MIPS
Funciones

Funciones

- ***Diseño de funciones.***
  - ***Uso de instrucción jal y retorno de subrutina.***
    - Se suelen denominar funciones, procedimientos, subrutinas, subprogramas a las estructuras que emplean los lenguajes de programación para introducir mecanismos de abstracción en la codificación.
    - En el caso de programación assembler suele llamárselas subrutinas; en lenguajes de alto nivel se denominan funciones (abstracción de una expresión) y procedimientos (abstracción de acción).

Funciones

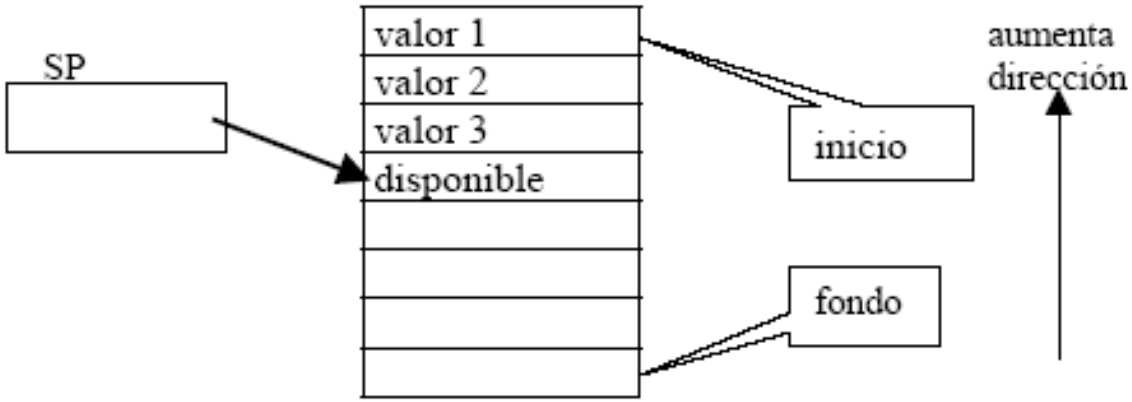
- ***Diseño de funciones.***
  - ***Uso de instrucción jal y retorno de subrutina.***
    - Puede detectarse la necesidad de una función, si existen segmentos que se repiten en el programa.
    - Si se define una función con la secuencia que se repite, puede lograrse un ahorro del tamaño del segmento de texto, ya que se escribe sólo una vez la secuencia, y se colocan llamados o **invocaciones** en los puntos donde ésta aparece.
    - Además debe agregarse, al fin de la secuencia otra instrucción que permita regresar, o **retornar**, al punto desde donde se la invocó.
    - De lo anterior se desprende la necesidad de dos instrucciones adicionales (en nivel de máquina), una para invocar o llamar a la subrutina y otra para retornar desde ella.

Funciones

- ***Diseño de funciones.***
  - ***Uso de instrucción jal y retorno de subrutina.***
    - En el procesador MIPS, se emplea el registro ra (por **r**eturn **a**ddress) para guardar las direcciones de retorno.
    - El llamado (call), en caso del procesador MIPS, emplea la instrucción jal, que efectúa dos acciones: guardar en el registro ra la dirección de retorno, y colocar en PC, la dirección de inicio de la subrutina.
    - El retorno se implementa con un salto incondicional relativo a registro, en éste procesador el registro ra.

Funciones

- **Uso del Stack**
  - Ha sido frecuente emplear el stack (pila) para almacenar las direcciones de retorno.



Funciones

- **Uso del Stack**
  - En caso de usar stack, la instrucción de llamado a subrutina, se implementa según:
    - Guarda la dirección de retorno; en el tope
    - Salto a dirección de inicio de subrutina; invocación.
  - Y la instrucción de retorno, según:
    - Retorna la dirección guardada a PC;
    - Vuelve a la siguiente instrucción, desde el punto de invocación.

Funciones

- **Uso del Stack**
  - Otro uso frecuente de la pila es el **almacenamiento** o salvado **temporal de registros** en la memoria.
  - Ocurre muy a menudo en arquitecturas con número reducido de registros.
  - La necesidad aparece cuando dentro de una subrutina, se requieren registros y los disponibles están almacenando variables del programa.
  - En este caso se empuja un registro, quedando éste disponible, luego se restaura o recupera el valor original.

Funciones

- **Frame**
  - Es el espacio de memoria que cada función ocupa para almacenar sus variables en el stack.
  - Logra mantener en direcciones contiguas de memoria a las variables que puede emplear la función mientras se ejecuta su código.
  - Sólo es necesario disponer del frame mientras la función esté en ejecución, lo cual permite reutilizar las celdas ocupadas.
  - Las variables locales, las definidas dentro del cuerpo de acciones de la función, se guardan en el frame.

