Laboratorium 5. Switch – Case - Break

Konstrukcja warunkowa „*if – else_if – else*”, np.:

```
if (a==b)                    //warunek-wyrażenie 1
   {
       blok kodu 1
    }
else if (a==c)               //warunek-wyrażenie 2
   {
       blok kodu 2
    }
else if (a==d)               //warunek-wyrażenie 3
    {
       blok kodu 3
    }
else
   {kod ”alternatywny”}

//end_if
```

może zostać przetłumaczona na ciąg instrukcji testujących (porównujących, arytmetycznych, logicznych…) oraz zestawu skoków, do odpowiadających początkom poszczególnych bloków kodu adresów (etykiet).

Jedna z możliwych implementacji wygląda tak:

```
cmp    b , a          # porównaj a z b (np. rejestry)
jne    wariant2       # jeśli warunek 1 nie spełniony – idź dalej testować kolejny
#blok kodu 1          # jeśli warunek 1 jest jednak spełniony, wykonaj kod 1
jmp    end_if         # i przeskocz na koniec (end_if).

wariant2:
cmp    c , a          # jeśli warunek 2 nie był spełniony – to testuj kolejny
jne    wariant3
#blok kodu 2
jmp    end_if

wariant3:
cmp    d , a          # jeśli warunek 3 nie był spełniony – przejdź do sekcji else
jne    else
#blok kodu 3
jmp    end_if

else:
#kod “alternatywny”

end_if:
```

Rozwiązanie takie jest elastyczne i wyrażenia warunkowe mogą mieć złożoną postać (np. zawierać wiele operacji arytmetycznych i logicznych).

W pewnych zastosowaniach posiada jednak wadę: czas wykonania nie jest stały i zależy od liczby przetestowanych warunków, niezbędnych do „dotarcia” do właściwej części kodu (liczby instrukcji wykonanych przez CPU i zużytych cykli zegarowych).

Najbardziej „pesymistyczny” wariant w powyższym układzie – dotarcie do sekcji *else* – wymaga przetestowania wszystkich warunków.

W przypadku, gdy wybór fragmentu kodu do wykonania (sterowanie przebiegiem programu) jest uzależniony od **tylko od konkretnych wartości jednej zmiennej**, kompilator może dokonać pewnej optymalizacji np. zamiast testować kolejne przypadki w porządku liniowym (np. rosnąco), zacząć od wartości „środkowej” (i dalszego połowienia podzakresów), itp.

Jeżeli wartości zmiennej sterującej, decydujące o wyborze konkretnego przypadku nie są zbyt oddalone od siebie*, możliwa jest inna, niskopoziomowa, realizacja – często szybsza i posiadająca (prawie) stały czas wykonania.

Konstrukcję typu *switch*:

```
switch(zmienna)
   {
       case 2: {blok kodu 2; break;**}
       case 3: {blok kodu 3; break;}
       case 8: {blok kodu 8; break;}
       case 10: {blok kodu 10; break;}
       …
       default: {alternatywny blok kodu;}
```

** bez *break* zostanie wykonana kolejna sekcja…

można oprzeć na tzw. tablicy skoków (*jump table*, *lookup table*), która przechowuje adresy początków odpowiednich bloków kodu (tzn. element tablicy o indeksie *n* zawiera adres kodu dla *n*–tego przypadku).

Powyższy *switch* zapisany w asemblerze:

```
.data
jump_table:    .quad   case2, case3, default, default, default, default, case8, default, case10

.text
main:                  #założenie: wartość zmiennej sterującej w %eax

cmp     $2 , %eax      #jeśli mniejsze niż minimalny przypadek w tablicy skocz do „default”
jb      default

cmp     $10 , %eax     #analogicznie sprawdź drugi koniec
ja      default

jmp     *jump_table-16( ,%eax,8)      #skok pośredni, pod adres odczytany z tablicy.
                                      #Z tablicy odczytywana jest liczba (8 bajtów)
                                      #zaczynająca się pod adresem=(jump_table-16)+%eax*8
case2:
#blok kodu 2
jmp     end_switch                    #czyli ”break”

case3:
#blok kodu 3
jmp     end_switch

case8:
#blok kodu 8
jmp     end_switch

case10:
#blok kodu 10
jmp     end_switch

default:
#”alternatywny” blok kodu

end_switch:
```

Kilka uwag:

1. Elementy tablicy skoków są 8–bajtowe – programy w przypadku współczesnego PC i 64–bitowego systemu operacyjnego pracują w dużej, wirtualnej przestrzeni adresowej (w *x86-64* z reguły 48–bitowej).

