kto ciekawy, tego zapraszam do eksperymentu.

★

Ponieważ nie wszyscy Czytelnicy dysponują odpowiednim sprzętem tudzież zapasem grochu i fasoli, więc eksperyment zasymlujemy na domowym komputerze, pisząc prosty program. Program będzie czytywał z wejścia zero-jedynkowy ciąg, który kodować będzie rodzaje kolejnych nasionek. Zamiast sortowania, program będzie miał na celu policzenie nasionek, tzn. policzenie ile zer i ile jedynek występuje w wejściowym ciągu. Jeśli te wartości będzie przechowywał na zmiennych  $x_0$  i  $x_1$ , to po każdorazowym wczytaniu nowego bitu  $b$  należy wykonać taki kod:

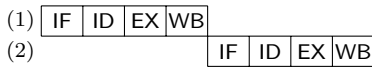
```
if b = 0 then  $x_0 := x_0 + 1$ 
else  $x_1 := x_1 + 1$ ;
```

Skompilujmy nasz program i uruchommy go dla różnych ciągów zero-jedynkowych równej długości, w których dokładnie połowa bitów to zera. Wydawałoby się, że czas działania naszego programu musi być zawsze taki sam, w końcu wykonamy dokładnie tyle samo operacji porównań i dodawania, co więcej wartości zmiennych  $x_0$  i  $x_1$  zwiększymy tyle samo razy. Okazuje się jednak, że na moim komputerze dla ciągu  $B_1 = 0^{n/2}1^{n/2}$  długości  $n = 10^7$ , w którym pierwsza połowa ciągu to zera, a druga połowa to jedynek, program działa 0,11 s, natomiast dla ciągu  $B_2$ , w którym pozycje zer i jedynek są losowe, program działa 0,16 s, czyli prawie o połowę dłużej! Nie jest to bynajmniej błąd pomiaru czasu, gdyż kolejne uruchomienia programu potwierdzają pierwotną obserwację. Ponadto w obu uruchomieniach programu wczytywanie danych zajmuje 0,03 s, więc dysproporcja czasu „właściwych obliczeń” jest w rzeczywistości jeszcze większa.

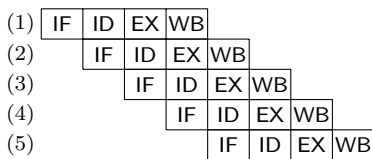
Jaka jest przyczyna takiego zachowania? Czyżby domowy komputer z jakichś powodów, radził sobie lepiej z „przewidywalnymi” danymi, tak jak nasza sierotka? A jeśli tak jest w istocie, to jakie są tego powody?

Jak wiemy, kompilacja powoduje przetworzenie kodu programu na ciąg instrukcji niskiego poziomu. Na przykład zwiększenie wartości zmiennej znajdującej się w rejestrze  $x$  procesora mogłoby być zapisane w wewnętrznym języku komputera jako

INC  $x$



Rys. 1. Na wykonanie instrukcji procesora składają się cztery kroki: IF (ang. *instruction fetch*), ID (*instruction decode*), EX (*execute*), WB (*write back*). Na wykonanie dwóch instrukcji potrzebujemy 8 jednostek czasu.



Rys. 2. Dzięki przetwarzaniu potokowemu (ang. *pipelining*) w 8 jednostkach czasu jesteśmy w stanie wykonać 5 instrukcji w przypadku pustego potoku i aż 8 instrukcji w przypadku wypełnionego potoku.

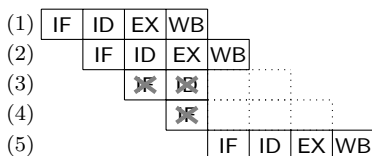
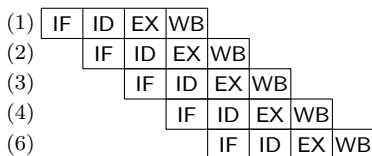
Na wykonanie powyższej instrukcji składa się kilka kroków. Po pierwsze instrukcja musi być pobrana z pamięci. Następnie musi zostać zdekodowana: procesor odkrywa, że chodzi o zwiększenie zmiennej z rejestru. W kolejnym kroku należy wykonać instrukcję: przesłać wartość zmiennej z rejestru do jednostki arytmetycznej procesora i dokonać zwiększenia. I ostatecznie należy przesłać uaktualnioną wartość z powrotem do rejestru. (Jest to przykład dość uproszczony, w ogólnym przypadku wykonanie instrukcji może składać się z większej liczby kroków, np. przekopiowanie danych z pamięci do rejestru procesora. We współczesnych procesorach liczba takich kroków może sięgać kilkudziesięciu.) Widać więc, że wykonanie pojedynczej instrukcji nie jest takie banalne. Jeśli założymy, że każdy krok wykonuje się w jednej jednostce czasu, to na wykonanie  $n$  instrukcji potrzebujemy  $4n$  jednostek czasu (rys. 1).

