Universidad de Buenos Aires
Facultad de Ingeniería

86.37 - Organización de Computadoras

Trabajo Práctico - Organización de Computadoras

Batallan, David Leonardo, *Padrón 97529*
dbatallan@fi.uba.ar

20 de septiembre de 2024

Índice

1. Objetivo

Analizar el funcionamiento de un programa C + MIPS. Comprender el uso de la ABI y las herramientas explicadas en clase (QEMU + Linux MIPS32)

2. Introducción

En este trabajo Práctico, analizaremos un programa de ordenamiento de un arreglo de enteros escrito en Assembly. El método a analizar se trata del *Stooge Sort* cuya complegidad algorítmica es del orden $O(n^{2,7095})$. Dicho algoritmo esta implementado de forma recursiva en el cual primero se compara el primer elemento con el ultimo y se realiza el intercambio si el primer elemento es mayor que el ultimo. Luego se divide al arreglo en 2/3 a izquierda y se le aplica el método. Despues Se vuelve a aplicar el metodo a los 2/3 de la derecha y finalmente se lo vuelve a aplicar a los 2/3 de la izquierda.

```
function stoogesort(array L, i = 0, j = length(L)-1){
    if L[i] > L[j] then          // If the leftmost element is larger than the rightmost element
        swap(L[i],L[j])          // Then swap them
    if (j - i + 1) > 2 then      // If there are at least 3 elements in the array
        t = floor((j - i + 1) / 3)
        stoogesort(L, i, j-t) // Sort the first 2/3 of the array
        stoogesort(L, i+t, j) // Sort the last 2/3 of the array
        stoogesort(L, i, j-t) // Sort the first 2/3 of the array again
    return L
}
```

Listing 1: Pseudocódigo de stooge sort

Concepto de ABI

La *Application Binary Interface*, o por sus siglas **ABI**, es una convención que abarca el intercambio de información, tanto entre funciones como entre funciones y sistema operativo.

Esta convención señala cómo se debe manejar excepciones, organización de memoria virtual, y la iniciación de procesos, lo que facilita la implementación de la programación modular, permite que el sistema operativo preste servicios a los programas, así como también la reutilización de componentes como por ejemplo bibliotecas de funciones, entre otros beneficios.

En particular, la **ABI** señala cómo se debe cargar el **Stack**, el lugar donde la función guarda valores temporarios declarados en una función. En el stack se encuentran valores de los registros *callee-saved* (registros de la función llamada), variables locales, almacenamiento temporario, y argumentos de funciones a llamar.

La **ABI** divide al stack en tres secciones:

- **SRA:** Saved Register Area, son los registros que se deben restaurar al devolver el control. El tamaño mínimo son 8B, y el espacio debe ser múltiplo de 8B. De no tener una cantidad múltiplo de 8B se debe colocar padding.
- **LTA:** Local and Temporary Area, se trata de las variables locales y variables temporales que pueden no tener una variable correspondiente. El tamaño debe ser múltiplo de 8B, pero puede no existir en el caso de que no se utilicen.
- **ABA:** Argument Building Area, es el sitio donde se almacenan los argumentos de las funciones que son llamadas. Tiene inicialmente 16B para 4 argumentos, pero puede incrementarse con múltiplos de 8B si se emplean más parámetros.

3. Analisis Stooge Sort

A continuación se muestra el ordenamiento del arreglo paso a paso para uno de 4 elementos.

