Fiuba
Facultad de Ingeniería
Universidad de Buenos Aires

U.B.A. FACULTAD DE INGENIERÍA

Departamento de Electrónica

Organización de computadoras 66-20

TRABAJO PRÁCTICO #1

Conjunto de instrucciones MIPS

Curso: 2018 - 2do Cuatrimestre

Turno: Martes

GRUPO N°	
Integrantes	**Padrón**
Verón, Lucas	89341
Gamarra Silva, Cynthia Marlene	92702
Gatti, Nicolás	93570
Fecha de entrega:	16-10-2018
Fecha de aprobación:	
Calificación:	
Firma de aprobación:	

Observaciones:

Índice

1. Enunciado del trabajo práctico

66.20 Organización de Computadoras
Trabajo práctico 1: conjunto de instrucciones MIPS
`$Date: 2018/10/14 03:07:24 $`

1. Objetivos

Familiarizarse con el conjunto de instrucciones MIPS y el concepto de ABI, extendiendo un programa que resuelva el problema descripto en la sección 4.

2. Alcance

Este trabajo práctico es de elaboración grupal, evaluación individual, y de carácter obligatorio para todos alumnos del curso.

3. Requisitos

El informe deberá ser entregado personalmente, por escrito, en la fecha estipulada, con una carátula que contenga los datos completos de todos los integrantes.

Además, es necesario que el trabajo práctico incluya (entre otras cosas, ver sección 6), la presentación de los resultados obtenidos, explicando, cuando corresponda, con fundamentos reales, las causas o razones de cada caso.

4. Descripción

En este trabajo, se reimplementará parcialmente en assembly MIPS el programa desarrollado en el trabajo práctico anterior [1].

Para esto, se requiere reescribir el programa, de forma tal que quede organizado de la siguiente forma:

- `main.c`: contendrá todo el código necesario para el procesamiento de las opciones de línea de comandos, apertura y cierre de archivos (de ser necesario), y reporte de errores (`stderr`). Desde aquí se llama a las funciones de encoding y decoding siguientes.

- `base64.S`: contendrá el código MIPS32 assembly con las funciones `base64_encode()` y `base64_decode()`, y las funciones y estructuras de datos auxiliares para realizar los cómputo de encoding y decoding, que los alumnos crean convenientes. También contendrá la definición en assembly de un vector equivalente al siguiente vector C: `const char* errmsg[]`. Dicho vector contendrá los mensajes de error que las funciones antes mencionadas puedan generar, y cuyo índice es el código de error devuelto por las mismas.
- Los header files pertinentes (al menos, `base64.h`, con los prototipos de las funciones mencionadas, a incluir en `main.c`), y la declaración del vector `extern const char* errmsg[]`).

A su vez, las funciones MIPS32 `base64_encode()` y `base64_decode()` antes mencionadas, coresponden a los siguientes prototipos C:

- `int base64_encode(int infd, int outfd)`
- `int base64_decode(int infd, int outfd)`

Ambas funciones reciben por `infd` y `outfd` los file descriptors correspondientes a los archivos de entrada y salida pre-abiertos por `main.c`, la primera función realizará el encoding a base 64 de su entrada, y la segunda función el decoding de base 64 se su entrada.

Ante un error, ambas funciones volverán con un código de error numérico (índice del vector de mensajes de error de `base64.h`), o cero en caso de realizar el procesamiento de forma exitosa.

5. Implementación

El programa a implementar deberá satisfacer algunos requerimientos mínimos, que detallamos a continuación:

5.1. ABI

Será necesario que el código presentado utilice la ABI explicada en clase ([2] y [3]).

5.2. Syscalls

Es importante aclarar que desde el código assembly no podrán llamarse funciones que no fueran escritas originalmente en assembly por los alumnos. Por lo contrario, desde el código C sí podrá (y deberá) invocarse código assembly.

Por ende, y atendiendo a lo planteado en la sección 4, los alumnos deberán invocar algunos de los system calls disponibles en NetBSD (en particular, `SYS_read` y `SYS_write`).

5.3. Casos de prueba

Es necesario que la implementación propuesta pase todos los casos incluidos tanto en el enunciado del trabajo anterior [1] como en el conjunto de pruebas suministrado en el informe del trabajo, los cuales deberán estar debidamente documentados y justificados.

5.4. Documentación

El informe deberá incluir una descripción detallada de las técnicas y procesos de desarrollo y debugging empleados, ya que forman parte de los objetivos principales del trabajo.

6. Informe

El informe deberá incluir al menos las siguientes secciones:

- Documentación relevante al diseño, desarrollo y debugging del programa;
- Comando(s) para compilar el programa;
- Las corridas de prueba, (sección 5.3) con los comentarios pertinentes;
- El código fuente completoi, el cual deberá entregarse en formato digital compilable (incluyendo archivos de entrada y salida de pruebas);
- Este enunciado.

El informe deberá entregarse en formato impreso y digital.

7. Fechas

- Vencimiento: 30/10/2018.

Referencias

[1] Enunciado del primer trabajo práctico (TP0), primer cuatrimestre de 2018.

[2] System V application binary interface, MIPS RISC processor supplement (third edition). Santa Cruz Operations, Inc.

[3] MIPS ABI: Function Calling Convention, Organización de computadoras - 66.20 (archivo "func_call_conv.pdf", `http://groups.yahoo.com/groups/orga-comp/Material/`).

