6620. Organización de Computadoras
Trabajo Práctico 1:
Conjunto de instrucciones MIPS

González, Juan Manuel, *Padrón Nro. 79.979*
juan0511@yahoo.com

Argüello, Osiris, *Padrón Nro. 83.062*
osirisarguello@yahoo.com.ar

Paez, Ezequiel Alejandro, *Padrón Nro. 84.474*
skiel85@gmail.com

1er. Cuatrimestre de 2012
66.20 Organización de Computadoras – Prática Jueves
Facultad de Ingeniería, Universidad de Buenos Aires

Resumen

El presente trabajo práctico tiene como objetivo familiarizarse con el conjunto de instrucciones MIPS32 y el concepto de ABI, desarrollando para ello un programa que implemente el algoritmo de ordenamiento *mergesort*.

1. Introducción

El presente trabajo tiene como objetivo familiarizarse con el conjunto de instrucciones MIPS32 y el concepto de ABI. Se utilizará el programa GXemul para simular un entorno de desarrollo en una máquina MIPS (corriendo una versión del sistema operativo NetBSD).
Se desarrollará, en lenguaje assembly, un programa que implemente el algorimto de ordenamiento *mergesort*. La implementación de este trabajo práctico debe tomar entrada exclusivamente desde *stdin*. La salida debe imprimirse por *stdout*, mientras que los errores deben imprimirse por *stderr*.

2. Programa a implementar

Al igual que el TP0, el presente trabajo práctico tiene como objetivo implementar el algoritmo de ordenamiento *mergesort*, esta vez en assembly de MIPS32. La implementación de este trabajo práctico debe tomar entrada exclusivamente desde *stdin*. La salida debe imprimirse por *stdout*, mientras que los errores deben imprimirse por *stderr*.
Asimismo, la implementación a realizar debe respetar la ABI usada por la cátedra[7], que difiere de la utilizada por el GCC.
Por otro lado, para hacer uso de memoria dinámica, se ha provisto junto con el enunciado de una implementación de *malloc* y *free* escrita en assembly MIPS32. La función *realloc* debe ser implementada como parte del trabajo práctico.
Finalmente, para poder realizar el trabajo práctico, será necesario interactuar con el sistema operativo mediante las llamadas al sistema SYS_read y SYS_write por medio de *syscalls*. Los pasos necesarios para lograr esto fueron expuestos en clase durante la explicación del enunciado del trabajo:

- Cargar el número de llamada al sistema en el registro $v0.
- Cargar los argumentos en los registros correspondientes.
- Ejecutar la llamada mediante la instrucción *syscall*.
- Verificar la existencia de errores en el registro $a3.
- Obtener el valor retornado del registro $v0.

La cátedra ha adjuntado al enunciado del trabajo práctico una versión minimalista del comando echo de UNIX a modo de ejemplo.

3. Consideraciones sobre el desarrollo

A continuación se detallan las consideraciones tenidas en cuenta para el desarrollo del trabajo práctico. Hemos decidido separar las que surgen del Diseño de las que surgen en la Implementación.

3.1. Consideraciones de Diseño

El programa desarrollado para resolver el presente trabajo práctico consta a grandes rasgos de un intérprete de línea de comandos (simplemente verifica que no se pasen argumentos al programa), la rutina que lee los datos de entrada desde *stdin*, una estructura de datos diseñada para almacenarlos, la función encargada de efectuar el ordenamiento y finalmente la función que imprime la salida del programa. Se decidió separar el código fuente en una serie de archivos, a modo de mejorar su legibilidad y organización. A continuación se presenta la lista de archivos que componen el proyecto:

Archivo	Descripción
tp1.S	Función main e intérprete de línea de comandos.
manejoes.S	Función de lectura de entrada y escritura de salida.
sort.S	Función de ordenamiento por *mergesort*.
mymemory.S	Funciones adicionales de manejo de memoria dinámica.
mymalloc.S	Implementación de mymalloc y myfree.
io.S	Implementación de my_read y my_write.
constantes.h	Definición de las constantes y códigos de error internos del programa.