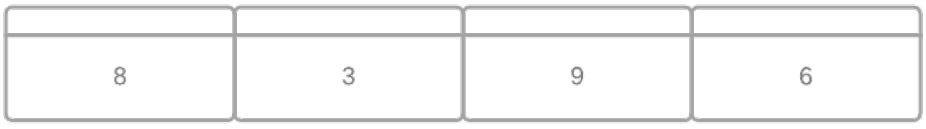

Figura 1: Arreglo desordenado

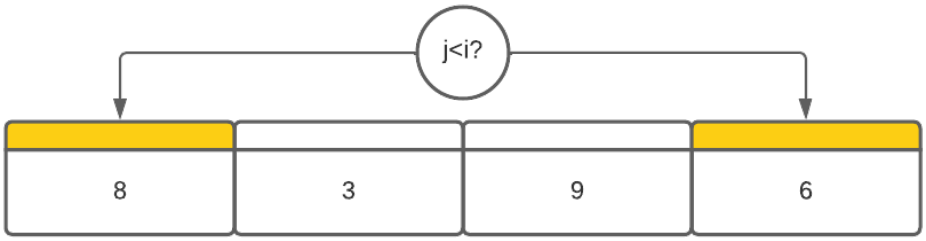

Figura 2: Comparación entre extremos del arreglo más grande

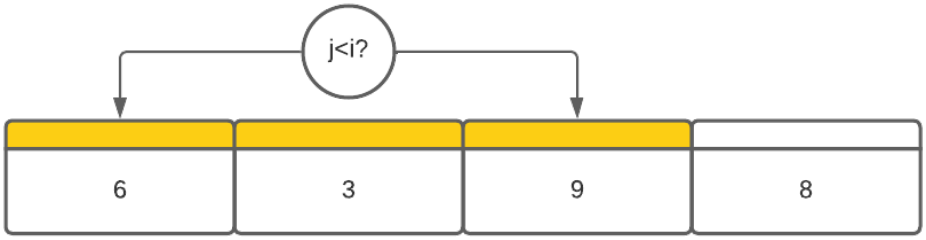

Figura 3: Comparación entre los extremos del arreglo izquierdo

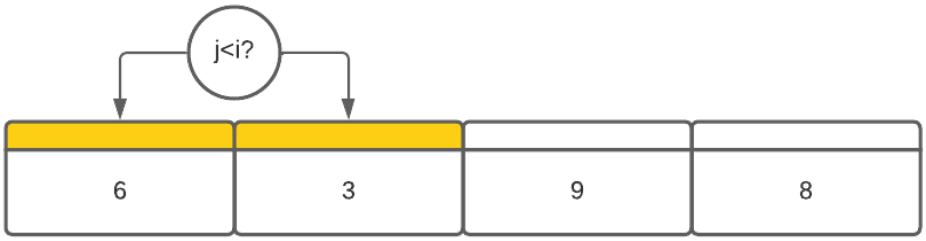

Figura 4: Comparación entre los primeros dos elementos del arreglo

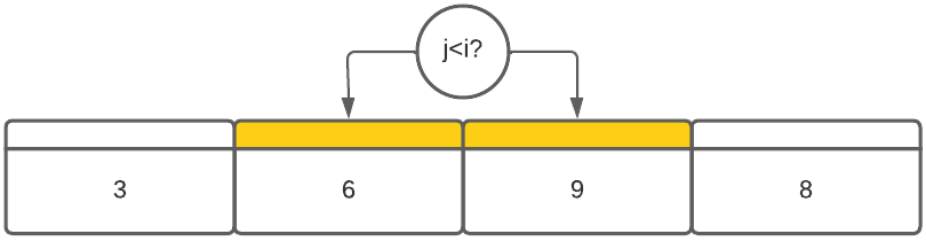

Figura 5: Comparación entre los segundos dos elementos

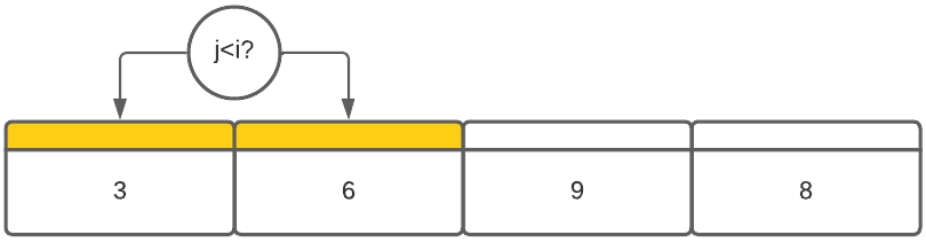

Figura 6: Comparación entre los primeros dos elementos

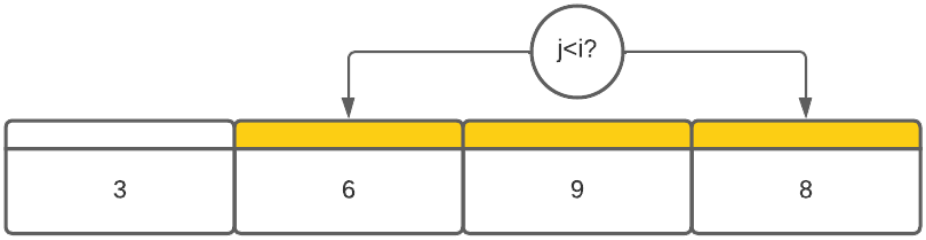

Figura 7: Comparación entre extremos del arreglo derecho

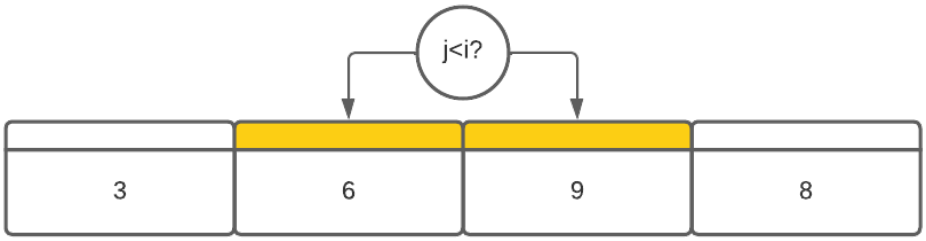

Figura 8: Comparación entre los dos elementos del medio

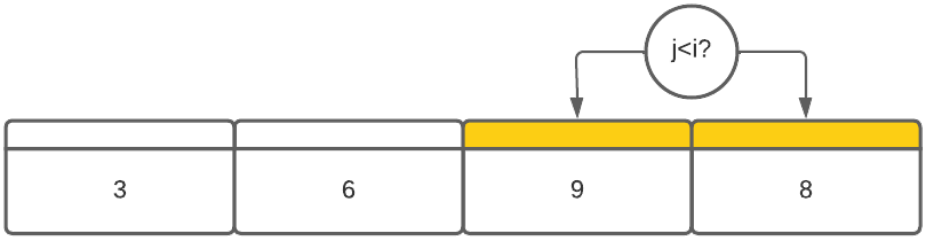

Figura 9: Comparación entre los dos elementos al final del arreglo

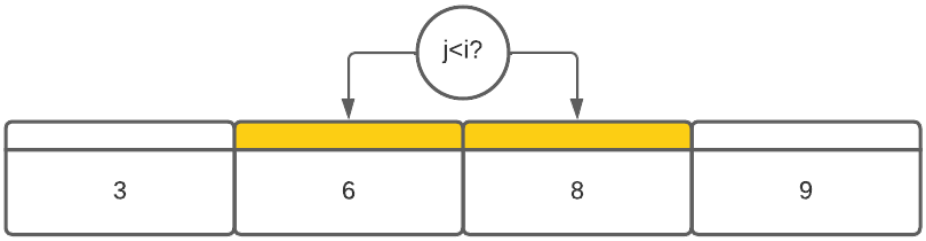

Figura 10: Comparación entre los dos elementos del medio

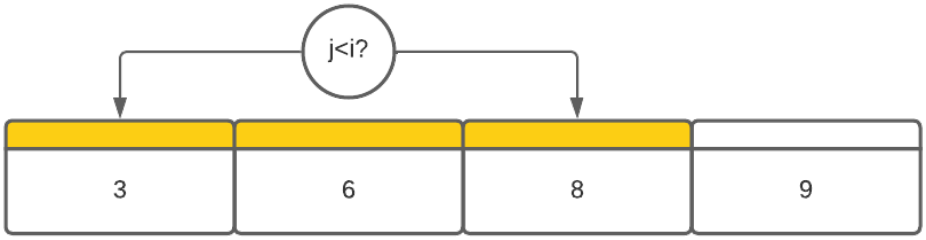

Figura 11: Comparación entre extremos del arreglo derecho

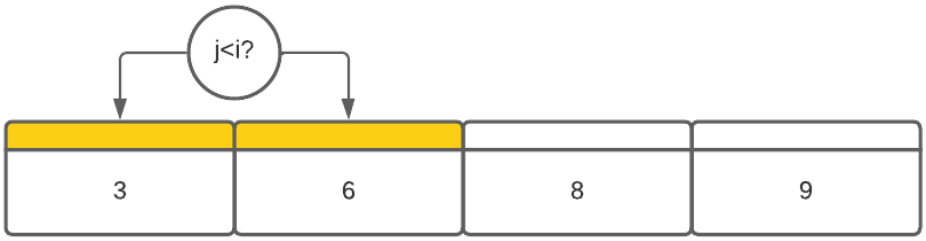

Figura 12: Comparación entre los primeros dos elementos

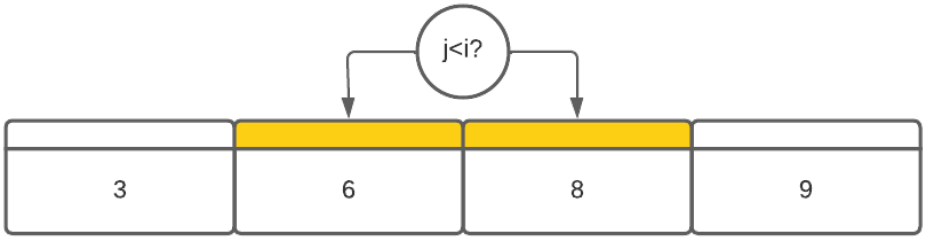

Figura 13: Comparación entre los dos elementos del medio

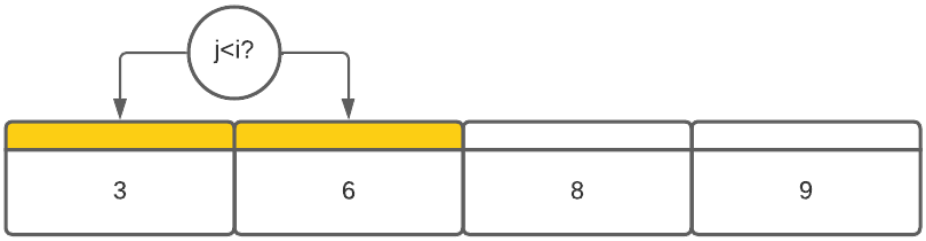

Figura 14: Comparación entre los primeros dos elementos

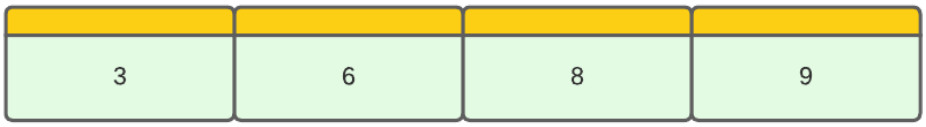

Figura 15: Arreglo ordenado

4. Desarrollo

