Ćwiczenie 4

Wywoływanie i powrót z podprogramów z wykorzystaniem stosu - zasady

UWAGA: Dla celów dydaktycznych należy ściśle przestrzegać konwencji opisanych w instrukcji poniżej. Za odstępstwa będą odejmowane punkty. Dla uproszczenia, niekiedy odbiegają one od konwencji stosowanych w architekturze MIPS.

Celem zadania jest oprogramowanie mechanizm wywoływania i powrotu z funkcji/procedur (ogólnie zwanymi dalej podprogramami). Mechanizm powinien wykorzystywać stos zorganizowany we własnym obszarze pamięci (nie wykorzystujemy stosu systemowego). Zakładamy że adres wierzchołka stosu znajduje się w rejestrze `$sp` i rejestr ten nie jest w żaden inny sposób wykorzystywany niż tylko do przechowywania podczas całego działania programu aktualnego adresu wierzchołka stosu. Obszar na stos rezerwujemy w sekcji `.data`

```
#=============================================
.eqv STACK_SIZE 2048

#=============================================
.data

# obszar na zapamiętanie adresu stosu systemowego
sys_stack_addr:  .word  0

# deklaracja własnego obszaru stosu
stack:           .space STACK_SIZE


# =============================================
.text

# czynności inicjalizacyjne
    sw  $sp,  sys_stack_addr   # zachowanie adresu stosu systemowego
    la  $sp,  stack+STACK_SIZE # zainicjowanie obszaru stosu


# początek programu programisty - zakładamy, że main
# wywoływany jest tylko raz
main:
    # ciało programu . . .

    # koniec podprogramu  main:
    lw  $sp,  sys_stack_addr  # odtworzenie wskaźnika stosu
                              # systemowego
    li  $v0,  10
    syscall
```

Zgodnie ze stosowaną konwencją zakładamy, że stos rozrasta się w stronę malejących adresów. Dlatego program powinien rozpocząć się od zainicjowania wskaźnika stosu - jak pokazano w kodzie powyżej. Dodatkowo przyjmijmy, że `$sp` jest adresem bajtu znajdującego się na szczycie stosu.

Jeśli np. adres obszaru `stack` jest 100 a rozmiar stosu `STACK_SIZE` jest 150 to początkową wartość `$sp` inicjujemy na 100 + STACK_SIZE = 250. Jeśli położymy na stos słowo 4-bajtowe to zmniejszamy $sp = rozmiar słowa (4) i otrzymujemy $sp = 246. Jest to adres pierwszego bajtu słowa położonego na stos. Jeśli chcemy pobrać to słowo do rejestru to stosujemy np. rozkaz:

```
lw $t1, ($sp)
```

Jeśli teraz położymy na stos kolejne słowo to po raz drugi zmniejszamy `$sp` o 4. Teraz `$sp=242`.

Teraz rozkaz:

```
lw $t1, ($sp)
```

pobierze do rejestru drugi z położonych na stos słów. Jeśli teraz chcemy pobrać pierwsze ze słów położonych na stosie to należy wykonać

```
lw $t1, 4($sp)
```

UWAGA: powyższe operacje pobierają dane z wnętrza stosu i nie są równoważne typowej operacji pop.

Wszystkie dane związane z wywołaniem podprogramu, włączając w to wszystkie jego zmienne lokalne należy umieścić w bloku pamięci na stosie. Blok ten jest zwykle nazywany ***rekordem aktywacji*** podprogramu.

Aby zarezerwować miejsce na stosie (np. na zmienne lokalne podprogramu) wystarczy zmniejszyć adres w `$sp` o rozmiar alokowanego obszaru. Np. jeśli chcemy zarezerwować na stosie miejsce dla dwóch zmiennych lokalnych typu `int` to wystarczy wykonać operację:

```
    subi $sp, $sp, 8  # 8 = 2 * sizeof( int )
```

Wewnątrz kodu podprogramu `$sp` nie zmienia się. Można zatem odwoływać się do zmiennych lokalnych stosując adresowanie rejestrowe pośrednie z przesunięciem względem rejestru `$sp`. (w literaturze anglojęzycznej - *base addressing*) Rozważmy następującą funkcję:

```
int f()
{
   int i;
   int j;
   i = 10;
   j = 20
   i = i + j;
   return j;
}
```