Dada la similitud del problema a resolver con el TP0, se decidió como punto de partida modificar el código de dicho trabajo práctico para hacerlo cumplir con la especificación del nuevo enunciado. El haber hecho esto permitió contar con una versión del programa implementada en alto nivel para utilizar de guía a la hora de codificar en assembly MIPS32. Dicha solución se entrega junto con el presente informe a modo de referencia.
La estrategia utilizada, una vez obtenido dicho código C, fue la de ir reemplazando cada uno de los archivos fuente por su equivalente codificado en MIPS32, en forma parcial, de manera de ir haciendo pruebas incrementales de la implementación en assembly. De esta manera, se codificó primero el algoritmo de *sort*, y se reemplazó el archivo **'sort.c'** de la solución original por uno nuevo, **'sort.S'**, ya perteneciente a la solución definitiva. Esto simplificó de manera significativa la depuración del código assembly, ya que no eran agregados nuevos archivos codificados en assembly a la solución final hasta no tener la certeza de que el último de los archivos **.S** agregado funcionara de manera correcta. De esta manera se continuó con el desarrollo hasta que fueron reemplazados todos los archivos codificados en alto nivel y el programa contó con la totalidad del código desarrollado en assembly.

En cuanto al diseño del programa en sí, no difiere en gran parte del TP0, salvo por el hecho de que fueron eliminadas tanto la impresión de la ayuda como la de la versión, y de que en este trabajo los datos de entrada son leídos exclusivamente por *stdin*. Para mejorar el *feedback* al usuario, se agregó una validación sobre el número argc, de modo que el programa devuelva error si es llamado con algún argumento.
Por otro lado, los códigos de error internos al programa fueron actualizados, y también se hizo necesario contar con un archivo común no sólo para definir dichos valores pero a su vez también algunas constantes propias de la operación del programa, que anteriormente se encontraban en las distintas cabeceras (archivos **.h**). Dichos valores y códigos fueron definidos en el archivo 'constantes.h'. Su contenido es el siguiente:

```
#define stdinfd 0
#define stdoutfd 1
#define stderrfd 2

#define BUFFER_SIZE 2048

#define ERROR_RESERVA_INICIAL_MEMORIA 2
#define ERROR_RESERVA_MEMORIA 3
```

La ejecución del programa se interrumpe por cualquiera de estos errores, mostrando al usuario un mensaje afin. Los códigos 0 y 1 son utilizados para la devolución al SO del resultado de la ejecución, siendo:

0: Ejecución sin problemas.
1: Error de ejecución.

Finalmente, cabe destacar que todas las impresiones del programa son ruteadas por *stdout*, y los mensajes de error a través de *stderr*.

Si bien se contaba con una implementación en C, esto no fue suficiente para realizar un pasaje directo a MIPS32, dada la utilización en dicho código de funciones pertenecientes a la biblioteca estándar de C, como por ejemplo *malloc* y *fread* entre otras. En el caso de *malloc* y *free*, la cátedra proveyó un archivo con dichas funciones ya implementadas (**'mymalloc.S'**), mientras que para la implementación de la lectura y escritura se aprovechó el archivo (**'io.S'**), también entregado junto con el enunciado. Se aclara de todas maneras que para poder utilizar estas dos últimas como parte de la solución fue necesario corregir parte del código, dado que faltaba en ambos casos la instrucción encargada de modificar el *stack pointer* una vez ejecutada la función. Distinto fue el caso de las funciones *realloc* y *memcpy*, que debieron ser implementadas como parte del trabajo práctico. Para resolver su codificación, se repitió el ejercicio descripto anteriormente de logar primero una versión en C y luego realizar el pasaje a assembly.
En síntesis, el listado de funciones que fueron implementadas en MIPS32 para resolver el TP son:

```
int main(int argc,char **argv)
int mergeSort(char *list,unsigned long length)
void *mymemcpy(void *destination,const void *source,size_t num)
void *myrealloc(void *ptr,size_t old_size,size_t num)
int procesarEntrada(tDynArray *datos_sort)
int imprimirSalida(tDynArray *datos_sort)
```

3.2. Consideraciones de Implementación

En esta sección del informe se procederá a exponer los puntos mas relevantes relacionados con la implementación del trabajo práctico.

