Ćwiczenie nr 2

Zastosowania instrukcji sterujących

2.1 Zasady wykonywania instrukcji przez procesor

Znaczna część instrukcji (rozkazów) wykonywanych przez procesor nie zmienia naturalnego porządku wykonywania obliczeń. Oznacza to, że po wykonaniu instrukcji, jako następna zostanie wykonana instrukcja znajdująca w kolejnych komórkach pamięci. W taki sposób wykonywane są zazwyczaj instrukcje arytmetyczne, logiczne, przesyłania i wiele innych. Na poziomie sprzętowym pobieranie instrukcji przez procesor jest sterowane przez zawartość rejestru zwanego wskaźnikiem instrukcji, niekiedy nazywany też licznikiem rozkazów lub licznikiem programu. Rejestr ten pełni kluczową rolę w pracy procesora: po wykonaniu instrukcji, rejestr wskaźnika instrukcji zawiera adres komórki pamięci, począwszy od której, w kolejnych bajtach, umieszczona jest następna instrukcja do wykonania. Jeśli więc instrukcje mają być wykonywane po kolei, to w trakcie wykonywania zawartość wskaźnika instrukcji powinna zostać zwiększona o liczbę bajtów aktualnie wykonywanej instrukcji. W procesorze Pentium (i jego poprzednikach) wskaźnik instrukcji oznaczany jest symbolem EIP. Można zatem napisać

EIP ← EIP + *liczba bajtów aktualnie wykonywanej instrukcji*

2.2 Instrukcje sterujące i niesterujące

Procesor, który potrafiłby tylko wykonywać instrukcje „po kolei" był by urządzeniem o bardzo małym zastosowaniu. W prawie wszystkich bowiem algorytmach występują rozwidlenia, co oznacza, że dalsze postępowanie zależy od uzyskanych wyników pośrednich. Zatem konieczne jest wprowadzenie do procesora mechanizmów pozwalających na zmianę naturalnego porządku wykonywania instrukcji w zależności od spełnienia lub nie pewnych warunków.

We wielu współczesnych procesorach zmiana porządku wykonywania instrukcji może być wykonana za pomocą instrukcji sterujących nazywanych też instrukcjami skoków. Instrukcje te zazwyczaj nie wykonują żadnych obliczeń, lecz wyłącznie testują spełnienie pewnych warunków, na przykład czy liczba w rejestrze akumulatora jest ujemna, czy podczas ostatniej operacji arytmetycznej wystąpiło przeniesienie, itd. W procesorach Pentium istnieje kilkadziesiąt instrukcji sterujących testujących rozmaite warunki.

Instrukcje sterujące oddziaływają na przebieg obliczeń poprzez zmianę zawartości wskaźnika instrukcji EIP. W kodzie instrukcji sterujących występuje zazwyczaj pole adresowe, w którym zapisana jest pewna liczba. Jeśli warunek testowany przez instrukcję sterującą nie jest spełniony, to wskaźnik instrukcji EIP zostaje tylko zwiększony o liczbę bajtów aktualnie wykonywanej instrukcji (sterującej). Jeśli warunek jest spełniony, to do rejestru EIP zostaje jeszcze dodana liczba umieszczona w polu adresowym. Reguły te można zapisać w postaci sformalizowanej

1. jeśli warunek nie jest spełniony:
   EIP ← EIP + *liczba bajtów aktualnie wykonywanej instrukcji*
2. jeśli warunek jest spełniony:
   EIP ← EIP + *liczba bajtów aktualnie wykonywanej instrukcji* +
   + *liczba umieszczona w polu adresowym instrukcji*

Zatem jeśli warunek jest spełniony, to naturalny porządek wykonywania instrukcji zostanie przerwany: rejestr EIP nie będzie wskazywać sąsiedniej instrukcji, lecz inną instrukcję odległą o podaną liczbę bajtów. Instrukcja ta może znajdować w komórkach pamięci o

adresach wyższych, jeśli liczba umieszczona w polu adresowym jest dodatnia, lub o adresach niższych, jeśli liczba jest ujemna. W tym ostatnim przypadku może to oznaczać, że pewna instrukcje zostaną wykonane ponownie, czyli będzie wykonywana pętla.

Instrukcje, które nie zmieniają kolejności wykonywania, nazywane są instrukcjami niesterującymi.