Para el análisis del *stack frame* se utilizó la herramienta GDB la cual nos permite ejecutar de manera controlada determinadas partes del código. Para esto se establecieron *Breakpoint* en los cuales se detendrá la ejecución del programa (Listing 2). LEFT1, RIGHT y LEFT2 son etiquetas que se agregaron para poder detener la ejecucion antes de subdividir el arreglo en dos tercios y podes observar como varían las posiciones de los elementos del mismo.

```
(gdb) break main
Breakpoint 1 at 0x970: file 03-stooge_sort.c, line 36.
(gdb) break stooge_sort
Breakpoint 2 at 0xa90: file 03-stooge_sort.S, line 43.
(gdb) break LEFT1
Breakpoint 3 at 0xb5c: file 03-stooge_sort.S, line 117.
(gdb) break RIGHT
Breakpoint 4 at 0xb80: file 03-stooge_sort.S, line 127.
(gdb) break LEFT2
Breakpoint 5 at 0xba4: file 03-stooge_sort.S, line 137.
(gdb) break SWAP
Breakpoint 6 at 0xb3c: file 03-stooge_sort.S, line 100.
```

Listing 2: GDB breakpoint

En el fragmento de código mostrado en Listing 10 se observa un *stack frame* de 48 Bytes en los cuales seran 4 *Words* para la SRA, 4 para LTA y 4 para la ABA.

```
1  #include <sys/regdef.h>
2
3      .text
4      .align  2
5      .globl  stooge_sort
6      .ent    stooge_sort
7  stooge_sort:
8      .frame  fp, 48, ra
9      .set    noreordern
10     .cpload t9
11     .set    reorder
12     subu    sp, sp, 48      # 4 (SRA) + 4 (LTA) + 4 (ABA)
13     .cprestore 32           # sw gp, 32(sp)
14     sw  ra,  40(sp)
15     sw  fp, 36(sp)
16     move    fp, sp
17     sw  gp,  32(sp)
18     sw  a0, 48(sp)
19     sw  a1, 52(sp)
