lab_6 - Instrukcja do ćwiczenia

Teoria:

http://galaxy.agh.edu.pl/~amrozek/AK/lab_5.pdf

Wariantowa realizacja kodu:

Celem ćwiczenia jest zrozumienie zasad wariantowej realizacji kodu. Programy **lab_6a.s** i **lab_6b.s** wymagają podania przy uruchomieniu trzech liczb – dwie pierwsze to argumenty na których należy wykonać pewną operację, zaś trzecia stanowi kod operacji. Niewłaściwa liczba argumentów lub niewłaściwa wartość kodu operacji są sygnalizowane wyświetleniem odpowiedniego komunikatu.

Wykorzystane zostały następujące kody operacji:

- AND – kod 3
- OR – kod 5
- XOR – kod 6
- ADD – kod 10
- SUB – kod 15

Zagnieżdżone instrukcje if … else:

W programie **lab_6a.s** wykorzystane zostały zagnieżdżone instrukcje **if … else**. Struktura zasadniczej części programu ma następującą postać:

```
if( kod == 3 ) …
else if( kod == 5 ) …
else if( kod == 6 ) …
else if( kod == 10 ) …
else if( kod == 15 ) …
else …
```

Zaletą takiego podejścia jest prosta realizacja, wadą względna czasochłonność (dla znacznej liczby wariantów) – dla kodu = **15** konieczne jest sprawdzenie wszystkich wcześniejszych warunków.

Bezwarunkowy skok pośredni:

W programie **lab_6b.s** wykorzystana została instrukcja bezwarunkowego skoku pośredniego w połączeniu z tablicą skoków – to często wykorzystywany sposób implementacji instrukcji **switch**. Struktura zasadniczej części programu ma następującą postać:

```
switch( kod ) {
     case 3: …; break;
     case 5: …; break;
     case 6: …; break;
     case 10: …; break;
     case 15: …; break;
     default: …
}
```

Zaletą takiego podejścia jest krótki (i w miarę stały) czas realizacji każdego wariantu, wadą konieczność budowy tablicy skoków i jej szybko rosnące rozmiary (dla wariantów znacznie różniących się od siebie).

Aby dało się w sensowny sposób korzystać z tablicy skoków, musi ona zawierać co najmniej tyle elementów ile wynosi różnica pomiędzy maksymalną i minimalną wartością wariantu powiększona o **1** – dla zdefiniowanych wcześniej kodów wielkość tablicy to **15 – 3 + 1 = 13** elementów. Zwiększenie rozmiaru tablicy tak, aby indeks ostatniego elementu był równy maksymalnej wartości wariantu pozwala na uproszczenie kodu – w rozważanym przypadku tablica skoków będzie mieć 16 elementów (o indeksach **0..15**).

Tablicę należy wypełnić adresami tych fragmentów kodu, które realizują określoną operację – w puste miejsca (tablica ma **16** elementów a operacji jest **5**) należy wstawić adres fragmentu kodu odpowiedzialnego za wykonanie akcji zapisanej dla przypadku **default**.

Przygotowana tablica skoków pozwala na przejście do odpowiedniego fragmentu kodu przy pomocy instrukcji bezwarunkowego skoku pośredniego (bo adres miejsca, gdzie należy wykonać skok jest zawarty w tablicy, a w odwołaniu wykorzystywany jest indeks elementu tablicy (wybrany wariant)). Zakładając, że kod operacji znajduje się w rejestrze **%rax**, stosowny kod wygląda następująco:

```
cmp $0, %rax             # check for number of operation
jl oper_err              # < 0, so error
cmp $15, %eax            # check for number of operation
jg oper_err              # > 15, so error
jmp *jump_tab(,%rax,8)   # jump to code of operation
```

Instrukcje porównań i skoków warunkowych służą do upewnienia się że w instrukcji bezwarunkowego skoku pośredniego wykorzystany zostanie jeden ze zdefiniowanych (w tablicy skoków) adresów – każdy adres to 8 bajtów.

Wyświetlanie danych:

Liczby na których wykonywane są wybrane operacje oraz rezultaty tych operacji są wyświetlane zarówno w postaci dziesiętnej jak i szesnastkowej. Wyświetlana jest też nazwa operacji. Ponieważ przy wywołaniu funkcji **printf** (i innych) tylko sześć argumentów może być

przekazanych w sposób szybki i prosty (w rejestrach), a ich łączna liczba jest większa, więc dane są wyświetlane przy pomocy dwóch wywołań:

printf(fmt1, arg1, arg2, operation_name, arg2, arg2)

printf(fmt2, result, result)

Praktyka (lab_6.s, lab_6b, test_6.c):

Działania:

1. Testujemy działąnie programu lab_6a.
2. CL (Compile&Link) – polecenie: **gcc –no-pie –o lab_6a lab_6a.s**
3. R (Run) – polecenie: **./lab_6a**
4. Przykladowe efekty uzyskane po uruchomieniu wyglądają następująco:

```
buba@buba-pc:~/asm/l6$ ./lab_6a
No or bad numbers!

buba@buba-pc:~/asm/l6$ ./lab_6a  15  7  1
Bad operation!
```