3.2.1. Generación de ejecutables

Para generar el ejecutable del programa utilizando la versión implementada en lenguaje C, debe correrse la siguiente sentencia en una terminal:

```
$ gcc -Wall -o tp1 tp1.c manejoes.c sort.c
```

Para utilizar el código assembly MIPS32, debe ejecutarse lo siguiente:

```
$ gcc -Wall -o tp1 tp1.S manejoes.S sort.S mymemory.S mymalloc.S io.S
```

Nótese que para ambos casos se han activado todos los mensajes de 'Warning' (-Wall).

3.2.2. Descripción del algoritmo de la función *main*

La función main tiene como objetivo la validación de la sintaxis de ejecución, la reserva inicial de memoria para la estructura de almacenamiento de los datos de entrada, la llamada a las distintas funciones que tiene el programa y el manejo e impresión de los mensaje de error si se da el caso.

El planteo de la solución en MIPS32 comenzó mediante el análisis de la implementación previamente obtenida en C, dando como resultado el stack de la función y un lineamiento de las estructuras de control a utilizar. Con esto, se procedió a efectuar un proceso de traducción del código de alto nivel al assembly, por etapas:

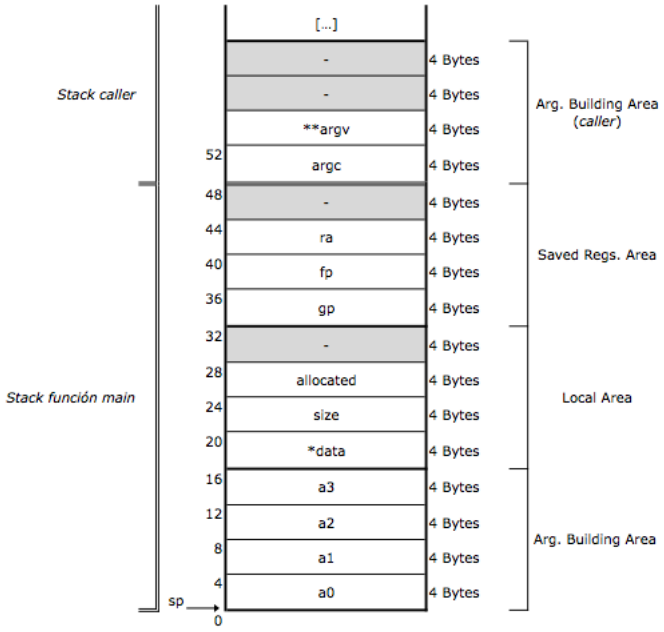
1. Manejo de la creación y destrucción del *stack* de la función.
2. Reserva y liberación de memoria para la estructura de almacenamiento utilizada.
3. Llamada a la función de lectura de datos de entrada.
4. Llamada a la función de escritura de la salida del programa.
5. Llamada al algoritmo de ordenamiento.
6. Uso del registro *v0*, para detectar el código de error devuelto por las distintas operaciones.
7. Presentación de mensajes de error al usuario.

Al ser finalizado cada uno de éstos puntos, se probaba el programa, para asegurar un correcto funcionamiento y evitar los inconvenientes derivados de realizar las primeras pruebas sobre el desarrollo completo.