1.1. Diseño e implementación

Tomando como referencia el Trabajo Práctico #0 en donde el programa contenía la lógica tanto del codificador y decodificador y de otras funciones auxiliares, para este nuevo programa, se requirió re-escribirlo, de forma tal que quede organizado de la siguiente forma:

- main.c: contendrá todo el código necesario para el procesamiento de las opciones de línea de comandos, apertura y cierre de archivos (de ser necesario), y reporte de errores (stderr). Desde aquí se llama a las funciones de encoding y decoding siguientes.
- base64.S: contendrá el cádigo MIPS32 assembly con las funciones base64_encode() y base64_decode(), y las funciones y estructuras de datos auxiliares para realizar los cómputo de encoding y decoding, que los alumnos crean convenientes. También contendrá la definición en asembly de un vector equivalente al siguiente vector C: const char errmsg[]. Dicho vector contendrá los mensajes de error que las funciones antes mencionadas puedan generar, y cuyo índice es el código de error devuelto por las mismas.
- Los header files pertinentes (al menos, base64.h, con los prototipos de las funciones mencionadas, a incluir en ***main.c***), y la declaración del vector extern const char errmsg[]).

A su vez, las funciones MIPS32 base64 encode() y base64 decode() antes mencionadas, corresponden a los siguientes prototipos C:

```
1                int base64_encode(int infd, int outfd)
2                int base64_decode(int infd, int outfd)
3
```

Ambas funciones reciben por *infd* y *outfd* los *file descriptors* correspondientes a los archivos de entrada y salida pre-abiertos por ***main.c***, la primera función realizará el encoding a base 64 de su entrada, y la segunda función el decoding de base 64 de su entrada. Ante un error, ambas funciones volverán con un código de error numérico índice del vector de mensajes de error de ***base64.h***), o cero en caso de realizar el procesamiento de forma exitosa.

El programa implementado satisface los siguientes requerimientos, que se detallan a continuación:

- **ABI**
  El código presentado utilice la ABI explicada en clase([2] y [3]).
- **Syscalls**
  Se aclara que desde el código assembly no se llaman funciones que no son escritas originalmente en assembly. Por lo contrario, desde el código C sí se invoca código assembly, particularmente se invocan algunos de los system calls disponibles en NetBSD (en particular, ***SYS_read*** y ***SYS_write***).

Como en el Trabajo Práctico #0, el programa se estructura en los siguientes pasos:

- Análisis de las parámetros de la línea de comandos: se analizan las opciones ingresadas por la línea de comandos utilizando la función `getopt_long()`, la cual puede procesar cada opción que es leída de forma simplificada. Se extraen los argumentos de cada opción y se los guarda dentro de una estructura para su posterior acceso del tipo `CommandOptions` cuya definición es

```
1        typedef struct {
2                File input;
3                File output;
```

```
4                   const char* input_route;
5                   const char* output_route;
6                   char error;
7                   char encode_opt;
8           } CommandOptions;
9
```

En caso de que no se encuentre alguna opción, se muestra el mensaje de ayuda al usuario para que identifique el prototipo de cómo debe ejecutar el programa.

- Validación de opciones: a medida que se va analizando cada opción de la línea de comandos, se valida cada una de ellas. Si se ingresó algún parámetro no válido para el programa o si se encuentró un error se lo informa al usuario por pantalla y se aborta la ejecución del programa. Se utiliza para ello se la función `CommandErrArg()` cuyo resultado es:

```
1                   fprintf(stderr, "Invalid Arguments\n");
2                   fprintf(stderr,"Options:\n");
3                   fprintf(stderr,"  -V, --version    Print version and quit.\n");
4                   fprintf(stderr,"  -h, --help       Print this information.\n");
5                   fprintf(stderr,"  -i, --input      Location of the input file.\n
    ");
6                   fprintf(stderr,"  -o, --output     Location of the output file.\
    n");
7                   fprintf(stderr,"  -a, --action     Program action: encode (
    default) or decode.\n");
8                   fprintf(stderr,"Examples:\n");
9                   fprintf(stderr,"  tp0 -a encode -i ~/input -o ~/output\n");
10                  fprintf(stderr,"  tp0 -a decode\n");
11
```

Para el caso en que no hubo errores a la validación de los argumentos se procede a llamar a las funciones correspondientes a:

  - `Mensaje de ayuda:` Función `CommandVersion()`
  - `Mensaje de versión:` Función `CommandHelp()`
  - `Input file :` Función `CommandSetInput()` que guarda la entrada del archivo donde será leído el texto.
  - `Output file:` Función CommandSetOutput() que guarda la entrada del archivo de salida donde se escribirá el texto codificado.
  - `Acción del programa a ejecutar:` Función CommandSetEncodeOpt() que setea la variable $opt- > encode\_opt$ indicando si es una operación de ENCODE o DECODE respectivamente.

- Encode/Decode: una vez que se procesó correctamente las opciones de la línea de comandos se procede a llamar a las funciones correspondientes que ejecutarán la operación de ENCODE o DECODE dependiendo del argumento pasado en la línea de comandos. Como se especifico más arriba está parte del programa es implementada en lenguaje assembly MIPS y cumplen lo siguientes:

  - DECODE
    La operación de DECODE está implementada en el archivo ***decode.S*** que contiene una función `Decode()` que básicamente lo que realiza es la lectura del archivo para procesarlo

teniendo en cuenta la longitud del archivo a procesar y el padding a decodificar. Esta función recibe los files descriptor de entrada y salida procesándolo, según la ABI requerida y luego en la salida si no hubo errores se retorna cero sino se retorna un código de error númerico .