5. Podanie właściwych kodów operacji skutkuje poprawnym działaniem programu. Przykładowe rezultaty mogą wyglądać następująco:

```
buba@buba-pc:~/asm/l6$ ./lab_6a  15  7  3
15 (0x0000000f) AND 7 (0x00000007) = 7 (0x00000007)

buba@buba-pc:~/asm/l6$ ./lab_6a  15  7  5
15 (0x0000000f) OR 7 (0x00000007) = 15 (0x0000000f)

buba@buba-pc:~/asm/l6$ ./lab_6a  15  7  6
15 (0x0000000f) XOR 7 (0x00000007) = 8 (0x00000008)

buba@buba-pc:~/asm/l6$ ./lab_6a  15  7  10
15 (0x0000000f) ADD 7 (0x00000007) = 22 (0x00000016)

buba@buba-pc:~/asm/l6$ ./lab_6a  15  7  15
15 (0x0000000f) SUB 7 (0x00000007) = 8 (0x00000008)
```

6. Wszystko działa zgodnie z oczekiwaniami.
7. Przechodzimy do programu **lab_6b**.
8. CL (Compile&Link) – polecenie: **gcc –no-pie –o lab_6b lab_6b.s**
9. R (Run) – polecenie: **./lab_6b**
10. Przykladowe efekty uzyskane po uruchomieniu wyglądają następująco:

```
buba@buba-pc:~/asm/l6$ ./lab_6b
No or bad numbers!

buba@buba-pc:~/asm/l6$ ./lab_6b  15  7  1
Bad operation!
```

11. Podanie właściwych kodów operacji skutkuje poprawnym działaniem programu. Przykładowe rezultaty mogą wyglądać następująco:

```
buba@buba-pc:~/asm/l6$ ./lab_6b  15  7  3
15 (0x0000000f) AND 7 (0x00000007) = 7 (0x00000007)

buba@buba-pc:~/asm/l6$ ./lab_6b  15  7  5
15 (0x0000000f) OR 7 (0x00000007) = 15 (0x0000000f)

buba@buba-pc:~/asm/l6$ ./lab_6b  15  7  6
15 (0x0000000f) XOR 7 (0x00000007) = 8 (0x00000008)

buba@buba-pc:~/asm/l6$ ./lab_6b  15  7  10
15 (0x0000000f) ADD 7 (0x00000007) = 22 (0x00000016)

buba@buba-pc:~/asm/l6$ ./lab_6b  15  7  15
15 (0x0000000f) SUB 7 (0x00000007) = 8 (0x00000008)
```

12. Ponownie wszystko działa zgodnie z oczekiwaniami.
13. Przechodzimy do programu **test_6**.
14. Program napisany jest w języku **C** – użyta w nim funkcja **func** zawiera instrukcję **switch**, a celem tej części ćwiczenia jest sprawdzenie, jaki kod zostanie wygenerowany przez kompilator dla różnych postaci tej instrukcji.
15. Ponieważ nie zależy nam na uruchomieniu programu, a jedynie na wglądzie w wynik kompilacji, więc używamy następującego polecenia:

```
gcc -S test_6.c
```

16. Pojawił się plik **test_6.s**, który otwieramy w edytorze.
17. Lokalizujemy kod funkcji **func** (szukamy etykiety **func:**).
18. Mniej więcej w połowie kodu funkcji wygenerowanego przez kompilator możemy znaleźć instrukcję:

```
jmp  *%rax
```

19. Oznacza to, że kompilator wybrał identyczne podejście jak zastosowane w kodzie programu **lab_6b** – bezwarunkowy skok pośredni. Inna forma skoku wynika z tego, że zawartość odpowiedniego elementu tablicy skoków (w wygenerowanym kodzie to etykieta **.L4**) została wcześniej przeniesiona do rejestru **%rax**.
20. W programie **test_6.c** zmieniamy zawartość instrukcji **switch** – po modyfikacji zawartość powinna być następująca:

```
int func( int x )    {
      switch( x )    {
           case  0: return 0; break;
           case  100: return 1; break;
           case  200: return 2; break;
           case  300: return 3; break;
           case  400: return 4; break;
           default: return -1;
       }
       return 0;
}
```

21. Ponownie zmuszamy kompilator do pracy:

```
gcc -S test_6.c
```

22. Ponownie otwieramy plik **test_6.s** w edytorze.
23. Ponownie lokalizujemy kod funkcji **func**.
24. Tym razem w kodzie nie znajdziemy instrukcji bezwarunkowego skoku pośredniego – kompilator uznał, że w tym przypadku tworzenie tablicy skoków będzie nieefektywne (ze względu na jej wielkość) i zapisał kod instrukcji **switch** jako sekwencję zagnieżdżonych instrukcji **if … else** – czyli wybrał identyczne podejście jak zastosowane w kodzie programu **lab_6a**.
25. Przeprowadź więcej prób zmieniając wartości w instrukcji **switch** – spróbuj ustalić, kiedy kompilator zmieni sposób realizacji tej instrukcji.