Zauważmy jednak, że skoro każdy z kroków jest wykonywany przez inną część procesora, to można przyspieszyć cały proces, umożliwiając tym częściom procesora pracę równoległą przez przetwarzanie kilku instrukcji naraz. (Tak jak na linii produkcyjnej w fabryce samochodów ekipa montująca podwozie pracuje równoległe z ekipą malującą karoserię – pracują po prostu na innych egzemplarzach.) Można postąpić następująco: w pierwszym kroku pobieramy pierwszą instrukcję z pamięci. W drugim kroku dekodujemy tę instrukcję, ale w tym momencie część procesora odpowiedzialna za pobieranie instrukcji jest bezrobotna, wobec tego możemy jednocześnie pobrać z pamięci drugą instrukcję. W trzecim kroku wykonujemy pierwszą instrukcję, dekodujemy drugą i pobieramy z pamięci trzecią. Dzięki takiemu potokowemu przetwarzaniu instrukcji na wykonanie  $n$  z nich potrzebujemy tylko  $n + 3$  jednostek czasu (rys. 2).

Jak każdy świetny pomysł tak i ten ma pewne wady. Zauważmy, że przyjęliśmy tu milczące założenie, że zanim zakończymy wykonywanie danej instrukcji musimy być w stanie rozpocząć wykonywanie następczej, a nawet trzeciej instrukcji z kolei. W szczególności musimy wiedzieć co to będą za instrukcje. O tym, że nie zawsze mamy tę wiedzę, możemy się przekonać kompilując nasz pierwszy program:

- (1) CMP  $b, 0$
- (2) IFEQJMP (5)
- (3) INC  $x_1$
- (4) JMP (6)
- (5) INC  $x_0$
- (6) ...

W pierwszej instrukcji zapisanej pod adresem (1) wykonujemy porównanie zmiennej  $b$  z zerem. Następcza instrukcja jest instrukcją skoku warunkowego: jeśli liczby porównywane ostatnio były równe, to instrukcja ta powoduje skok do instrukcji zapisanej pod adresem (5), w przeciwnym wypadku nie dzieje się nic. Odpowiada to wybraniu odpowiedniej gałęzi w instrukcji **if**. Jeśli wykonaliśmy skok (pierwsza gałąź), to w instrukcji (5) zwiększamy wartość  $x_0$ . Jeśli skoku nie wykonaliśmy (druga gałąź), to w instrukcji (3) zwiększamy wartość  $x_1$ , a następnie wykonujemy bezwarunkowy skok do instrukcji (6). Kluczowa jest w tym programie instrukcja skoku warunkowego **IFEQJMP**: mianowicie dopóki jej nie wykonamy nie wiemy, czy następczą po niej będzie instrukcja (3), czy też instrukcja (5), nie wiemy więc, która z nich powinna być jako następcza uwzględniona w potoku. Procesor jest zatem w sytuacji naszej sierotki, która dopóki nie dotknie nowego ziarenka, nie będzie wiedziała czy należało użyć lewej, czy też prawej ręki.



Rys. 3. Sytuacja w potoku w przypadku, gdy skok warunkowy w instrukcji (2) nie nastąpił i gdy nastąpił.

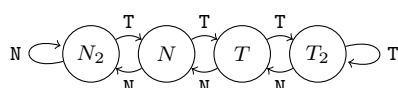
Można ten kłopot spróbować rozwiązać następująco: procesor zawsze będzie zakładał, że po instrukcji zapisanej pod adresem ( $i$ ) następczą są instrukcje pod adresami ( $i + 1$ ), ( $i + 2$ ) itd. lub pod adresem, który wskazuje skok bezwarunkowy, i te właśnie instrukcje będzie uwzględniał w potoku. Jeśli jednak instrukcja ( $i$ ) była rozkazem warunkowego skoku i po jej wykonaniu okazało się, że należy skoczyć pod adres ( $j$ ), to częściowa praca związana z wykonaniem instrukcji ( $i + 1$ ) i kolejnych jest unieważniana, a potok jest opróżniany i przetwarzanie zaczyna się na nowo od instrukcji ( $j$ ) (rys. 3).

W przypadku programów, w których jest mało skoków warunkowych lub są rzadko wykonywane, taka strategia może się opłacać. Jednak w pesymistycznym przypadku,

gdy każda kolejna instrukcja jest skokiem warunkowym, musimy opróżnić potok po wykonaniu każdej instrukcji, zatem czas wykonania  $n$  instrukcji będzie wynosił  $3n$  jednostek – nasz potok nie będzie zatem wykorzystywany w pełni efektywnie.

Zauważmy jednak, że taka strategia nie może być stosowana w naszym procesorze: przecież program dla ciągów  $B_1$  i  $B_2$  powinien działać tak samo długo, gdyż w obu przypadkach potok byłby opróżniany w połowie instrukcji (2). Taka strategia odpowiada sytuacji, gdy nasza sierotka postanowiłaby niezależnie od niczego zawsze używać lewej ręki.

