Además de chequear igualdad (**beq**) o desigualdad (**bne**), a veces es útil chequear si una variable es menor que otra

La instrucción

`slt  $t0, $s3, $s4` —*set on less than*

… compara los valores de los registros $s3 y $s4,

… y deja el registro **`$t0`** en 1 si el valor de **`$s3`** es menor que el de **`$s4`**, o en 0 en otro caso

También podemos comparar el valor de un registro con una constante:

`slti  $t0, $s2, 10` —*set on less than inmediate*

Al ejecutar una función, el programa debe seguir estos pasos:

- poner los parámetros en un lugar donde la función tenga acceso a ellos
- transferir el control a la función
- obtener la memoria necesaria para la ejecución de la función
- ejecutar la tarea correspondiente
- poner el resultado en un lugar donde el programa que hizo la llamada tenga acceso a él
- devolver el control al punto de origen, ya que una función puede ser llamada desde varios puntos en un programa

Dado que los registros son el lugar de más rápido acceso para tener datos en un computador, y dado que tenemos 32 de ellos, los usamos así:

- **`$a0 - $s3`**: cuatro registros para pasar parámetros
- **`$v0 - $v1`**: dos registros para retornar valores
- **`$ra`**: un registro para la dirección de retorno para volver al punto de origen

Tenemos además una instrucción solo para funciones:

- **`jal`** *direcciónDeLaFunción* —*jump and link*
- salta a una dirección —*la direcciónDeLaFunción*
- … y simultáneamente guarda la dirección de la siguiente instrucción —la **dirección de retorno**— en el registro **`$ra`** (el registro 31)

… y usamos la instrucción *jump register* para saltar incondicionalmente a la dirección especificada en un registro:

- **`jr $ra`**

El programa que hace la llamada primero pone los valores de los parámetros en los registros **`$a0`** – **`$a3`**

… y luego ejecuta **`jal X`** para saltar (al código correspondiente) a la función **`X`**

La función **`X`** entonces hace sus cálculos (o lo que sea)

… pone los resultados en los registros **`$v0`** y **`$v1`**

… y finalmente devuelve el control al programa que hizo la llamada ejecutando **`jr $ra`**

Por supuesto, el registro **`PC`** —*program counter*— guarda la dirección de la instrucción que está siendo ejecutada en este momento

… por lo que **`jal`** pone realmente **`PC`** + 4 en **`$ra`**

Si el compilador necesita más registros al llamar a una función que los cuatro registros **$a0** – **$a3**, entonces

… todos los otros registros (adicionales) que sean usados deben ser finalmente restaurados a los valores que tenían antes de que se produjera la llamada —sus valores originales:

- los valores originales de esos registros deben ser guardados en la memoria

La estructura de datos para esto es un ***stack*** —una cola del tipo ”el último en entrar es el primero en salir”, o LIFO

… y un puntero —el *stack pointer*— a la dirección más recientemente reservada en el stack para los registros de la (posible) próxima función:

- **$sp** (el registro 29)

¿Cuál es el código assembly compilado de la siguiente función?

```
int xmpl(int g, int h, int i, int j):
    int f
    f = (g+h) - (i+j)
    return f
```

—*ver diapositiva #7*

Los parámetros **`g`**, **`h`**, **`i`** y **`j`** corresponden, como dijimos, a los registros **`$a0`**, **`$a1`**, **`$a2`** y **`$a3`**

La variable **`f`** corresponde al registro **`$s0`** (este es un registro adicional)

Y el código compilado comienza con la etiqueta **`xmpl:`**

Además, para calcular el valor de **`f`**, usamos dos registros temporales, **`$t0`** y **`$t1`** (también adicionales)

Así, lo primero es guardar en el stack los valores de los registros **$s0**, **$t0** y **$t1** —tres palabras (12 bytes):

```
addi  $sp, $sp, -12
sw    $t1, 8($sp)
sw    $t0, 4($sp)
sw    $s0, 0($sp)
```

Luego, hay que calcular el valor de la expresión y guardarlo en **$s0**:

```
add   $t0, $a0, $a1
add   $t1, $a2, $a3
sub   $s0, $t0, $t1
```

… y hay que retornar este valor, guardándolo en **$v0**:

```
add   $v0, $s0, $zero
```

En preparación para volver al punto donde se produjo la llamada, restauramos los valores de los tres registros guardados en el stack:

```
lw    $s0,  0($sp)
lw    $t0,  4($sp)
lw    $t1,  8($sp)
addi  $sp, $sp, 12
```

… y finalmente retornamos ejecutando

```
jr $ra
```

En realidad, según lo que vimos antes (diap. #7), no es necesario guardar en el stack los valores de los registros **`$t0`** y **`$t1`**, ya que no representan valores de variables del programa, sino resultados intermedios

¿Qué pasa en el siguiente caso?

```
int factorial(int n):
   if n < 1:
      return 1
   else:
      return n*factorial(n-1)
```

Dada la naturaleza recursiva de la función, ahora también es necesario guardar en el stack los valores de los registros **$a0** y **$ra**, para poder restaurarlos al volver a la llamada anterior:

```
factorial:
   addi  $sp, $sp, -8
   sw    $ra, 4($sp)
   sw    $a0, 0($sp)
```

Luego, comparamos **n** con 1, y vamos al *label* **L1** si **n** ≥ 1:

```
    slti  $t0, $a0, 1
    beq   $t0, $zero, L1
```

Si **n** < 1, **factorial** devuelve 1, a través del registro **$v0**:

```
    addi  $v0, $zero, 1
    addi  $sp, $sp, 8
    jr    $ra
```

Si **n** ≥ 1, **n** es decrementado y **factorial** es llamado recursivamente:

```
L1:
    addi  $a0, $a0, -1
    jal   factorial
```

De ahí, empezamos a “salir” de (esta llamada de) la función:

```
lw    $a0, 0($sp)
lw    $ra, 4($sp)
addi  $sp, $sp, 8
```

… ponemos en **`$v0`** el producto de **`$a0`** y **`$v0`**:

```
mul   $v0, $a0, $v0
```

… y finalmente, **`factorial`** salta nuevamente a la dirección de retorno:

```
jr    $ra
```

Para la mayoría de las funciones en la práctica, es suficiente con

4 parámetros (los registros 4 al 7)

2 registros para el valor de retorno (los registros 2 y 3)

y otros 10 registros temporales (los registros 8 al 15, y los registros 24 y 25)