```

```
20      sw  a2, 56(sp)
21
22      # Local area: array, i, j, k.
23      #
24      sw  a0,   16(sp) # array
25      sw  a1,   20(sp) # i
26      sw  a2,   24(sp) # j
27      sw  zero, 28(sp) # k
```

Listing 3: Creación del Stack Frame

En la figura 16 se mustran las distintas secciones del *Stack Frame*

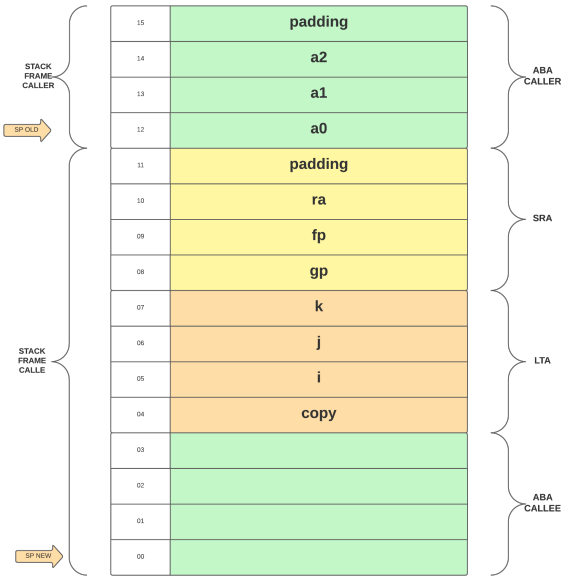

Figura 16: Diagrama del Stack Frame

Se inició la ejecución del programa el cual se detuvo en el primer *Breakpoint*, en el fragmento mostrado por GDB se puede ver como pasa de tener datos "basura.ª los valores prestablecidos. Para el analisis de este codigo solo se ejecutará el ordenamiento para los primeros 4 elementos ya que se consideró suficiente para ver como se crean distintos *Stack Frame* a medida que se llama recursivamente. Cabe destacar que el arreglo a analizar no esta en la posicion de memoria **0x4FFF6BA8** ya que la función **SORT_AND_PRINT(what, len)** realiza una copia del arreglo. Para ello se añadió un **printf()** con formato para ver la dirección de memoria del arreglo copia.

```
(gdb) run
Starting program: /home/Organizacion-de-Computadoras/TP/03-stooge_sort
Breakpoint 1, main (argc=1, argv=0x7fff6cd4) at 03-stooge_sort.c:36
36              int array[] = {8, 3, 9, 6, 1, 2, 7, 5, 4, 0};
(gdb) x/4x array
0x7fff6ba8:     0x00000000      0x00000000      0x00000001      0x77ff2000
(gdb) n
48              SORT_AND_PRINT(array, 4);
(gdb) x/4x array
0x7fff6ba8:     0x00000008      0x00000003      0x00000009      0x00000006
```