Załóżmy, że przydzielamy miejsca na stosie kolejno definiowanym zmiennym tak, że ostatnia zmienna (wg kolejności deklarowania) znajduje pod adresem w `$sp`. Wtedy w powyższym przykładzie po zaalokowaniu zmiennych w rekordzie aktywacji:

adres zmiennej `j`: `$sp`

adres zmiennej `i`: `$sp+4`

Kod odpowiadający instrukcji `j = i + j` wyglądałby następująco

```
lw $t1,  ($sp)   # ładujemy zmienną j
lw $t2,  4($sp)  # ładujemy zmienną i - kompilator może na
                 # etapie kompilacji wyznaczyć offset 4 dla
                 # zmiennej i ponieważ zna rozlokowanie
                 # zmiennych lokalnych w rekordzie aktywacji
add $t1, $t1, $t2
sw  $t1, 4($sp)  # zapisujemy obliczoną sumę do zmiennej w
                 # rekordzie aktywacji
```

Przyjmij następujący schemat wywoływania i powrotu z funkcji:

Wywołanie - strona wywołująca:

- umieszcza na stosie kolejne argumenty funkcji - od lewej do prawej (UWAGA: nie wykorzystujemy w tym celu rejestrów)
- wywołuje funkcję rozkazem

```
jal  <adres_początku kodu podprogramu>
```

Wywołanie - strona wywołana:

- przesuwa adres wierzchołka stosu tak aby zrobić miejsce na wartość zwracaną (UWAGA: nie wykorzystujemy w tym celu rejestrów)
- umieszcza na wierzchołku stosu adres powrotu z rejestru `$ra`
- przesuwa adres wierzchołka stosu tak aby zrobić miejsce na zmienne lokalne (zakładamy, że zmienne lokalne będą przechowywane na stosie

Powrót - strona wywołana

- zapisuje na stosie (w miejscu poprzednio zarezerwowanym do tego celu) wartość zwracaną
- przesuwa wskaźnik stosu tak aby usunąć ze stosu zmienne lokalne
- pobiera ze stosu adres powrotu i umieszcza go w rejestrze `$ra`
- przesuwa wskaźnik stosu tak aby na wierzchołku stosu znalazła się wartość zwracana
- wykonuje powrót rozkazem

```
jr  $ra
```

Powrót - strona wywołująca:

- pobiera ze stosu wartość zwróconą przez wywołany podprogram
- przesuwa wskaźnik stosu tak aby usunąć z niego wartość zwracaną oraz argumenty podprogramu położone na stos przed jego wywołaniem

Po wykonaniu tych czynności wskaźnik stosu musi mieć taką samą wartość jak przed rozpoczęciem wywoływania podprogramu (tzn. przed położeniem na stos argumentów podprogramu)

Zadanie do wykonania

Stosując podane powyżej zasady zaimplementuj kod odpowiadający następującemu programowi:

```
int global_array[10] = { 1,2,3,4,5,6,7,8,9,10 };

int sum ( int *array, int array_size )
     // notacja int *array oznacza przekazanie do funkcji
     // adresu początku tablicy array
{
   int i;
   int s;

   s = 0;
   i = array_size - 1;
   while ( i >= 0 ) {
      s = s + array[i];
      i = i - 1;
   }
   return s;
}

void main()
{
   int s;

   s = sum( global_array, 10 );
   print( s );
   return;
}
```

Zastosuj dokładnie opisane zasady konstrukcji kodu po stronie wywołującej i wywoływanej.

UWAGA: nie należy stosować żadnych optymalizacji, w szczególności przyjąć założenie, że każda instrukcja "kompilowana" jest niezależnie od kontekstu, zatem w szczególności operacja podstawienie zawsze musi skutkować rzeczywistym zapisem do pamięci (optymalizacja pozwalałaby tego uniknąć I przechowywać tymczasowe wartości zmiennych tylko w rejestrach - ze względu na prostotę kodu nie należy tutaj tego robić.

Literatura:

Patterson D.A., Hennessy J.L.: Computer Organization and Design, 4th ed.