Możemy przyjąć inną metodę postępowania i próbować zgadnąć, czy dany skok warunkowy będzie wykonany, czy też nie. Okazuje się, że współczesne procesory tak właśnie postępują. Co więcej, ponieważ nie znają przyszłości, więc zachowują się jak nasza sierotka i zakładają, że będą miały szczęście i na podstawie zachowań instrukcji skoku warunkowego w przeszłości będą w stanie przewidzieć jak będzie się ona zachowywała w przyszłości.



Rys. 4. Czterostanowy wskaźnik zgadujący, czy skok zostanie wykonany.

Zapiszmy historię wykonywania skoku warunkowego w postaci ciągu złożonego z liter  $T$  i  $N$ . Jedną z najprostszych strategii zgadywania jest następująca: przyjmujemy, że skok będzie wykonany, jeśli ostatnim razem był wykonany (czyli ostatnia litera w ciągu to  $T$ ). Ta strategia ma sens w przypadku długich sekwencji analogicznych wyborów, jak to ma miejsce dla ciągu  $B_1$ . Można ją trochę ulepszyć, aby była niewrażliwa na sporadyczne przekłamania w historii: trzymamy wskaźnik mogący przyjmować cztery stany:  $\{N_2, N, T, T_2\}$ . Każde wykonanie skoku powoduje zmianę stanu o jeden w prawo zgodnie z rys. 4, a niewykonanie skoku – w lewo. Zgadujemy, że skok zostanie wykonany, jeśli wskaźnik znajduje się w stanie  $T$  lub  $T_2$ .

Gdyby w naszym procesorze zastosowano powyższą strategię, wyjaśniałaby ona różnice w działaniu programu dla ciągów  $B_1$  i  $B_2$ : w przypadku pierwszego z nich nasza strategia zgadywania nie zadziałała tylko kilka razy (w środku ciągu i być może na początku), z kolei jest ona zupełnie bezużyteczna (jak w zasadzie jakkolwiek strategia zgadywania) w przypadku drugiego z nich.

Niestety to nie wyjaśnia dlaczego dla ciągu  $B_3 = (001)^{n/3}$ , w którym dokładnie co trzeci bit to 1, czas działania jest taki sam jak dla ciągu  $B_1$ . Wszak dla ciągu  $B_3$  musielibyśmy pomylić się dla każdej jedynek, czyli co najmniej  $n/3$  razy. Okazuje się, że strategia stosowana przez współczesne procesory próbuje uwzględnić możliwość wystąpienia krótkich cykli w historii wykonywania skoku i jest z tego powodu trochę bardziej skomplikowana. Mianowicie przy podejmowaniu decyzji patrzymy na  $k$  ostatnich liter w historii i na podstawie tych liter wybieramy jeden z  $2^k$  czterostanowych wskaźników, który uaktualniamy i podejmujemy decyzję zgodnie z jego wskazaniem. W ten sposób dostajemy mechanizm, który będzie zgadywał poprawnie, o ile w historii wykonywania skoku każde  $k$  kolejnych bitów jednoznacznie wyznacza bit  $(k + 1)$ -szy. W moim komputerze jest  $k = 4$ , zatem w ciągu  $B_3$ , którego historię można opisać regułami: po każdym  $TT$  jest  $N$ , po każdym  $TN$  jest  $T$ , a po każdym  $NT$  jest  $T$ , procesor nie pomylił się ani razu (z wyłączeniem kilku potencjalnych pomyłek na początku ciągu).

Opisane strategię nie są jedynymi, które są stosowane we współczesnych procesorach. Istnieje cała gama rozwiązań, które pozwalają przewidywać typowe historie wykonywania skoku. W niektórych procesorach do problemu podchodzi się zupełnie inaczej. Zakłada się na przykład, że w skompilowanym kodzie po każdej instrukcji skoku warunkowego musi wystąpić pewna liczba zwykłych instrukcji, które będą wykonane niezależnie od tego, czy skok nastąpi, czy nie, aby po ich wykonaniu adres docelowy był już obliczony.

★

Morał z tej bajki jest taki, że nawet kolejność w jakiej podajemy dane programowi komputerowemu ma znaczenie (mimo że na pierwszy rzut oka może wydawać się to niewiarygodne). Drugi morał jest ważną wskazówką dla programisty: w kodzie, w którym ważna jest efektywność wykonania, należy, o ile to możliwe, unikać skoków. Okazuje się, że nasz pierwszy program możemy przepisać tak, by instrukcja `if` w nim nie występowała:

$$\begin{aligned} x_0 &:= x_0 + 1 - b; \\ x_1 &:= x_1 + b; \end{aligned}$$

Tak napisany program działa w czasie 0,11 s dla każdego z ciągów  $B_1$ ,  $B_2$  i  $B_3$ .