- ENCODE
  La operación de ENCODE está implementada en el archivo ***encode.S*** que contiene una función `Encode()` que básicamente lo que realiza es la lectura del archivo para procesarlo teniendo en cuenta la longitud del archivo a procesar y el padding a decodificar. Esta función recibe los files descriptor de entrada y salida procesándolo, según la ABI requerida y luego en la salida si no hubo errores se retorna cero sino se retorna un código de error númerico .

1.2. Parámetros del programa

Se detallan a continuación los parámetros del programa

- -h: Visualiza la ayuda del programa, en la que se indican los parámetros y sus objetivos.
- -V: Indica la versión del programa.
- -i: Archivo de entrada del programa.
- -o: Archivo de salida del programa.
- -a: Acción a llevar a cabo: codificación o decodificación.

Se indica a continuación detalles respecto a los parámetros:

- Si no se explicitan -i y -o, se utilizarán stdin y stdout, respectivamente.
- -V es una opción "show and quit". Si se explicita este parámetro, sólo se imprimirá la versión, aunque el resto de los parámetros se hayan explicitado.
- -h también es de tipo "show and quit z se comporta de forma similar a -V.
- en caso de que se use la entrada estándar (con comando echo texto | ./tp0 -a encode) y luego se especifique un archivo de salida con -i, prevalecerá el establecido por parámetro.

1.3. Compilación del programa

Para ejecutarlo, posicionarse en el directorio `src/` y ejecutar el siguiente comando:

```
$ gcc -std=c99 -Wall -o0 -g -o tp1 main.c base64.S
```

Para proceder a la ejecución del programa, se debe llamar a:

```
$ ./tp1
```

seguido de los parámetros que se desee modificar, los cuales se indicaron en la sección 1.2.

En caso de ser entrada estándar (stdin) se podrá ejecutar de la siguiente forma:

```
$ echo texto | ./tp1 -a encode
```

También en este caso, se indican a continuación los parámetros a usar.