Listing 4: **Inicialización del array**

En el siguiente fragmento de shell, se puede observar el orden original del arreglo copia ([8, 3, 9, 6]) cuya dirección de memoria es **0x7FFF6BD0** que es lo que se esperaba tener en el registro a0. Mientras que al ser el primer llamado i sera 0 y j sera igual a 4-1, que son los valores de a1 y a2 respectivamente. Finalemnte si accedemos a los primeros 4 valores del arreglo copia vemos que efectivamente son los valores [8,3,9,6].

```
in : 8, 3, 9, 6.
Direccion del arreglo copy: 0x7fff6bd0

Breakpoint 2, stooge_sort () at 03-stooge_sort.S:43
43              .cpload t9
(gdb) info register
          zero       at       v0       v1       a0       a1       a2       a3
```

```
 R0   00000000 00000001 55550a90 55550d5f 7fff6bd0 00000000 00000003 00000000
            t0       t1       t2       t3       t4       t5       t6       t7
 R8   7fff6bd0 77f92118 00000801 00055561 77fbf744 8043569c 80287c48 77e4eaa8
            s0       s1       s2       s3       s4       s5       s6       s7
 R16  00000000 55550be0 77ff6edc 00000000 00000000 005426e8 00540c08 00000000
            t8       t9       k0       k1       gp       sp       s8       ra
 R24  00000000 55550a90 55561008 00000000 55568db0 7fff6b90 7fff6b90 55550a38
        status       lo       hi badvaddr    cause       pc
      0000a4f3 00000000 00000000 55561004 10800024 55550a90
          fcsr      fir  restart
      00000000 007f0000 00000000
(gdb) x/4x 7fff6bd0
Invalid number "7fff6bd0".
(gdb) x/4x 0x7fff6bd0
0x7fff6bd0:     0x00000008      0x00000003      0x00000009      0x00000006
```

Listing 5: **Primer llamado a la función Stooge Sort**

El siguiente fragmento se ejecutó hasta que se cargaron todos los elementos de **Stack Frame**

```
(gdb) n
45              subu    sp, sp, 48
(gdb)
46              .cprestore 32
(gdb)
47              sw      ra,  40(sp)
(gdb)
48              sw      fp, 36(sp)
(gdb)
49              move    fp, sp
(gdb)
50              sw      gp,  32(sp)
(gdb)
51              sw      a0, 48(sp)
(gdb)
52              sw      a1, 52(sp)
(gdb)
53              sw      a2, 56(sp)
(gdb)
57              sw      a0,   16(sp) # array
(gdb)
58              sw      a1,   20(sp) # i
(gdb)
59              sw      a2,   24(sp) # j
(gdb)
60              sw      zero, 28(sp) # k
```

Listing 6: **Creacion del Primer Stack Frame**

Mediante la instrucción **info registers** se pueden observar los valores de los registros y su correspondencia para aquellos que se guardan en el **Stack Frame**.

```
(gdb) info register
          zero       at       v0       v1       a0       a1       a2       a3
 R0   00000000 00000001 55550a90 55550d5f 7fff6bd0 00000000 00000003 00000000
            t0       t1       t2       t3       t4       t5       t6       t7
 R8   7fff6bd0 77f92118 00000801 00055561 77fbf744 8043569c 80287c48 77e4eaa8
            s0       s1       s2       s3       s4       s5       s6       s7
 R16  00000000 55550be0 77ff6edc 00000000 00000000 005426e8 00540c08 00000000
            t8       t9       k0       k1       gp       sp       s8       ra
 R24  00000000 55550a90 55561008 00000000 55568db0 7fff6b60 7fff6b60 55550a38
        status       lo       hi badvaddr    cause       pc
      0000a4f3 00000000 00000000 55561004 10800024 55550ad0
          fcsr      fir  restart
```

```
      00000000 007f0000 00000000
(gdb) x/13x $sp
0x7fff6b60:     0x7fff6b90      0x77e91c54      0x7fff6c68      0x55550d30
0x7fff6b70:     0x7fff6bd0      0x00000000      0x00000003      0x00000000
0x7fff6b80:     0x55568db0      0x7fff6b90      0x55550a38      0x55550a14
0x7fff6b90:     0x7fff6bd0
```