2.3 Rejestr znaczników

W procesorze Pentium (i jego poprzednikach) dostępny jest 32-bitowy rejestr znaczników (EFLAGS). Z punktu widzenia programowania istotne znaczenie mają niektóre młodsze bity tego rejestru. Niżej wymienione bity opisują wyniki operacji wykonywanych przez instrukcje arytmetyczne i logiczne.

- CF – znacznik przeniesienia, ustawiany w stan 1 jeśli w trakcie wykonywania instrukcji wystąpiło przeniesienie lub pożyczka;
- ZF – znacznik zera, ustawiany w stan 1, jeśli wynikiem instrukcji było zero;
- SF – znacznik znaku, ustawiany w stan 1, jeśli wynikiem instrukcji jest wartość ujemna;
- OF – znacznik nadmiaru, ustawiany w stan 1, jeśli w wyniku operacji arytmetycznej na liczbach ze znakiem wystąpił nadmiar.

2.4 Porównywanie zawartości rejestrów

W trakcie wielu algorytmów powstaje problem sprawdzenia czy liczba umieszczona w jednym rejestrze jest większa od liczby umieszczonej w innym rejestrze. Dla ustalenia uwagi przyjmijmy, że porównywane liczby znajdują się w rejestrach BX i CX. Ażeby porównać dwie liczby należy je odjąć od siebie. Odejmowanie spowoduje ustawienie znaczników wg wartości uzyskanego wyniku. Między innymi zostaną ustawione bity:

$$\mathrm{ZF} = \begin{cases} 0, \text{ gdy wynik odejmowania jest różny od zera} \\ 1, \text{ gdy wynik odejmowania jest rowny zero} \end{cases}$$

$$\mathrm{CF} = \begin{cases} 0, \text{ gdy w trakcie odejmowania nie było pożyczki} \\ 1, \text{ gdy w trakcie odejmowania była pożyczka} \end{cases}$$

Jak wiadomo, prośba o pożyczkę w odejmowaniu występuje wówczas, gdy od liczby mniejszej odejmowana jest liczba większa.
Rozpatrzmy teraz stan znaczników ZF i CF w zależności od zawartości rejestrów BX i CX:

1. jeśli BX > CX, to ZF = 0 i CF = 0;
2. jeśli BX = CX, to ZF = 1 i CF = 0;
3. jeśli BX < CX, to ZF = 0 i CF = 1.

Z powyższej analizy wynika, że dla każdego przypadku znaczniki ZF i CF przyjmują różne wartości. Zatem w celu porównania dwóch liczb wystarczy wykonać odejmowanie, a następnie odczytać stan znaczników ZF i CF. Dodatkowo zauważmy, że przypadek CF = 1 i ZF = 1 nigdy nie wystąpi, bo jak wynik odejmowania jest równy zero, to nie ma pożyczki.

Ponieważ oddzielne sprawdzanie znaczników ZF i CF byłoby niewygodne, więc konstruktorzy procesora zdefiniowali instrukcję sterującą specjalnie przewidzianą do testowania stanu obu tych znaczników. Instrukcja ta, kodowana na dwóch bajtach, oznaczona jest mnemonikiem `ja` (ang. *jump if above*).

01110111 pole adresowe

Instrukcja `ja` oddziaływuje na wskaźnik instrukcji EIP w następujący sposób:

1. jeśli co najmniej jeden ze znaczników ZF lub CF jest różny od zera (warunek nie jest spełniony) to EIP ← EIP + 2;
2. jeśli jednocześnie oba znaczniki ZF i CF zawierają zera (warunek jest spełniony) to EIP ← EIP + 2 + liczba umieszczona w polu adresowym instrukcji

Nietrudno zauważyć, że instrukcja `ja` może być użyta do sprawdzenia czy zawartość rejestru BX jest większa od zawartości rejestru CX. W takim bowiem przypadku, po obliczeniu różnicy BX–CX znaczniki ZF i CF będą zawierały zera. Zatem warunek testowany przez instrukcję `ja` będzie spełniony – rejestr EIP zostanie zwiększony o liczbę umieszczoną w polu adresowym instrukcji, i jeszcze plus dwa (liczba bajtów instrukcji `ja`). Na poziomie asemblera omawiane sprawdzenie może być zapisane w postaci:

```
        sub bx, cx  ; obliczenie BX←BX-CX
        ja  dalej
        …
dalej: nop
```