2. Pruebas realizadas

2.1. Pruebas con archivo bash test-automatic.sh

Para la ejecución del siguiente script se debe copiar, se debe ubicar el archivo ejecutable compilado dentro de la carpeta de test para que se ejecuten correctamente las pruebas. El script sería:

```
#!/bin/bash

echo "########################################"
echo "########## Tests automaticos  ##########"
echo "########################################"

mkdir ./outputs

echo "#---------# COMIENZA test ejercicio 0 archivo vacio #--------#"
touch ./outputs-aut/zero.txt
./tp1 -a encode -i ./outputs-aut/zero.txt -o ./outputs-aut/zero.txt.b64
ls -l ./outputs-aut/zero.txt.b64

if diff -b ./outputs-aut/zero.txt ./outputs-aut/zero_ok.txt; then
 echo "[OK]";
else echo ERROR;
fi

echo "#---------# FIN test ejercicio 0 archivo vacio #--------#"
echo "#-----------------------------------------------------#"
echo "#---------# COMIENZA test ejercicio 1 archivo vacio sin -a #--------#"

touch ./outputs-aut/zero.txt
./tp1 -i ./outputs-aut/zero.txt -o ./outputs-aut/zero.txt.b64
ls -l ./outputs-aut/zero.txt.b64

if diff -b ./outputs-aut/zero.txt ./outputs-aut/zero_ok.txt; then
 echo "[OK]";
else echo ERROR;
fi

echo "#---------# FIN test ejercicio 1 archivo vacio sin -a #--------#"
echo "#-------------------------------------------------------------#"
echo "#---------# COMIENZA test ejercicio 2 stdin y stdout #---------#"

echo -n Man | ./tp1 -a encode > ./outputs/outputEncode.txt
if diff -b ./outputs-aut/outputEncode-aut.txt ./outputs/outputEncode.txt; then echo
    "[OK]"; else
        echo ERROR;
fi

echo "#---------# FIN test ejercicio 2 stdin y stdout #--------#"
echo "#-----------------------------------------------------#"
echo "#---------# COMIENZA test ejercicio 3 stdin y stdout #--------#"

echo -n TWFu | ./tp1 -a decode > ./outputs/outputDecode.txt
if diff -b ./outputs-aut/outputDecode-aut.txt ./outputs/outputDecode.txt; then echo
    "[OK]"; else
        echo ERROR;
fi
```

```
49
50 echo "#---------# FIN test ejercicio 3 stdin y stdout #--------#"
51 echo "#-----------------------------------------------------#"
52 echo "#---------# COMIENZA test ejercicio 3 help sin parámetros #--------#"
53
54 ./tp1 > ./outputs/outputMenuHelp.txt
55 if diff -b ./outputs-aut/outputMenuHelp-aut.txt ./outputs/outputMenuH.txt; then echo
       "[OK]"; else
56         echo ERROR;
57 fi
58
59 echo "#---------# FIN test ejercicio 3 help sin parámetros #--------#"
60 echo "#-----------------------------------------------------#"
61 echo "#----------# COMIENZA test menu help (-h) #----------#"
62
63 ./tp1 -h > ./outputs/outputMenuH.txt
64
65 if diff -b ./outputs-aut/outputMenuHelp-aut.txt ./outputs/outputMenuH.txt; then echo
       "[OK]"; else
66         echo ERROR;
67 fi
68
69 echo "#----------# FIN test menu version (-h) #----------#"
70 echo "#-----------------------------------------------------#"
71 echo "#----------# COMIENZA test menu help (--help) #----------#"
72
73 ./tp1 --help > ./outputs/outputMenuHelp.txt
74
75 if diff -b ./outputs-aut/outputMenuHelp-aut.txt ./outputs/outputMenuHelp.txt; then
       echo "[OK]"; else
76                 echo ERROR;
77 fi
78
79 echo "#----------# FIN test menu version (--help) #----------#"
80 echo "#-----------------------------------------------------#"
81 echo "#----------# COMIENZA test menu version (-V) #----------#"
82
83 ./tp1 -V > ./outputs/outputMenuV.txt
84
85 if diff -b ./outputs-aut/outputMenuVersion-aut.txt ./outputs/outputMenuV.txt; then
       echo "[OK]"; else
86                 echo ERROR;
87 fi
88 echo "#----------# FIN test menu version (-V) #----------#"
89 echo "#-----------------------------------------------------#"
90 echo "#----------# COMIENZA test menu version (--version) #----------#"
91
92 ./tp1 --version > ./outputs/outputMenuVersion.txt
93
94 if diff -b ./outputs-aut/outputMenuVersion-aut.txt ./outputs/outputMenuVersion.txt;
       then echo "[OK]"; else
95                 echo ERROR;
96 fi
97 echo "#----------# FIN test menu version (--version) #----------#"
98 echo "#-----------------------------------------------------#"
99 echo "#---------# COMIENZA test ejercicio encode/decode #--------#"
100
101 echo xyz | ./tp1 -a encode | ./tp1 -a decode | od -t c
```

```
102
103 echo "#---------# FIN test ejercicio encode #--------#"
104 echo "#-----------------------------------------------------#"
105 echo "#---------# COMIENZA test ejercicio longitud maxima 76 #--------#"
106
107 yes | head -c 1024 | ./tp1 -a encode > ./outputs/outputSize76.txt
108
109 if diff -b ./outputs-aut/outputSize76-aut.txt ./outputs/outputSize76.txt; then echo
    "[OK]"; else
110                 echo ERROR;
111 fi
112
113 echo "#---------# FIN test ejercicio longitud maxima 76 #--------#"
114 echo "#-----------------------------------------------------#"
115 echo "#---------# COMIENZA test ejercicio decode 1024 #--------#"
116
117 yes | head -c 1024 | ./tp1 -a encode | ./tp1 -a decode | wc -c > ./outputs/
    outputSize1024.txt
118
119 if diff -b ./outputs-aut/outputSize1024-aut.txt ./outputs/outputSize1024.txt; then
    echo "[OK]"; else
120                 echo ERROR;
121 fi
122
123 echo "#---------# FIN test ejercicio decode 1024#--------#"
124 echo "#-----------------------------------------------------#"
125 echo "#---------# COMIENZA test ejercicio encode/decode random #--------#"
126
127 n=1;
128 while :; do
129 #while [$n -lt 10]; do
130 head -c $n </dev/urandom >/tmp/in.bin;
131 ./tp1 -a encode -i /tmp/in.bin -o /tmp/out.b64;
132 ./tp1 -a decode -i /tmp/out.b64 -o /tmp/out.bin;
133 if diff /tmp/in.bin /tmp/out.bin; then :; else
134 echo ERROR: $n;
135 break;
136 fi
137 echo [OK]: $n;
138 n=`expr $n + 1`;
139 rm -f /tmp/in.bin /tmp/out.b64 /tmp/out.bin
140 done
141
142 echo "#---------# FIN test ejercicio encode/decode random #--------#"
143 echo "#-----------------------------------------------------#"
144
145 echo "##########################################"
146 echo "####### FIN Tests automaticos  ###########"
147 echo "##########################################"
```

El cual no presenta errores en ninguna de las corridas llevadas a cabo.

Todas las pruebas que se presentan a continuación, están codificadas en los archivos de prueba ***.txt de forma que puedan ejecutarse y comprobar los resultados obtenidos.

Se indicaran a continuación lo siguiente: comandos para ejecutarlas, líneas de código que las componen y resultado esperado.

2.1.1. Generales

- Mensaje de ayuda

```
$ ./tp1 -h  o ./tp1 --help

Options:
  -V, --version    Print version and quit.
  -h, --help       Print this information.
  -i, --input      Location of the input file.
  -o, --output     Location of the output file.
  -a, --action     Program action: encode (default) or decode.
Examples:
  tp1 -a encode -i ~/input -o ~/output
  tp1 -a decode
```

- Mensaje de version

```
$ ./tp1 -V  o ./tp1 --version
Version: 0.2

```

- Archivo de entrada no válido

```
$ ./tp1 -i archivoInvalido.txt

Invalid Arguments
Options:
  -V, --version    Print version and quit.
  -h, --help       Print this information.
  -i, --input      Location of the input file.
  -o, --output     Location of the output file.
  -a, --action     Program action: encode (default) or decode.
Examples:
  tp1 -a encode -i ~/input -o ~/output
  tp1 -a decode


```

3. Conclusiones

El trabajo práctico nos permitió desarrollar una API para procesar archivos transformándolos a su equivalente `base64` en lenguaje C y, en parte, en lenguaje assembly MIPS para la codificación y decodificación de los archivos. Además, nos permitió familiarizarnos con las `syscalls` para el llamado de las funciones en lenguaje assembly y el consecuente análisis y desarrollo de código assembler MIPS utilizando el emulador GXemul.