Para el resto de las funciones implementadas se siguió una metodología análoga, por lo que se presentará solamente el *stack* de cada una de ellas, a menos que sean necesarias aclaraciones adicionales.

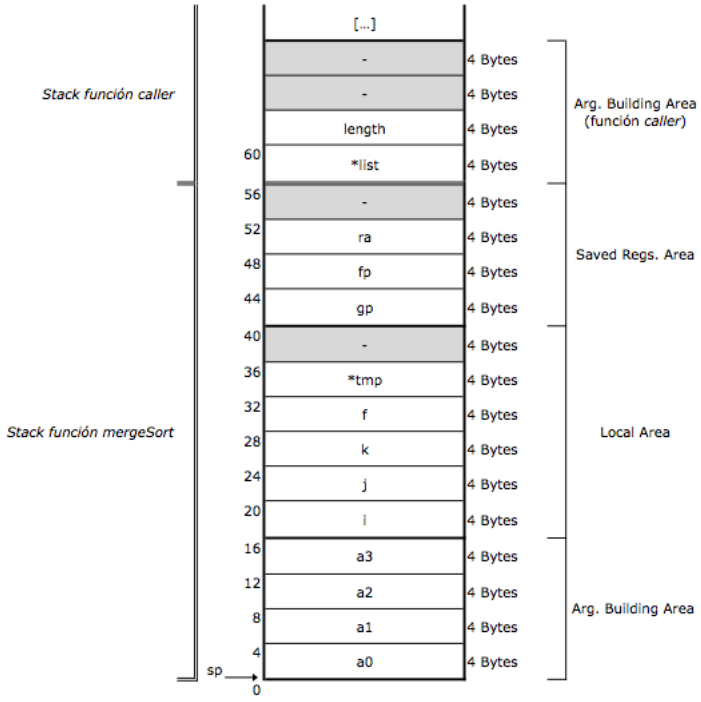
3.2.3. Diagrama del *stack* de la función *main*

A continuación se muestra el diagrama del *stack* para implementación en MIPS32 de la función *main*:



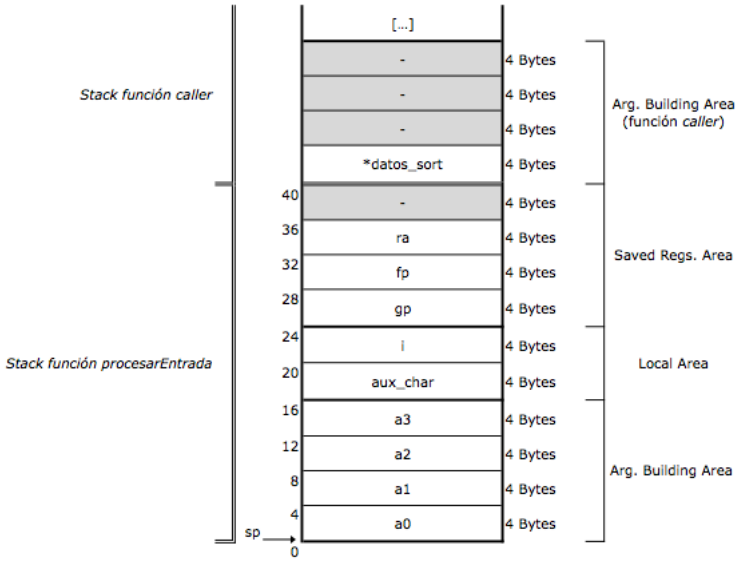
3.2.4. Diagrama del *stack* de la función *mergeSort*

A continuación se muestra el diagrama del *stack* para implementación en MIPS32 de la función *mergeSort*:



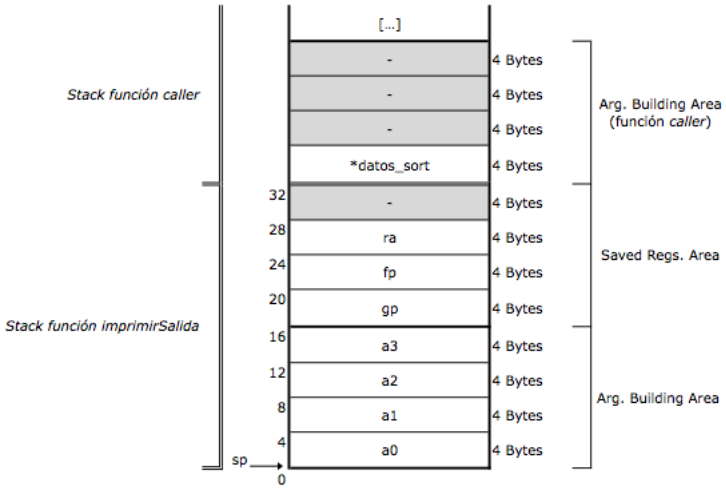
3.2.5. Diagrama del *stack* de la función *procesarEntrada*

A continuación se muestra el diagrama del *stack* para implementación en MIPS32 de la función *procesarEntrada*:



3.2.6. Diagrama del *stack* de la función *imprimirSalida*

A continuación se muestra el diagrama del *stack* para implementación en MIPS32 de la función *imprimirSalida*:



3.2.7. Descripción del algoritmo de la función *myrealloc*

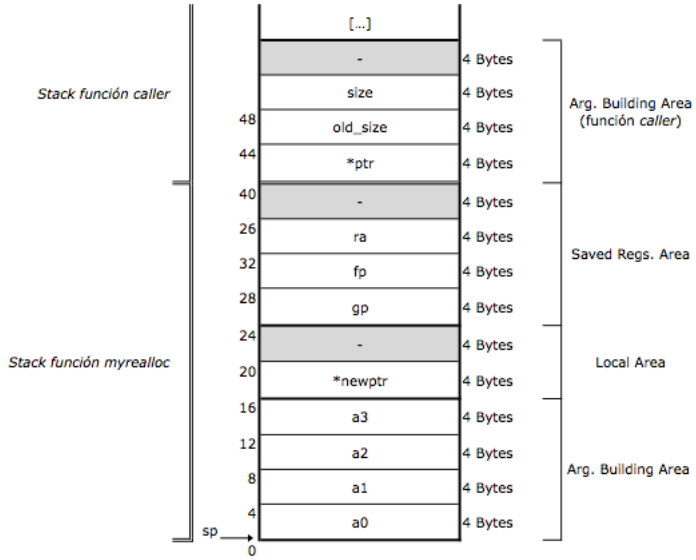
Para la función *myrealloc* se utilizó como guía el siguiente código, desarrollado en C:

```
void *myrealloc(void *ptr, size_t old_size, size_t size) {

        int minsize;
        void *newptr;

        newptr = mymalloc (size);
        if (newptr == NULL)
                return NULL;

        if (ptr != NULL) {
                minsize = old_size;
                mymemcpy (newptr, ptr, minsize);
                myfree (ptr);
        }

        return newptr;
}
```

A diferencia de la implementación estándar, fue necesario agregar un argumento mas, *old_size*, para conocer el tamaño del bloque pasado en **ptr*. Si bien en este trabajo práctico siempre se pide aumentar la memoria reservada, lo que haría innecesario conocer este dato, se prefirió hacer una implementación con la funcionalidad completa, a costa de no utilizar la definición estándar. A su vez, la implementación elegida requirió la codificación de *memcpy*, sobre esto se aclara que una optimización posible es no ejecutar *memcpy* siempre, ya que si solo se expande el bloque actual la operción es innecesaria.