Funciones

- **Frame**
  - Los argumentos también se guardan en el frame.
  - Si almacenamos en el frame la dirección de la instrucción a la que debe retornarse, luego de ejecutada la función, podremos invocar a funciones dentro de una función.
  - Cada vez que se llame a una función se crea un frame.

Funciones

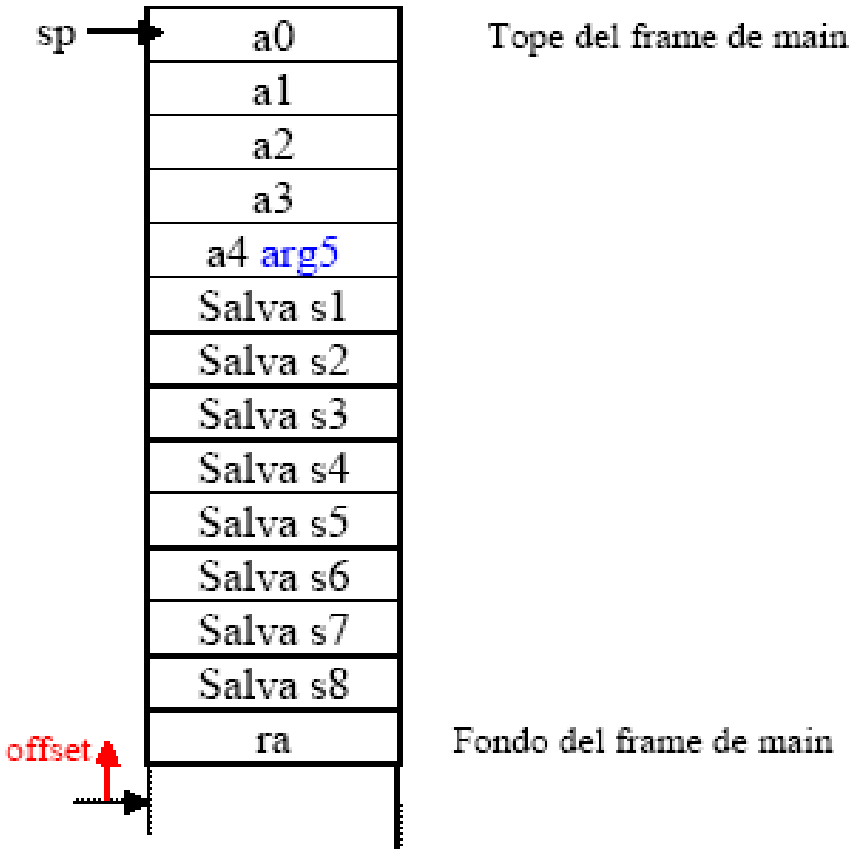
- **Frame**
  - Entonces, antes de llamar a una función se introducen en el frame, los valores de los argumentos actuales en celdas contiguas de memoria; luego se introduce la dirección de retorno, y finalmente se llama a la función.
  - El código de la función crea el espacio para las variables locales en el frame, en celdas contiguas.
  - Antes de salir de la función, se retorna el valor; luego se recupera la dirección de retorno y finalmente se desarma el frame.

Funciones

- **Argumentos en el frame.**
  - Se denomina frame a la zona de memoria asignada a una función, en forma dinámica, para almacenar variables y argumentos mientras el código de la función esté ejecutándose.
  - Los argumentos se pasan en los registros $a0 hasta $a4
  - Los valores de retorno en $v0 y $v1
  - Las variables locales que deseen ser almacenadas en registros emplean los registros s (pero deben ser preservados sus contenidos antes de ser empleados por la función, y recuperados sus valores originales antes de salir de la función)
  - Durante la ejecución pueden emplearse libremente los registros t.

Funciones

```
.frame $sp,56,$31
addu $sp,$sp,-56
.mask 0xc0fe0000,-4
sw $17,20($sp)
sw $18,24($sp)
sw $19,28($sp)
sw $20,32($sp)
sw $21,36($sp)
sw $22,40($sp)
sw $23,44($sp)
sw $30,48($sp)
sw $31,52($sp)
```



Comunicación de las funciones

- *Comunicación por variables globales.*
- *Comunicación por argumentos. Paso por valor.*
- *Comunicación por argumentos. Paso por referencia.*

Comunicación de las funciones

- *Comunicación por variables globales.*

```
int i, j, k;                    /*variables globales*/

/*Definición de la función */
void f1(void)
{ k = i + j; }
main( )
{ ....
f1( ); /*aquí se invoca a la subrutina o función f1 sin argumentos */
..... /*aquí debería retornar a ejecutar la siguiente sentencia */
}
```