Referencias

[1] Enunciado del primer trabajo práctico (TP0), primer cuatrimestre de 2018.

[2] Base64 (Wikipedia) http://en.wikipedia.org/wiki/Base64

[3] The NetBSD project, http://www.netbsd.org/

[4] Kernighan, B. W. - Ritchie, D. M. - *C Programming Language* - 2nd edition - Prentice Hall - 1988.

[5] *GNU Make* - https://www.gnu.org/software/make/

[6] *Valgrind* - http://valgrind.org/

[7] MIPS ABI: Function Calling, Convention Organización de computadoras(66.20) en archivo "func call conv.pdf" y enlace http://groups.yahoo.com/groups/orga-comp/Material/)

[8] System V application binary interface, MIPS RISC processor supplement (third edition). Santa Cruz Operations, Inc.

A. Código fuente

A.0.1. main.c

```
/**
 * Created by gatti2602 on 12/09/18.
 * Main
 */

#define FALSE 0
#define TRUE 1

#include <getopt.h>
#include <string.h>
#include <stdlib.h>
#include <errno.h>
#include <stdio.h>

#define CMD_ENCODE 1
#define CMD_DECODE 0
#define CMD_NOENCODE 2
#define FALSE 0
#define TRUE 1
#define ERROR 1
#define OK 0

#include "base64.h"

/*****************************
 * DECLARACION DE FUNCIONES *
 *****************************/

typedef struct{
    FILE* file;
    char eof;
} File;

typedef struct {
    File input;
    File output;
    const char* input_route;
    const char* output_route;
    char error;
    char encode_opt;
} CommandOptions;

/**
 * Inicializa TDA CommandOptions
 * Pre: Puntero a Command Options escribible
 * Post: CommandOptions Inicializados a valores por default
 * Valores default:
 *      input: stdin
 *      output stdout
 *      error: FALSE
 *      encode_opt: decode
 */
void CommandCreate(CommandOptions* opt);
```

```
54
55 /**
56  * Setea ruta de entrada
57  * Pre: ruta valida
58  * Post: ruta lista para abrir file
59  */
60 void CommandSetInput(CommandOptions* opt, const char* input);
61
62 /**
63  * Setea ruta de salida
64  * Pre: ruta valida
65  * Post: ruta lista para abrir file
66  */
67 void CommandSetOutput(CommandOptions* opt, const char* output);
68
69 /**Setea Command Option
70  * Pre: opt inicializado
71  * Post: Setea el encoding.
72  *       Si string no es encode/decode setea opt error flag.
73  */
74 void CommandSetEncodeOpt(CommandOptions* opt, const char* encode_opt);
75
76 /**
77  * Devuelve el flag de error
78  */
79 char CommandHasError(CommandOptions *opt);
80
81 /**
82  * Indica que hubo un error
83  */
84 void CommandSetError(CommandOptions *opt);
85
86 /**
87  * Ejecuta el comando
88  * Pre: Asume parametros previamente validados y ok
89  * Post: Ejecuta el comando generando la salida esperada
90  *        Devuelve 0 si error y 1 si OK.
91  */
92 char CommandProcess(CommandOptions* opt);
93
94 /**
95  * Help Command
96  * Imprime por salida estandar los distintos comandos posibles.
97  * Pre: N/A
98  * Post: N/A
99  */
100 void CommandHelp();
101
102 /**
103  * Imprime la ayuda por la salida de errores
104  */
105 void CommandErrArg();
106
107 /**
108  * Version Command
109  * Imprime por salida estandar la version del codigo
110  * Pre: N/A
111  * Post: N/A
```

```
112  */
113 void CommandVersion();
114 
115 /**
116  * Recibe los archivos abiertos y debe ejecutar la operacion de codificacion
117  * Pre: opt->input posee el stream de entrada
118  *      opt->output posee el stream de salida
119  *      opt->encode_opt posee la opcion de codificacion
120  * Post: Datos procesados y escritos en el stream, si error devuelve 0, sino 1.
121  */
122 char _CommandEncodeDecode(CommandOptions *opt);
123 
124 /**
125  * Construye el TDA.
126  * Post: TDA construido
127  */
128 void FileCreate(File *f);
129 
130 /**
131  * Abre un File, devuelve 0 (NULL) si falla
132  * Pre: Ptr a File Inicializado ,
133  *      Ruta a archivo, si es 0 (NULL) utiliza stdin
134  */
135 char FileOpenForRead(File* file, const char* route);
136 
137 /**
138  * Abre un File, devuelve 0 (NULL) si falla
139  * Pre: Ptr a File Inicializado ,
140  *      Ruta a archivo, si es 0 (NULL) utiliza stdout
141  */
142 char FileOpenForWrite(File* file, const char* route);
143 
144 /*
145  * Cierra archivo abierto
146  * Pre: Archivo previamente abierto
147  */
148 int FileClose(File* file);
149 
150 /*********************************
151  * FIN: DECLARACION DE FUNCIONES *
152  *********************************/
153 
154 /*************************
155  * DEFINICION DE FUNCIONES *
156  *************************/
157 
158 void CommandHelp(){
159     printf("Options:\n");
160     printf("  -V, --version    Print version and quit.\n");
161     printf("  -h, --help       Print this information.\n");
162     printf("  -i, --input      Location of the input file.\n");
163     printf("  -o, --output     Location of the output file.\n");
164     printf("  -a, --action     Program action: encode (default) or decode.\n");
165     printf("Examples:\n");
166     printf("  tp0 -a encode -i ~/input -o ~/output\n");
167     printf("  tp0 -a decode\n");
168 }
169 
```

```
170 void CommandVersion() {
171     printf("Version: 0.2\n");
172 }
173
174 void CommandCreate(CommandOptions *opt) {
175     FileCreate(&opt->input);
176     FileCreate(&opt->output);
177     opt->error = FALSE;
178     opt->encode_opt = CMD_ENCODE;
179     opt->input_route = 0;
180     opt->output_route = 0;
181 }
182
183 void CommandSetInput(CommandOptions *opt, const char *input) {
184     opt->input_route = input;
185 }
186
187 void CommandSetOutput(CommandOptions *opt, const char *output) {
188     opt->output_route = output;
189 }
190
191 void CommandSetEncodeOpt(CommandOptions *opt, const char *encode_opt) {