3.2.8. Diagrama del *stack* de la función *myrealloc*

A continuación se muestra el diagrama del *stack* para implementación en MIPS32 de la función *myrealloc*:



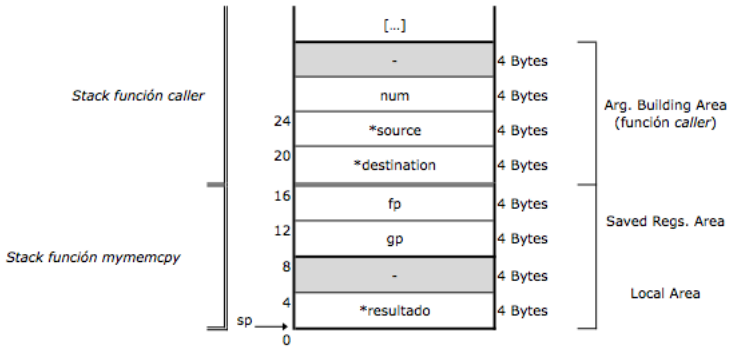
3.2.9. Descripción del algoritmo de la función *mymemcpy*

La implementación elegida para *myrealloc* requirió del desarrollo de la función *memcpy*. A continuación se presenta el código C que fue utilizado como guía:

```
void * mymemcpy (void * destination, const void * source, size_t num) {

        void * resultado = destination;

        while (num--) {
                *(char *)destination = *(char *)source;
                destination = (char *)destination + 1;
                source = (char *)source + 1;
        }

        return(resultado);
}
```

3.2.10. Diagrama del *stack* de la función *mymemcpy*

A continuación se muestra el diagrama del *stack* para implementación en MIPS32 de la función *mymemcpy*:



4. Corridas de prueba

En esta sección se presentan algunas de las distintas corridas que se realizaron para probar el funcionamiento del trabajo práctico.

En primer lugar se generó el ejecutable, se comprobó el funcionamiento de la validación de sintaxis de llamada y finalmente se ejecutó una prueba trivial:

```
root@:~/TP1# gcc -Wall -o tp1 tp1.S manejoes.S sort.S mymemory.S mymalloc.S io.S
root@:~/TP1# cat test.in
ESTOESUNAPRUEBA!
root@:~/TP1# ./tp1 test.in
tp1: syntax error, this program takes no arguments.
root@:~/TP1# ./tp1 < test.in

!AABEEENOPRSSTUUroot@:~/TP1#
```

Luego, se ejecutó la prueba provista por la cátedra en el enunciado del trabajo práctico:

```
root@:~/TP1# echo -n "9876543210" > digits.txt
root@:~/TP1# ./tp1 < digits.txt
0123456789root@:~/TP1#
root@:~/TP1# cat letters.txt
aAbBcCdDeEfFgGhHiIjJkKlLmMnNoOpPqQrRsStTuUvVwWxXyYzZroot@:~/TP1#
root@:~/TP1# ./tp1 < letters.txt
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyzroot@:~/TP1#
```

```
root@:~/TP1# head -c 64 /dev/urandom > random.txt
root@:~/TP1# hexdump -C random.txt
00000000  2a 43 21 ef ee 24 1d 6e  23 78 9a 3f 30 e6 85 f8  |*C!..$.n#x.?0...|
00000010  ea 9f 82 96 bb 1b 3e 0b  ff 2b f2 9e 78 ae aa 66  |......>..+..x..f|
00000020  dc 5e 25 83 ef 0c 93 4b  02 84 ca 06 e4 c6 a6 fe  |.^%....K........|
00000030  92 6f 0c af c9 d5 b2 74  9e 9f 43 c9 04 5d d8 c8  |.o.....t..C..]..|
00000040
root@:~/TP1# ./tp1 < random.txt > sorted.txt
root@:~/TP1# hexdump -C sorted.txt
00000000  02 04 06 0b 0c 0c 1b 1d  21 23 24 25 2a 2b 30 3e  |........!#$%*+0>|
00000010  3f 43 43 4b 5d 5e 66 6e  6f 74 78 78 82 83 84 85  |?CCK]^fnotxx....|
00000020  92 93 96 9a 9e 9e 9f 9f  a6 aa ae af b2 bb c6 c8  |................|
00000030  c9 c9 ca d5 d8 dc e4 e6  ea ee ef ef f2 f8 fe ff  |................|
00000040
root@:~/TP1#
```