W trakcie tłumaczenia podanego fragmentu programu asembler wyznaczy pole adresowe instrukcji `ja` w taki sposób, że jeśli warunek będzie spełniony, to wskaźnik instrukcji EIP zostanie odpowiednio zwiększony tak, by jako następna została wykonana instrukcja opatrzona etykietą dalej. Jeśli warunek nie będzie spełniony, to EIP zostanie zwiększony o 2, co w rezultacie spowoduje wykonanie instrukcji zapisanej bezpośrednio za instrukcją `ja`.

Podany fragment programu można jeszcze ulepszyć. W operacjach porównywania odejmowanie wykonywane jest w celu uzyskania pewnych cech wyniku odejmowania (które wpisywane są do rejestru znaczników). Liczbowy wynik odejmowania zazwyczaj nie jest potrzebny. Z tego powodu konstruktorzy procesora zdefiniowali instrukcję `cmp`, która wykonuje te same operacje co instrukcja odejmowania `sub`, z tą różnicą że nie wpisuje nigdzie wyniku odejmowania (ale cechy tego wyniku wpisuje do rejestru znaczników). Podany fragment można zatem zakodować w podanej niżej postaci:

```
        cmp bx, cx  ; obliczenie BX-CX
        ja  dalej
        …
dalej: nop
```

2.5 Interpretacja pola adresowego instrukcji sterującej

We wcześniejszych rozważaniach podaliśmy formułę opisującą zmiany rejestru EIP w przypadku gdy testowany warunek jest spełniony.

Analizując omawianą formułę trzeba najpierw wyjaśnić postępowanie, w przypadku, gdy w wyniku sumowania powstanie liczba zawierająca 33 bity, a więc liczba, która nie może zostać wpisana do 32-bitowego rejestru EIP. Z tego względu przyjęto zasadę, że powyższe sumowanie wykonywane jest modulo 32, tzn. że z uzyskanego wyniku pozostawia się tylko 32 najmniej znaczące cyfry. Zauważmy dalej, że gdyby podane wyrażenie były wykonywane „dosłownie”, to rejestr EIP mógłby być tylko zwiększany, co nie pozwoliłoby na zbudowanie pętli (której realizacja wymaga zmniejszenia EIP). Z tego przyjęto nieco bardziej skomplikowany schemat dodawania: 8-bitowe pole adresowe rozszerza się do 32 bitów poprzez powielenie najstarszego bitu w lewo. Dopiero tak przekształconą liczbę dodaje się do zawartości EIP. Omówiony schemat wyjaśnia poniższy przykład.

Przed wykonaniem pokazanej niżej instrukcji `ja` rejestr EIP zawierał 0000030AH.

01110111 11111011

Ponieważ testowany warunek był spełniony, więc zawartość pola adresowego została rozszerzona do 32 bitów:

przed powieleniem : 11111011= FBH
po powieleniu: 11111111111111111111111111111011 = FFFFFFFBH

Tak uzyskana liczba i liczba 2 zostaną dodane do zawartości rejestru EIP. Sumowanie to wygodnie przedstawić w zapisie szesnastkowym:

0000030A+FFFFFFFB+00000002=00000307

Z uzyskanej sumy pozostawiamy 32 najmłodsze bity, czyli 00000307H. Tak więc przy odpowiednio dobranej liczbie w polu adresowym można spowodować, że zawartość rejestru EIP zostanie zmniejszona, co umożliwia implementację pętli.

Jednak istotną wadą jednobajtowego pola adresowego instrukcji sterującej jest ograniczony zasięg przekazywania sterowania – można bowiem przekazać sterowanie (gdy warunek jest spełniony) tylko do instrukcji odległej najwyżej o 127 bajtów. W dalszej części instrukcji podamy sposoby na poziomie programowania, które pozwalają przełamać te ograniczenia.

2.6 Instrukcje sterujące warunkowe i bezwarunkowe

W praktyce programowania dość często występują sytuacje, które wymagają bezwarunkowej zmiany porządku wykonywania instrukcji. W tym przypadku stosuje się instrukcje sterujące, które nie testują żadnego warunku i działają tak jak gdyby warunek był zawsze spełniony. Tego rodzaju instrukcje nazywane instrukcjami sterującymi bezwarunkowymi albo instrukcjami skoku bezwarunkowego – instrukcje te oznaczane są mnemonikiem `jmp`.