192         if(strcmp(encode_opt,"decode") == 0) {
193             opt->encode_opt = CMD_DECODE;
194         } else {
195             opt->encode_opt = CMD_ENCODE;
196         }
197 }
198
199 char CommandHasError(CommandOptions *opt) {
200     return opt->error || opt->encode_opt == CMD_NOENCODE;
201 }
202
203 void CommandSetError(CommandOptions *opt) {
204     opt->error = TRUE;
205 }
206
207 char CommandProcess(CommandOptions *opt) {
208     opt->error = FileOpenForRead(&opt->input, opt->input_route);
209
210     if(opt->error != ERROR){
211         opt->error = FileOpenForWrite(&opt->output, opt->output_route);
212
213         if(opt->error != ERROR){
214                 opt->error = _CommandEncodeDecode(opt);
215                 FileClose(&opt->input);
216                 FileClose(&opt->output);
217         } else {
218                 FileClose(&opt->input);
219         }
220     }
221     return opt->error;
222 }
223
224 char _CommandEncodeDecode(CommandOptions *opt) {
225     if(opt->encode_opt == CMD_ENCODE){
226         int filein = fileno((opt->input).file);
227         int fileout = fileno((opt->output).file);
```

```
228         int res = base64_encode(filein, fileout);
229         if(res != 0)
230                 fprintf(stderr, "%s\n",errmsg[res]);
231 
232     }
233 
234    if (opt->encode_opt == CMD_DECODE) {
235         int filein = fileno((opt->input).file);
236         int fileout = fileno((opt->output).file);
237         int res = base64_decode(filein, fileout);
238         if(res != 0)
239                 fprintf(stderr, "%s\n",errmsg[res]);
240    }
241 
242     return opt->error;
243 }
244 
245 void CommandErrArg() {
246     fprintf(stderr, "Invalid Arguments\n");
247     fprintf(stderr,"Options:\n");
248     fprintf(stderr,"  -V, --version    Print version and quit.\n");
249     fprintf(stderr,"  -h, --help       Print this information.\n");
250     fprintf(stderr,"  -i, --input      Location of the input file.\n");
251     fprintf(stderr,"  -o, --output     Location of the output file.\n");
252     fprintf(stderr,"  -a, --action     Program action: encode (default) or decode.\n"
    );
253     fprintf(stderr,"Examples:\n");
254     fprintf(stderr,"  tp0 -a encode -i ~/input -o ~/output\n");
255     fprintf(stderr,"  tp0 -a decode\n");
256 }
257 
258 void FileCreate(File *file){
259     file->file = 0;
260     file->eof = 0;
261 }
262 
263 char FileOpenForRead(File* file, const char *route ){
264     if(route == NULL) {
265         file->file = stdin;
266     } else {
267         file->file = fopen(route, "rb");
268         if (file->file == NULL) {
269             int err = errno;
270             fprintf(stderr, "File Open Error; %s\n", strerror(err));
271             return ERROR;
272         }
273     }
274     return OK;
275 }
276 
277 char FileOpenForWrite(File* file, const char *route ) {
278     if(route == NULL) {
279         file->file = stdout;
280     } else {
281         file->file = fopen(route, "wb");
282         if (file->file == NULL) {
283             int err = errno;
284             fprintf(stderr, "File Open Error; %s\n", strerror(err));
```

```
285             return ERROR;
286         }
287     }
288     return OK;
289 }
290 
291 int FileClose(File* file){
292     if(file->file == stdin || file->file == stdout)
293         return OK;
294 
295     int result = fclose(file->file);
296     if (result == EOF){
297         int err = errno;
298         fprintf(stderr, "File Close Error; %s\n", strerror(err));
299         return ERROR;
300     }
301     return OK;
302 }
303 
304 /*******************************
305  * FIN: DEFINICION DE FUNCIONES *
306  *******************************/
307 
308 int main(int argc, char** argv) {
309     struct option arg_long[] = {
310             {"input",   required_argument,  NULL,  'i'},
311             {"output",  required_argument,  NULL,  'o'},
312             {"action",  required_argument,  NULL,  'a'},
313             {"help",    no_argument,        NULL,  'h'},
314             {"version", no_argument,        NULL,  'V'},
315     };
316     char arg_opt_str[] = "i:o:a:hV";
317     int arg_opt;
318     int arg_opt_idx = 0;
319     char should_finish = FALSE;
320 
321     CommandOptions cmd_opt;
322     CommandCreate(&cmd_opt);
323 
324     if(argc == 1)
325         CommandSetError(&cmd_opt);
326 
327     while((arg_opt =
328                     getopt_long(argc, argv, arg_opt_str, arg_long, &arg_opt_idx)) !=
    -1 && !should_finish) {
329         switch(arg_opt){
330                 case 'i':
331                         CommandSetInput(&cmd_opt, optarg);
332                         break;
333                 case 'o':
334                 CommandSetOutput(&cmd_opt, optarg);
335                 break;
336                 case 'h':
337                         CommandHelp();
338                 should_finish = TRUE;
339                 break;
340                 case 'V':
341                         CommandVersion();
```

```
342                 should_finish = TRUE;
343                 break;
344                 case 'a':
345                     CommandSetEncodeOpt(&cmd_opt, optarg);
346                                 break;
347                 default:
348                         CommandSetError(&cmd_opt);
349                         break;
350         }
351     }
352
353     if(should_finish)
354         return 0;
355
356     if(!CommandHasError(&cmd_opt)) {
357         CommandProcess(&cmd_opt);
358     } else {
359         CommandErrArg();
360         return 1;
361     }
362     return 0;
363 }
```