Finalmente, se presenta un ejemplo más mostrando alguna de las salidas posibles, obtenida para casos en los que se trabaja con archivos binarios y cuando se toma entrada por *stdin* en modo interactivo:

```
root@:~/TP1# ./tp1
0987654321zyxcba0123456789abcxyzroot@:~/TP1#
root@:~/TP1# head -c 1048576 /dev/urandom > random1M.in
root@:~/TP1# wc -c random1M.in
 1048576 random1M.in
root@:~/TP1# ./tp1 < random1M.in | wc -c
 1048576
root@:~/TP1#
```

5. Código fuente C y MIPS32

Tanto el código C como el código MIPS32 que fue generado en el trabajo práctico ha sido incluido en el CD entregado junto al presente informe.

6. Conclusiones

A modo de conclusión, nos es inmediato resaltar la dificultad que nos ha presentado el programa motivo de este informe. No solo la adaptación requerida para trabajar con un lenguaje ensamblador y el conjunto de instrucciones del procesador, sino la incorporación del método de programación que un proyecto de este tipo conlleva.

La resolución del TP nos permitió familiarizarnos a fondo con el conjunto de instrucciones de MIPS32, así como también el concepto de ABI. La comprensión de este último concepto fue fundamental para el desarrollo de la solución, ya que en más de una oportunidad nuestro código assembly falló por estar mal diagramando el *stack* de la función a desarrollar. El partir de una implementación previa como la del TP0 nos impuso la necesidad de investigar formas de resolver la traducción del código C utilizando instrucciones que nos eran desconocidas.

Asimismo, consideramos acertada la decisión de haber realizado el desarrollo de la funciones involucradas en la solución en etapas, con pruebas intermedias como se describe anteriormente, debido a nuestra falta de experiencia con MIPS32. Lo mismo puede afirmarse de haber tomado como guía el programa desarrollado en lenguaje C.

Finalmente, en base a los resultados obtenidos y el tiempo que demandó el desarrollo de la tarea, en comparación con el esfuerzo requerido para resolver el mismo problema en lenguaje C, creemos que no es redituable la demora y dificultad encontrados. Si bien creemos que el TP constituye un ejercicio adecuado para aprender a trabajar con MIPS32, de tratarse de una decisión de implementación real, nuestra opinión es que una solución en un lenguaje de alto nivel resultaría más económica, sobre todo considerando que su rendimiento no se aprecia (por lo menos a simple observación, en base a las pruebas realizadas) lo suficientemente inferior al de una solución de bajo nivel como para justificar tiempos y esfuerzos.

Referencias

[1] GXemul, http://gxemul.sourceforge.net/

[2] Kernighan, Brian W. / Ritchie, Dennis M. El Lenguaje De Prorgamación C. Segunda Edición, PRENTICE-HALL HISPANOAMERICANA SA, 1991.

[3] Patterson / Hennesy. Computer Organization and Design; The Hardware-Software Interface. Second Edition. Morgan Kaufman, 1997.

[4] S/R autor. MIPS32^{TM}Architecture For Programmers Volume I. Revision 0.95. MIPS Technologies Inc, 2001.

[5] S/R autor. System V ABI, MIPS RISC processor supplement. Third Edition. Santa Cruz Operations, Inc, S/R AÑO.

[6] Merge-sort, http://en.wikipedia.org/wiki/Merge_sort/

[7] func_call_conv.pdf, http://groups.yahoo.com/group/orga6620/files/

[8] Oetiker, Tobias, "The Not So Short Introduction To LaTeX2", http://tobi.oetiker.ch/lshort/