Obok omówionej wcześniej instrukcji sterującej warunkowej `ja` istnieje wiele innych instrukcji testujących stany znaczników. Poniżej podano krótki ich przegląd.

Mnemonik	Testowany warunek	Zastosowanie
jz(je)	ZF=1	testowanie czy oba argumenty są równe
jnz (jne)	ZF = 0	testowanie czy oba argumenty są różne
ja (jnbe)	ZF = 0 i CF=0	testowanie czy pierwszy argument jest większy od drugiego (liczby bez znaku)
jae (jnb, jnc)	CF = 0	testowanie czy pierwszy argument jest większy lub równy od drugiego (liczby bez znaku)
jb (jc)	CF = 1	testowanie czy pierwszy argument jest mniejszy od drugiego (liczby bez znaku)
jbe (jna)	ZF=1 lub CF = 1	testowanie czy pierwszy argument jest mniejszy lub równy od drugiego (liczby bez znaku)
jg (jnle)	ZF = 0 i SF = OF	testowanie czy pierwszy argument jest większy od drugiego (liczby ze znakiem)
jge (jnl)	SF = OF	testowanie czy pierwszy argument jest większy lub równy od drugiego (liczby, ze znakiem)
jl (jnge)	SF o OF	testowanie czy pierwszy argument jest mniejszy od drugiego (liczby ze znakiem)
jle (jng)	ZF=1 lub SF<>OF	testowanie czy pierwszy argument jest mniejszy lub równy od drugiego (liczby ze znakiem)

Zauważmy, że poszczególne instrukcje sterujące są kodowane za pomocą dwóch lub trzech mnemoników. W zależności od kontekstu programista może wybrać odpowiedni mnemonik, np. można testować czy liczba ze znakiem w rejestrze AL jest większa lub równa od liczby ze znakiem w rejestrze CL (mnemonik `jge`). Można też sprawdzać czy liczba ze znakiem w rejestrze AL jest niemniejsza od liczby ze znakiem w rejestrze CL (mnemonik `jnl`). Oczywiście mnemoniki `jge` i `jnl` są tłumaczone na ten sam ciąg zero-jedynkowy (01111101).

2.7 Przykładowy program

Pokazany poniżej przykładowy program ignoruje znaki wprowadzane z klawiatury z wyjątkiem cyfry „1" lub „2" oraz spacji. W zależności od naciśniętego klawisza wyświetlany jest odpowiedni napis. Naciśnięcie spacji powoduje wydrukowanie na ekran napisu „Koniec programu", i wykonywanie programu zostaje zakończone.

```
dane        SEGMENT ;segment danych
tekst_1     db ": To jest pierwszy tekst", 0dh, 0ah, "$"
tekst_2     db ": To jest drugi tekst", 0dh, 0ah, "$"
tekst_3     db ": Koniec programu", 0dh, 0ah, "$"
nlcr        db    0dh, 0ah, "$"
dane        ENDS

rozkazy     SEGMENT ;segment rozkazu
            ASSUME cs:rozkazy, ds:dane
      startuj:    mov ax, SEG dane
                  mov ds, ax
      czytaj:     mov ah, 01h
                  int 21h           ;czytanie znaku z klawiatury do AL
                  cmp al, "1"
                  jz    jeden ;     ;skok gdy naciśnięto 1
                  cmp al, "2"
                  je    dwa         ;skok gdy nacisnięto 2
                  cmp   al, " "
                  jz    koniec      ;skok gdy nacięto spację
                  mov   dx,offset nlcr
                  mov ah, 09h
                  int 21h
                  jmp czytaj
      jeden:      mov dx, offset tekst_1
                  mov ah, 09h
                  int 21h
                  jmp czytaj
      dwa:        mov dx, offset tekst_2
                  mov ah, 09h
                  int 21h
                  jmp czytaj
      koniec:     mov dx, offset tekst_3
                  mov ah, 09h
                  int 21h
                  mov al, 0
                  mov ah, 4CH
                  int 21H
rozkazy ENDS

stosik SEGMENT stack
      dw    128 dup(?)
stosik ENDS

END   startuj
```