A.0.2. Header file base64.h

```
#ifndef TP1_BASE64_H
#define TP1_BASE64_H

extern const char* errmsg[];

int base64_encode(int infd, int outfd);
int base64_decode(int infd, int outfd);

#endif
```

A.0.3. Assembly base64.S

```
#include <mips/regdef.h>
#include <sys/syscall.h>
#define STACK_FRAME_ENCODE 16

#define OFFSET_OUTPUT_ENCODE 24
#define OFFSET_LENGTH_ENCODE 20
#define OFFSET_BUFFER_ENCODE 16
#define OFFSET_FP_ENCODE 12
#define OFFSET_GP_ENCODE 8

#define OFFSET_B4_AUX 7
#define OFFSET_B3_AUX_2 6
#define OFFSET_B3_AUX 5
#define OFFSET_B2_AUX 4
#define OFFSET_B1_AUX 3
#define OFFSET_B3 2
#define OFFSET_B2 1
#define OFFSET_B1 0
#define EQUAL_CHAR 61

#define RETURNO_OK 1
#define DECODE_ERROR   100
#define SIZE_DECODE_CHAR 4

#define SHIFT_2 2
#define SHIFT_4 4
#define SHIFT_6 6

#define EQUAL_CHAR 61

#define STACK_FRAME_DECODECHAR 32

#define OFFSET_FP_DECODECHAR 32
#define OFFSET_GP_DECODECHAR 28
#define OFFSET_CHARACTER_DECODECHAR 16
#define OFFSET_I_DECODECHAR 20
#define OFFSET_RETURN_DECODECHAR 24

#define STACK_FRAME_DECODE 64

#define OFFSET_BUFFER_OUTPUT_ENCODE 68
#define OFFSET_BUFFER_INPUT_ENCODE 64
#define OFFSET_RA_DECODE 60
#define OFFSET_FP_DECODE 56
#define OFFSET_GP_DECODE 52
#define OFFSET_S0_DECODE 48
#define OFFSET_CHAR1_AUX_ENCODE 37
#define OFFSET_CHAR0_AUX_ENCODE 36
#define OFFSET_CHARS3_ENCODE 27
#define OFFSET_CHARS2_ENCODE 26
#define OFFSET_CHARS1_ENCODE 25
#define OFFSET_CHARS0_ENCODE 24
#define OFFSET_RETURN_ENCODE 20
#define OFFSET_I_DECODE 32

        .data
```

```
        .align  2
sep:    .ascii  "\n"
pad:    .ascii  "="
        .globl errmsg
errmsg:
        .word base64_ok, base64_err1, base64_err2, base64_err3
        .size errmsg, 16
base64_ok:
        .asciiz "OK"
base64_err1:
        .asciiz "I/O Error"
base64_err2:
        .asciiz "File no es multiplo de 4"
base64_err3:
        .asciiz "File contiene caracteres invalidos"
        .text
        .align  2
        .globl  base64_encode
        .ent    base64_encode
base64_encode:
        // debugging info: descripcion del stack frame
        .frame  $fp, 40, ra     // $fp: registro usado como frame pointer
                                // 32: tamañodel stack frame
                                // ra: registro que almacena el return address
        // bloque para codigo PIC
        .set    noreorder       // apaga reordenamiento de instrucciones
        .cpload t9              // directiva usada para codigo PIC
        .set    reorder         // enciende reordenamiento de instrucciones
        // creo stack frame
        subu    sp, sp, 40      // 4 (SRA) + 2 (LTA) + 4 (ABA)
        // directiva para codigo PIC
        .cprestore 24           // inserta aqui "sw gp, 24(sp)",
                                // mas "lw gp, 24(sp)" luego de cada jal.
        // salvado de callee-saved regs en SRA
        sw      $fp, 28(sp)
        sw      ra, 32(sp)
        // de aqui al fin de la funcion uso $fp en lugar de sp.
        move    $fp, sp
        // salvo 1er arg (siempre)
        sw      a0, 40($fp)     // a0 contiene file input
        sw      a1, 44($fp)     // a1 contiene file output
        li      s1, 0           // count = 0

        //Limpio input para read
base64_encode_loop:
        sw      zero, 20($fp)   //input = 0

        //Leo archivo
        lw      a0, 40($fp)
        addi    a1, $fp, 20
        li      a2, 3
        li      v0, SYS_read
        syscall
        beqz    v0, base64_encode_return_ok     //Si no lei nada finalizo
        blt     v0, 0, base64_encode_io_error
        //Paso parametros y llamo a Encode
        addi    a0, $fp, 20
        move    a1, v0
```

```
        addi    a2, $fp, 16
        la      t9, Encode
        jal     ra, t9

        //Grabo en file
        lw      a0, 44($fp)     // File descriptor out
        addi    a1, $fp, 16     // Apunto a buffer out
        li      a2, 4           // length = 4
        li      v0, SYS_write
        syscall
        addi s1, s1, 1          // count++
        bne s1, 18, base64_encode_loop // Si count = 18 agrego un salto
        lw      a0, 44($fp)     // file out
        la      a1, sep         // sep = '\n'
        li      a2, 1           // length = 4
        li      v0, SYS_write
        syscall
        li      s1, 0
        j base64_encode_loop

base64_encode_return_ok:        // return;
