Architektura komputerów – laboratorium 13 (2 godziny)
Temat: Wywoływanie funkcji i operacje na stosie

Stos to ciągły obszar pamięci wykorzystywany do tymczasowego przechowywania zmiennych. Z punktu widzenia struktur danych stos jest kolejką LIFO (Last In First Out). Stos jest strukturą dynamiczną. W zależności od aktualnych potrzeb zajmuje więcej lub mniej pamięci. Ważne jest, by zarezerwować na stos odpowiednią ilość pamięci. Wierzchołek stosu znajduje się w pamięci o adresie mniejszym niż jego dno – stos rośnie w stronę malejących wartości adresów. W asemblerze MIPS stos opisany jest trzema podstawowymi parametrami:

- Najniższy poprawny adres stosu – wierzchołek stosu (ang. Stack limit / origin)
- Wskaźnik na wierzchołek stosu (ang. Stack pointer)
- Najwyższy poprawny adres stosu – dno stosu (ang. Stack bottom)

Do przechowywania wskaźnika do wierzchołka stosu zwyczajowo wykorzystuje się rejestr $sp ($29), chociaż może to być dowolny rejestr ogólnego przeznaczenia. W niektórych sytuacjach (gdy przydzielimy zbyt mało pamięci na stos) może się zdarzyć, że wskaźnik na wierzchołek stosu wskazuje mniejszy adres niż najniższy poprawny adres stosu. Wówczas mamy do czynienia z przepełnieniem stosu (stack overflow). Poniższy ciąg instrukcji przedstawia sposób odkładania i pobierania zawartości rejestru na i ze stosu:

```
                addi $t1, $0, 7      #wpisz wartość 7 do rejestru $t1
na_stos:        addi $sp, $sp, -4    #pomniejsz zawartość wskaźnika stosu $sp o 4, tj.
                                     #zrób miejsce na wpisanie słowa (4 bajtów) na stos
                sw $t1, 0(#sp)       #odłóż zawartość $t1 na stos

ze_stosu:       lw $t2, 0($sp)       #załaduj zawartość wierzchołka stosu do $t2
                addi $sp, $sp, 4     #zwiększ zawartość wskaźnika stosu $sp o 4, tj.
                                     #zwolnij miejsce na stosie po zdjętym słowie
```

Na ogół zachodzi potrzeba odłożenia na stosie wielu danych, np. argumentów funkcji i zwracanych wyników. Wówczas obszar pamięci zajętej przez stos dzielony jest na ramki (ang. Stack frames), do których można się odwoływać wykorzystując dodatkowy wskaźnik, tzw. wskaźnik ramki (ang. Frame pointer) $fp. Wskaźnik ramki $fp przechowuje wartość wskaźnika stosu $sp przed jego zmianą.

Stos pełni ważną rolę w wywoływaniu funkcji, a w szczególności w wywoływaniu funkcji rekurencyjnych. Architektura MIPS oferuje dwie instrukcje przeznaczone do obsługi wywołania funkcji:

- instrukcję jal, która określa operację przekazania sterowania do funkcji (ze śladem); instrukcja ta nakazuje, aby procesor, zanim przekaże sterowanie do wskazanej funkcji (wykonując operację skoku), zapamiętał w rejestrze $ra adres instrukcji, występującej w programie tuż za instrukcją jal (czyli adres powrotu z funkcji do miejsca wywołania).
- instrukcję jr, określającą operację powrotu z funkcji do programu, który ją wywołał; instrukcja ta, faktycznie, nakazuje procesorowi wykonanie operacji skoku bezwarunkowego pod adres, który aktualnie jest w rejestrze $ra (return address).

Ogólnie procedura wywoływania funcji przedstawia się następująco:

- Wywołujący (ang. Caller):
  - przekazuje argumenty do wywoływanej funkcji (ang. Callee)
  - wykonuje skok do wywoływanej funkcji (callee)
- Wywoływana funkcja (Callee):
  - wykonuje funkcje
  - zwraca wynik do wywołującego (caller)
  - Wraca do punktu wywołania
  - nie może nadpisywać rejestrów albo pamięci wykorzystywanych przez wywołującego (caller)

Należy przy tym przestrzegać poniższych reguł:

- Wywołujący funkcję powinien mieć możliwość przekazania jej listy argumentów (kolejność może być istotna). W ciele funkcji, dostęp do argumentów powinien być taki, jak do zmiennych lokalnych.
- Funkcja powinna mieć możliwość tworzenia własnych zmiennych.
- Wywołujący funkcję (ang. caller) i wywoływana funkcja (ang. callee) muszą przestrzegać uzgodnień w zakresie korzystania z rejestrów procesora, aby zapobiec kolizjom, które mogłyby doprowadzić do utraty danych zawartych w rejestrach
- Funkcja musi mieć możliwość zwrócenia (wywołującemu) wartości, będącej rezultatem jej działania.
- W przypadku algorytmów rekurencyjnych funkcja musi mieć możliwość wywołania samej siebie bez ograniczania głębokości zagnieżdżenia.

W architekturze MIPS, standardowo, do przekazywania argumentów do funkcji, stosuje się rejestry $a0, $a1, $a2, $a3 – jeśli trzeba przekazać więcej argumentów, to należy je przekazać poprzez stos.
Rezultat powinien być, standardowo, zwracany w rejestrach $v0 i $v1. Wartość typu doubleword jest zwracana w parze rejestrów.