Comunicación de las funciones

- *Comunicación por variables globales.*
  - *En assembler*

```
    .data
i:  .word 0x0 #definición de variables globales (estáticas)
j:  .word 0x0
k:  .word 0x0
    .text
    .globl main #se especifica tipo de rótulo
main:
    ......
    jal f1 # invocación de la función (por su dirección).
    ....... # se guarda en ra, la dirección de la instrucción después del jal.
    j main # se vuelve a repetir la función principal.
```

Comunicación de las funciones

- *Comunicación por variables globales.*
  - *En assembler función f1*

```
f1: la $t0, i          # t0 apunta a variable i. t0 = &i
    lw $s1, 0($t0)     # s1 = i lee desde la memoria las variables
    lw $s2, 4($t0)     # s2 = j
    add $t1, $s1, $s2  # t1 = i + j
    sw $t1, 8($t0)     # k = t1 escribe en variable en memoria
    jr $ra             # retorna a la instrucción siguiente al jal.
```

Comunicación de las funciones

- *Comunicación por argumentos. Paso por valor.*

```
int i, j, k;                /* globales */
register int f2(register int a0, register int a1)   /* a0 y a1 son parámetros*/
{
return (a0+a1);
}
main()
{
....
k = f2(i, j);
/* invocación a f2 con parámetros actuales i y j. Pueden ser expresiones*/
.....
}
```

Comunicación de las funciones

- *Comunicación por argumentos. Paso por valor.*
  - *En assembler*

```
        .data
i:      .word 0x0
j:      .word 0x0
k:      .word 0x0
        .text
        .globl main
main:
        .....
        la $t0, i
        lw $a0,0($t0)     # a0 = i Carga de valores.
        lw $a1,4($t0)     # a1 = j
        jal f2            # invocación de función
        sw $v0,8($t0)     # k = v0 Almacenar resultado retornado.
        .......
        j main
```

Comunicación de las funciones

- *Comunicación por argumentos. Paso por valor.*
  - *En assembler función f2*

```
f2: add $v0, $a0, $a1        #La función comienza en la dirección f2.
    jr $ra
```

Comunicación de las funciones

- ***Comunicación por argumentos. Paso por referencia.***

```
int i, j, k;
register int f3(register int *a0, register int *a1) /*argumentos por
    referencia*/
{ return (*a0+*a1); }
main()
{ ....
k= f3(&i,&j); /*Invocación. Paso de las direcciones de las variables*/
.....
}
```

Comunicación de las funciones

- ***Comunicación por argumentos. Paso por referencia.***
  - En assembler

```
        .data
i:      .word 0x0
j:      .word 0x0
k:      .word 0x0
        .text
        .globl main
main:
        ......
        la $a0,i #carga punteros. Se pasa una referencia a las variables.
        la $a1,j #En los argumentos se pasan valores de punteros.
        jal f3 #invocación de función
        sw $v0,k(0) # k = v0 (es una macro)
        .......
        j main
```

Comunicación de las funciones

- ***Comunicación por argumentos. Paso por referencia.***
  - En assembler función f3

```
f3: lw $t1,0($a0) # t1 = i
    lw $t2,0($a1) # t2 = j
    add $v0, $t1, $t2
    jr $ra
```

Funciones

- Ejemplo. Función recursiva.
- ***Código en C.***

```
#include <stdio.h>
int fact(int n)
   { if (n>0) return( n*fact(n-1) ); else return( 1); }
void main(void)
   { printf("\nEl factorial de 3 es: %d", fact(3)); }
```

Funciones

- Ejemplo. Función recursiva.
- ***Código en assembler.***

```
            .data
mensaje: .asciiz "\nEl factorial de 3 es: "
            .text
            .globl main
main:       subu $sp, $sp, 4 #push ra
            sw $ra, 0($sp)

            #acciones de main
            li $a0, 3 # pasa constante 3 como argumento de fact
            jal fact

            move $a1, $v0 # el retorno de fact se lleva al argumento a1 de printd
            la $a0, mensaje # argumento a0 de printd. Imprime salida de fact
            jal printd

            lw $ra,0($sp) #restaura ra
            addu $sp,$sp,4
            jr $ra #retorna de main
```