Listing 7: **Registros y valores de Stack frame**

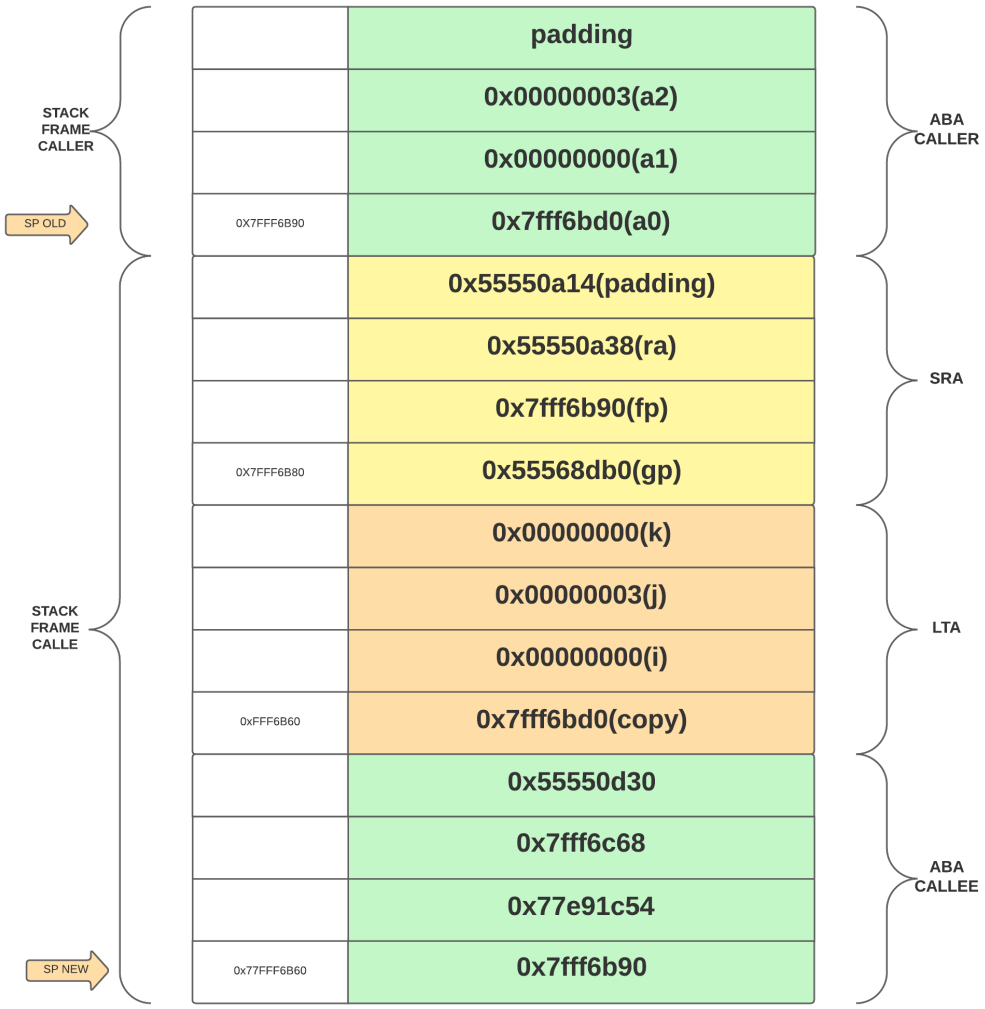

Figura 17: Diagrama del Stack Frame en la primer invocación de la función

	zero	at	v0	v1
R0	0x00000000	0x00000001	0x55550a90	0x55550d5f
	a0	a1	a2	a3
	0x7fff6bd0	0x00000000	0x00000003	0x00000000
	0xt0	t1	t2	t3
R8	0x7fff6bd0	0x77f92118	0x00000801	0x00055561
	t4	t5	t6	t7
	0x77fbf744	0x8043569c	0x80287c48	0x77e4eaa8
	0xs0	s1	s2	s3
R16	0x00000000	0x55550be0	0x77ff6edc	0x00000000
	s4	s5	s6	s7
	0x00000000	0x005426e8	0x00540c08	0x00000000
	0xt8	t9	k0	k1
R24	0x00000000	0x55550a90	0x55561008	0x00000000
	gp	sp	s8	ra
	0x55568db0	0x7fff6b60	0x7fff6b60	0x55550a38