Należy zwrócić uwagę na to, że za zachowanie stanu rejestrów odpowiadają zarówno wywołujący, jak i wywoływany, choć każdy w nieco innym zakresie.

- Wywołujący przed wywołaniem funkcji musi zachować stan rejestrów od $t0 do $t7 jeśli ich wartość jest istotna i nie może być zmieniona przez funkcję oraz stan rejestrów $a0, $a1, $a2, $a3 i $v0, $v1
- Wywoływana funkcji musi zachować stan tych spośród rejestrów $s0 .. $s7, $fp, $ra, które mają być używane przez tę funkcję, w celu przechowywania w nich nowych wartości.

Wywołanie funkcji oraz wykorzystanie stosu do przechowywania tymczasowych wartości rejestrów ilustruje przedstawiony poniżej fragment programu obliczający wartość wyrażenia (różnicy sum) $y = (f + g) - (h + j)$, dla $f = 2, g = 3, h = 4, j = 5$.

```
#Program różnica sum
# $s0 = y
main:
 ...
                addi $a0, $0, 2        # argument 0 = 2
                addi $a1, $0, 3        # argument 1 = 3
                addi $a2, $0, 4        # argument 2 = 4
                addi $a3, $0, 5        # argument 3 = 5
                jal  diff_of_sums      # wywołaj funkcję
                add  $s0, $v0, $0      # y = wartość zwrócona przez funkcję
 ...
diffofsums:
                addi $sp, $sp, -12     # przydziel pamięć dla stosu do przechowywaania
                                       #zawartości 3 rejestrów
                sw   $s0, 8($sp)       # zapisz $s0 na stosie
                sw   $t0, 4($sp)       # zapisz $t0 na stosie
                sw   $t1, 0($sp)       # zapisz $t1 na stosie
                add  $t0, $a0, $a1     # oblicz wartość wyrażenia $t0 = f + g
                add  $t1, $a2, $a3     # oblicz wartość wyrażenia $t1 = h + i
                sub  $s0, $t0, $t1     # oblicz różnicę wynik = (f + g) - (h + i)
                add  $v0, $s0, $0      # wstaw wartość wynikową do $v0
                lw   $t1, 0($sp)       # pobież $t1 ze stosu
                lw   $t0, 4($sp)       # pobież $t0 ze stosu
                lw   $s0, 8($sp)       # pobież $s0 ze stosu
                addi $sp, $sp, 12      # zwolnij pamięć przydzieloną na stos
                jr   $ra               # wróć w miejsce wywołującego (caller)
```

Ćwiczenia: Implementacja stosu

Proszę zaimplementować prosty parser (analizator syntaktyczny) języka asembler. Program ma pobrać od użytkownika (wczytać z wejścia) od 1 do maksymalnie 5 instrukcji **ze zbioru podanego poniżej.** Dla każdej instrukcji należy sprawdzić jej poprawność oraz typ i zgodność podanych parametrów. Jeśli instrukcja jest poprawna, to odłożyć ją (wraz z parametrami) na stos. Jeśli instrukcja nie jest poprawna program prosi o jej ponowne podanie (do skutku). Na każdy obiekt proszę użyć jednego słowa, np. instrukcja ADD $8, $9, $10 będzie wymagała użycia 4 słów.

Program ma odkładać na stosie instrukcje w następujący sposób (OST = 1, 2, 3, 4 lub 5):

```
INSTRUKCJAOST
PARAMETR1 INSTRUKCJIOST
PARAMETR2 INSTRUKCJIOST
----------
   ....
----------
INSTRUKCJA2
PARAMETR1 INSTRUKCJI2
```

PARAMETR2 INSTRUKCJI2

INSTRUKCJA1
PARAMETR1 INSTRUKCJI1
PARAMETR2 INSTRUKCJI1
PARAMETR3 INSTRUKCJI1
---------- (stack bottom)

Po ostatniej instrukcji program drukuje instrukcje na wyjściu w odwrotnej kolejności (ściągając je kolejno ze stosu i przy okazji zmieniając wartość wskaźnika stosu - $sp).

Uwaga: PROGRAM NIE WYKONUJE PODANEGO KODU, tylko weryfikuje instrukcje i jej parametry, a następnie odkłada je na stos.

Dane wejściowe:

- liczba instrukcji do przeanalizowania (od 1 do 5)
- kolejno instrukcje w pełnej postaci, :np. "ADD $8, $9, $10" - rejestry podawane są tylko w postaci numerów; aplikacja weryfikuje czy takie rejestry istnieją (p. slajdy z wykładu); UWAGA: liczba parametrów może być różna dla różnych instrukcji

Dane wyjściowe:

- instrukcje, które zostały wcześniej wprowadzone, wydrukowane w odwrotnej kolejności

Zbiór obsługiwanych instrukcji:

ADD $r1, $r2, $r3,
ADDI $r1, $r2, wartość
J label
NOOP
MULT $s, $t
JR $r1
JAL label

Zadanie jest za 8 punktów. Na 10 punktów należy program rozszerzyć o dodatkowy element, tj. na koniec aplikacja liczy ile pamięci zaalokowano na stosie.