Funciones

- Ejemplo. Función recursiva.
- Código en assembler

```
fact:       addiu $sp, $sp, -8    #crea espacio para n y ra (8 bytes)
            sw $ra, 0($sp)
            #acciones de fact
            sw $a0, 4($sp)         # salva a0 en frame. Inicia argumento.
            lw $v0, 4($sp)
            bgtz $v0, ifpos        # salta si positivo
            li $v0, 1              # f(0)=1
            j salirfact
ifpos:      lw $v1, 4($sp)         # v1 = n
            addi $v0, $v1, -1      # v0 = n - 1
            move $a0, $v0
            jal fact # fact(n-1)
            lw $v1, 4($sp)         # v1 = n
            mul $v0, $v0, $v1      # v0 = fact(n-1)*n
salirfact:  lw $ra, 0($sp)         #restaura ra
            addiu $sp, $sp,8
            jr $ra                 #retorna de fact
```

Funciones

- Ejemplo. Función recursiva.

STACK				
[0x7fffefd8]	0x00400078	0x00000000		
[0x7fffefe0]	0x00400078	0x00000001	0x00400078	0x00000002
[0x7fffeff0]	0x00400034	0x00000003	0x00400018	0x00000000

Estructuras de Datos

- Arreglos
  - Se requiere una variable entera sin signo, denominada el índice del arreglo, generalmente se emplea un registro.
  - Una zona de datos, palabras normalmente consecutivas en la memoria para almacenar las componentes.
  - Todas las componentes tienen igual tamaño, y si se asume que están almacenadas en forma contigua, se tendrá que si se conoce la dirección del primero, la dirección de la componente i, queda dada por:

    Dirección del primero + i * (tamaño de la componente)

Estructuras de Datos

- Arreglos
  - Este modelo de representación de los arreglos en assembler, es el que usa el lenguaje C, que emplea el nombre del arreglo como un puntero a la primera componente.
  - Permite accesar a una componente vía indirección, con la expresión a+i, que es la dirección de la componente i del arreglo a.
  - La aritmética de punteros calcula de acuerdo al tamaño, la correcta dirección de la componente; sea ésta de una, media o varias palabras.

Estructuras de Datos

- Ejemplo. Arreglos

```
int  a[ ] = {0,1,2,3,4,5,6};
int  i = 0;
int  k = 0;
void main(void)
{
      i = 5;
       ........
      k = a[i];
      .......
      a[i] = k;
}
```

Estructuras de Datos

- Ejemplo. Arreglos

```
int  a[ ] = {0,1,2,3,4,5,6};
int  i = 0;
int  k = 0;
void main(void)
{
      i = 5;
       ........
      k = *(a+i);
     .......
      *(a+i) = k;
}
```

Estructuras de Datos

- Ejemplo. Arreglos en assembler

```
.data
a:      .word  0,1,2,3,4,5,6
i:      .word  0
k:      .word  0
.text
.globl  main
```

Estructuras de Datos

- Ejemplo. Arreglos en assembler

```
main:
        ori     $t3,$zero,5
        la      $t0, i
        sw      $t3, 0($t0)
        la      $t0, i
        lw      $t0, 0($t0)
        sll     $t2, $t0, 2
        la      $t1, a
        addu    $t2, $t2, $t1
        lw      $t3, 0($t2)
        la      $t4, k
        sw      $t3, 0($t4)
        la      $t4, k
        lw      $t3, 0($t4)
        la      $t0, i
        sll     $t2, $t0, 2
        la      $t1, a
        addu    $t2, $t2, $t1
        sw      $t3, 0($t2)
        jr      ra
```

Estructuras de Datos

- Estructuras
  - El tamaño de las componentes es variable.
  - La dirección del primer campo está dada por:
    - Dirección de inicio de la estructura
  - La dirección del segundo campo está dada por:
    - Dirección de inicio de la estructura + tamaño del primer campo.
  - La dirección del tercer campo está dada por:
    - Dirección de inicio de la estructura + suma de los tamaños del primer y segundo campo.
  - Y así sucesivamente.

Estructuras de Datos

- Estructuras. Ejemplo

```
struct punto {
    int x;
    int y;
    };
struct punto a = { 1 , 2};  /*se inicializan al definir el espacio */
struct punto b = { 3 , 4};
void main(void)
{
    a.x=b.x; a.y=b.y;       /*Se puede escribir a= b; */
}
```

Estructuras de Datos

- Estructuras. Ejemplo

En asembler.

Se traslada a:

```
                .data
structa:        .word 1
                .word 2
structb:        .word 3
                .word 4
        .text
        .globl main
main:
        #apuntar a estructuras
        la $t0,structb
        la $t1,structa
        lw $t3,0($t0) #t3=b.x
        sw $t3,0($t1)
        lw $t4,4($t0) #t4=b.y
        sw $t4,4($t1)
        jr ra #retorna del main.
```