Tabla 1: Registros

En la siguiente ejecución se muestra como varia el arreglo copia a medida que se invoca recursivamente la función stooge sort.

```
Breakpoint 6, stooge_sort () at 03-stooge_sort.S:100
100             sw      t5, 0(t0)
(gdb) x/4x 0x7fff6bd0
0x7fff6bd0:     0x00000008      0x00000003      0x00000009      0x00000006
(gdb) continue
Continuing.

Breakpoint 3, stooge_sort () at 03-stooge_sort.S:117
117             lw      a0, 16(sp) # a0: array
(gdb) x/4x 0x7fff6bd0
0x7fff6bd0:     0x00000006      0x00000003      0x00000009      0x00000008
(gdb) continue
Continuing.

Breakpoint 2, stooge_sort () at 03-stooge_sort.S:43
43              .cpload t9
(gdb) x/4x 0x7fff6bd0
0x7fff6bd0:     0x00000006      0x00000003      0x00000009      0x00000008
(gdb) continue
Continuing.

Breakpoint 3, stooge_sort () at 03-stooge_sort.S:117
117             lw      a0, 16(sp) # a0: array
(gdb) x/4x 0x7fff6bd0
0x7fff6bd0:     0x00000006      0x00000003      0x00000009      0x00000008
(gdb) continue
Continuing.

Breakpoint 2, stooge_sort () at 03-stooge_sort.S:43
43              .cpload t9
(gdb) x/4x 0x7fff6bd0
0x7fff6bd0:     0x00000006      0x00000003      0x00000009      0x00000008
(gdb) continue
Continuing.

Breakpoint 6, stooge_sort () at 03-stooge_sort.S:100
100             sw      t5, 0(t0)
(gdb) x/4x 0x7fff6bd0
0x7fff6bd0:     0x00000006      0x00000003      0x00000009      0x00000008
(gdb) continue
```

```
Continuing.

Breakpoint 4, stooge_sort () at 03-stooge_sort.S:127
127             lw      a0, 16(sp) # a0: array
(gdb) x/4x 0x7fff6bd0
0x7fff6bd0:     0x00000003      0x00000006      0x00000009      0x00000008
(gdb) continue
Continuing.

Breakpoint 2, stooge_sort () at 03-stooge_sort.S:43
43              .cpload t9
(gdb) x/4x 0x7fff6bd0
0x7fff6bd0:     0x00000003      0x00000006      0x00000009      0x00000008
(gdb) continue
Continuing.

Breakpoint 5, stooge_sort () at 03-stooge_sort.S:137
137             lw      a0, 16(sp) # a0: array
(gdb) x/4x 0x7fff6bd0
0x7fff6bd0:     0x00000003      0x00000006      0x00000009      0x00000008
(gdb) continue
Continuing.

Breakpoint 2, stooge_sort () at 03-stooge_sort.S:43
43              .cpload t9
(gdb) x/4x 0x7fff6bd0
0x7fff6bd0:     0x00000003      0x00000006      0x00000009      0x00000008
(gdb) continue
Continuing.

Breakpoint 4, stooge_sort () at 03-stooge_sort.S:127
127             lw      a0, 16(sp) # a0: array
(gdb) x/4x 0x7fff6bd0
0x7fff6bd0:     0x00000003      0x00000006      0x00000009      0x00000008
(gdb) continue
Continuing.

Breakpoint 2, stooge_sort () at 03-stooge_sort.S:43
43              .cpload t9
(gdb) x/4x 0x7fff6bd0
0x7fff6bd0:     0x00000003      0x00000006      0x00000009      0x00000008
(gdb) continue
Continuing.

Breakpoint 3, stooge_sort () at 03-stooge_sort.S:117
117             lw      a0, 16(sp) # a0: array
(gdb) x/4x 0x7fff6bd0
0x7fff6bd0:     0x00000003      0x00000006      0x00000009      0x00000008
(gdb) continue
Continuing.

Breakpoint 2, stooge_sort () at 03-stooge_sort.S:43
43              .cpload t9
(gdb) x/4x 0x7fff6bd0
0x7fff6bd0:     0x00000003      0x00000006      0x00000009      0x00000008
(gdb) continue
Continuing.

Breakpoint 4, stooge_sort () at 03-stooge_sort.S:127
127             lw      a0, 16(sp) # a0: array
(gdb) x/4x 0x7fff6bd0
```

```
0x7fff6bd0:     0x00000003      0x00000006      0x00000009      0x00000008
(gdb) continue
Continuing.

Breakpoint 2, stooge_sort () at 03-stooge_sort.S:43
43              .cpload t9
(gdb) x/4x 0x7fff6bd0
0x7fff6bd0:     0x00000003      0x00000006      0x00000009      0x00000008
(gdb) continue
Continuing.

Breakpoint 6, stooge_sort () at 03-stooge_sort.S:100
100             sw      t5, 0(t0)
(gdb) x/4x 0x7fff6bd0
0x7fff6bd0:     0x00000003      0x00000006      0x00000009      0x00000008
(gdb) continue
Continuing.

Breakpoint 5, stooge_sort () at 03-stooge_sort.S:137
137             lw      a0, 16(sp) # a0: array
(gdb) x/4x 0x7fff6bd0
0x7fff6bd0:     0x00000003      0x00000006      0x00000008      0x00000009
(gdb)
```

Listing 8: **Ordenamiento del arreglo**

5. Código Fuente

```
1  /*
2   * 03-stooge_sort.c - Prueba de stooge_sort.S. La idea,
3   * es invocar sucesivamente a stooge_sort(), usando datos
4   * progresivamente mas complejos.
5   *
6   * Para compilar:
7   *
8   * $ cc -g -Wall -o x 03-stooge_sort.c 03-stooge_sort.S
9   *
10  * $Date: 2008/04/18 19:58:31 $
11  */
12
13 #include <stdio.h>
14 #include <string.h>
15 #include <sys/types.h>
16
17 extern void stooge_sort(int *, size_t, size_t);
18
19 static void
20 print(const char *prefix, int *array, size_t n)
21 {
22     const char *comma = "";
23
24     if (prefix != 0)
25         printf("%s", prefix);
26     while (n-- > 0) {
27         printf("%s%d", comma, *array++);
28         comma = ", ";
29     }
30     printf(".\n");
31 }
32
33 int
34 main(int argc, char * const argv[])
35 {
36     int array[] = {6, 8, 3, 9, 1, 2, 7, 5, 4, 0};
37
38 #define SORT_AND_PRINT(what, len)                       \
39     do {                                                \
40         int copy[10];                                   \
41                                                         \
```

```
        memcpy(copy, what, len * sizeof(int));                  \
        print("in : ", copy, len);                              \
        printf("Direccion del arreglo copy: %p\n",copy);        \
        stooge_sort(copy, 0, len - 1);                          \
        print("out: ", copy, len);                              \
    } while (0)

    SORT_AND_PRINT(array, 1);
    SORT_AND_PRINT(array, 2);
    SORT_AND_PRINT(array, 3);
    SORT_AND_PRINT(array, 4);
    SORT_AND_PRINT(array, 5);
    SORT_AND_PRINT(array, 6);
    SORT_AND_PRINT(array, 7);
    SORT_AND_PRINT(array, 8);
    SORT_AND_PRINT(array, 9);
    SORT_AND_PRINT(array, 10);

    return 0;
}
```