2. Elementy tablicy są indeksowane od zera. W powyższym przykładzie, element o indeksie zero zawiera adres przypadku 2 (i dalej: el. o indeksie 1 – adres przypadku 3) stąd konieczność odjęcia 16 bajtów (przesunięcie o dwa elementy 8–bajtowe bajtowe) podczas obliczania adresu skoku.

3. W znakomitej większości przypadków tworzy się fragmentaryczną tablicę skoków, obejmującą przedział od wartości minimalnej do maksymalnej, jakie są używane w *switch-case*.

Wymusza to niestety konieczność użycia dwóch porównań, (jeśli mniejsze od *min* albo większe od *max* skocz od razu do części *default*), ale zaoszczędza miejsce w pamięci danych (i *cache*), niezbędne na przechowywanie tablicy.

4. W sytuacjach, gdy najmniejsza wartość przypadku *switcha* leży blisko zera, tablicę skoków można rozpocząć od przypadku zerowego (ten sam przykład co wyżej):

```
.data
jump_table:    .quad  default, default case2, case3, default, default, default, default,
case8, default, case10
```

odpada więc jedno porównanie (i przesunięcie przy obliczaniu adresu), a kod i czas wykonania (kosztem większej tablicy) skraca się:

```
cmp     $10 , %eax
ja      default

jmp     *jump_table( ,%eax,8)
```

W szczególnym, krytycznym czasowo przypadku, gdy *switcha* da się sterować zmienną typu char (bajt) można utworzyć pełną tablicę. Np. 256 elementów z 32–bitowymi, młodszymi częściami adresów (lub adresowaniem względnym) zajmie jeden kilobajt. Żadnych porównań wtedy nie ma, a czas wyboru dowolnego fragmentu kodu jest stały.

* Metoda ma zastosowanie, gdy wartości poszczególnych przypadków „leżą” stosunkowo blisko siebie. W sytuacji gdy kolejne warianty są bardzo oddalone (np. 10, 100, 5000…), stworzenie tablicy skoków może być nieekonomiczne.

Współczesne kompilatory w zależności od liczby przypadków, ich „rozpiętości” oraz wybranych kryteriów optymalizacji kodu same decydują czy *switcha* tablicować, czy tłumaczyć jak standardowy *if–else_if–else*. Czasami stosowana jest kombinacja obu technik: grupy wartości niewiele się różniących od siebie są tablicowane, podczas gdy przypadki odległe sprawdzane są indywidualnie.

Zadanie:

Napisać program (uzupełnić plik *swcs.s*), który w zależności od wybranego numeru działania wykona jedną z operacji logicznych (arytmetycznych) na dwóch liczbach (32–bitowych) i wyświetli jej wynik.

Liczby, jak i numer działania, należy przekazać jako parametry z linii komend (opis w pliku *parametry.pdf*):

```
./nazwa_programu  liczba_1  liczba_2  nr_dzialania
```

Przykładowe numery działań:

3 - AND
5 - OR
6 - XOR
10 – ADD
14 – SUB

Dla pozostałych wartości program ma wyświetlić np. „BLAD”. W przypadku braku, lub podania złej liczby argumentów (!=3) powinien również zostać zgłoszony odpowiedni komunikat.

Np. wywołanie: `./prog4  64  128  5`

powinno skutkować wyświetleniem: 64 OR 128 = 192

Analogicznie dla pozostałych operacji.

W zadaniu należy wykorzystać tablicę skoków (*switch-case-break*).

W programie powinna zostać uwzględniona (przynajmniej minimalna) obsługa błędów. Np. po wywołaniu programu bez parametrów, bądź z podaniem nieprawidłowej ich liczby – powinno nastąpić wyjście poprzedzone stosownym komunikatem.

Liczby odpowiadające kolejnym działaniom zostaną podane przez prowadzącego zajęcia.