Listing 9: 03-stooge_sort.c

```
/*
 * 03-stooge_sort.S - Implementacion MIPS del siguiente metodo:
 *
 * void
 * stooge_sort(int *array, size_t i, size_t j)
 * {
 *         ssize_t k = (j - i + 1)/3;
 *
 * #define SWAP(type, x, y)        \
 *         do {                    \
 *                 type tmp = (x); \
 *                 (x) = (y);      \
 *                 (y) = tmp;      \
 *         } while (0)
 *
 *         if (i == j)
 *                 return;
 *         if (array[i] > array[j])
 *                 SWAP(int, array[i], array[j]);
 *         if (i + 1 < j) {
 *                 stooge_sort(array, i, j - k);
 *                 stooge_sort(array, i + k, j);
 *                 stooge_sort(array, i, j - k);
 *         }
 * }
 *
 * Notar, que este codigo no contempla la posibilidad de
 * codigo independiente de la posicion (PIC). Eso ser
 * objeto de analisis en los ejemplos que siguen.
 *
 * $Date: 2015/10/06 13:25:18 $
 */

#include <sys/regdef.h>

    .text
    .align  2
    .globl  stooge_sort
    .ent    stooge_sort
stooge_sort:
    .frame  fp, 48, ra
    .set    noreordern
    .cpload t9
    .set    reorder
    subu    sp, sp, 48
    .cprestore 32
    sw  ra,  40(sp)
    sw  fp, 36(sp)
    move    fp, sp
    sw  gp,  32(sp)
    sw  a0, 48(sp)
    sw  a1, 52(sp)
    sw  a2, 56(sp)

```

```
    # Local area: array, i, j, k.
    #
    sw  a0,   16(sp) # array
    sw  a1,   20(sp) # i
    sw  a2,   24(sp) # j
    sw  zero, 28(sp) # k

    /*
     * k = (j - i + 1)/3
     */
    lw  t0, 24(sp) # t0: j
    lw  t1, 20(sp) # t1: i
    subu    t0, t0, t1 # t0: j - i
    addiu   t0, t0, 1  # t0: j - i + 1
    li  t2, 3      # t2: 3
    divu    t0, t0, t2 # t0: (j - i + 1)/3
    sw  t0, 28(sp) # k = (j - i + 1)/3

    /*
     * if (i == j)
     *         return;
     */
    lw  t0, 20(sp)
    lw  t1, 24(sp)
    beq t0, t1, return

    /*
     *  if (array[i] > array[j])
     *          SWAP(int, array[i], array[j]);
     */
    lw  t0, 16(sp) # t0: array
    lw  t1, 20(sp) # t1: i
    sll t1, t1, 2
    addu    t0, t0, t1 # t0: &array[i]

    lw  t2, 16(sp) # t2: array
    lw  t3, 24(sp) # t3: j
    sll t3, t3, 2
    addu    t2, t2, t3 # t2: &array[j]

    lw  t4, 0(t0)  # t4: array[i]
    lw  t5, 0(t2)  # t5: array[j]
    ble t4, t5, if2

SWAP:
    sw  t5, 0(t0)
    sw  t4, 0(t2)

if2:
    /*
     * if (i + 1 < j)
     */
    lw  t0, 20(sp) # t0: i
    addiu   t0, t0, 1  # t0: i + 1
    lw  t1, 24(sp) # t1: j
    bgeu    t0, t1, return

recursion:
LEFT1:
    /*
     * stooge_sort(array, i, j - k);
     */
    lw  a0, 16(sp) # a0: array
    lw  a1, 20(sp) # a1: i
    lw  a2, 24(sp) # a2: j
    lw  t0, 28(sp) # t0: k
    subu    a2, a2, t0 # a2: j - k
    jal stooge_sort
RIGHT:
    /*
     * stooge_sort(array, i + k, j);
     */
    lw  a0, 16(sp) # a0: array
    lw  a1, 20(sp) # a1: i
    lw  t0, 28(sp) # t0: k
    addu    a1, a1, t0 # a1: i + k
    lw  a2, 24(sp) # a2: j
```

```
132     jal stooge_sort
133 LEFT2:
134     /*
135      * stooge_sort(array, i, j - k);
136      */
137     lw  a0, 16(sp) # a0: array
138     lw  a1, 20(sp) # a1: i
139     lw  a2, 24(sp) # a2: j
140     lw  t0, 28(sp) # t0: k
141     subu    a2, a2, t0 # a2: j - k
142     jal stooge_sort
143 
144 return:
145     # Destruimos el frame.
146     #
147     move    sp, fp
148     lw  fp, 36(sp)
149     lw  ra,  40(sp)
150     addu    sp, sp, 48
151 
152     # Retorno.
153     #
154     j   ra
155     .end    stooge_sort
```

Listing 10: 03-stooge_sort.S