        li      v0, 0
        j base64_encode_return
base64_encode_io_error:
        li      v0, 1
        // restauro callee-saved regs
base64_encode_return:
        lw      gp, 24(sp)
        lw      $fp, 28(sp)
        lw      ra, 32(sp)
        // destruyo stack frame
        addu    sp, sp, 40
        // vuelvo a funcion llamante
        jr      ra
        .end    base64_encode
        .size   base64_encode, .-base64_encode

        .globl  base64_decode
        .ent    base64_decode
base64_decode:
        // debugging info: descripcion del stack frame
        .frame  $fp, 40, ra     // $fp: registro usado como frame pointer
                                // 32: tamañodel stack frame
                                // ra: registro que almacena el return address
        // bloque para codigo PIC
        .set    noreorder       // apaga reordenamiento de instrucciones
        .cpload t9              // directiva usada para codigo PIC
        .set    reorder         // enciende reordenamiento de instrucciones
        // creo stack frame
        subu    sp, sp, 40      // 4 (SRA) + 2 (LTA) + 4 (ABA)
        // directiva para codigo PIC
        .cprestore 24           // inserta aqui "sw gp, 24(sp)",
                                // mas "lw gp, 24(sp)" luego de cada jal.
        // salvado de callee-saved regs en SRA
        sw      $fp, 28(sp)
        sw      ra, 32(sp)
        // de aqui al fin de la funcion uso $fp en lugar de sp.
        move    $fp, sp
```

```
        // salvo 1er arg (siempre)
        sw      a0, 40($fp)     // a0 contiene file input
        sw      a1, 44($fp)     // a1 contiene file output
        li      s1, 0           // count = 0
        la      s5, pad

        //Limpio input para read
base64_decode_loop:
        sw      zero, 20($fp)   //input = 0

        //Leo archivo
        lw      a0, 40($fp)
        addi    a1, $fp, 20
        li      a2, 4
        li      v0, SYS_read
        syscall
        beqz    v0, base64_decode_return_ok     //Si no lei nada finalizo
        blt     v0, 0, base64_decode_ioerror
        blt     v0, 4, base64_decode_nomult
        //Controlo si hay padding
        li      s3, 0                           //s3 = cant de padding a borrar
        lbu     s2, 43($fp)                     //s2 aux control padding
        bne     s2, s5, ctl1
        addi    s3, s3, 1
ctl1:
        lbu     s2, 42($fp)                     //s2 aux control padding
        bne     s2, s5, ctl2
        addi    s3, s3, 1
ctl2:
        //Controlo salto de linea
        addi s1, s1, 1          // count++
        bne s1, 18, not_sep     // Si count = 18 elimino un caracter
        lw      a0, 40($fp)     // file in
        addi    a1, $fp, 16     // grabo en out buffer, luego se pisa
        li      a2, 1           // length = 1
        li      v0, SYS_read
        syscall
        li      s1, 0
        //Paso parametros y llamo a Decode
not_sep:
        addi    a0, $fp, 20
        addi    a1, $fp, 16
        la      t9, Decode
        jal     ra, t9

        //Chequeo error
        beq     v0, DECODE_ERROR, base64_decode_decode_err

        //Grabo en file
        lw      a0, 44($fp)     // File descriptor out
        addi    a1, $fp, 16     // Apunto a buffer out
        li      s4, 3
        subu    a2, s4, s3      // a2 = 3 - cant de padding
        li      v0, SYS_write
        syscall
        j base64_decode_loop
base64_decode_return_ok:
        li v0, 0
```

```
231         j base64_decode_return
232 base64_decode_ioerror:
233         li v0, 1
234         j base64_decode_return
235 base64_decode_nomult:
236         li v0, 2
237         j base64_decode_return
238 base64_decode_decode_err:
239         li v0, 3
240 base64_decode_return:   // return;
241         // restauro callee-saved regs
242         lw      gp, 24(sp)
243         lw      $fp, 28(sp)
244         lw      ra, 32(sp)
245         // destruyo stack frame
246         addu    sp, sp, 40
247         // vuelvo a funcion llamante
248         jr      ra
249         .end    base64_decode
250         .size   base64_decode, .-base64_decode
251
252         //.file 1 "encode.c"
253         //.section .mdebug.abi32
254         //.previous
255         //.abicalls
256         .data
257         .align  2
258         .type   encoding_table, @object
259         .size   encoding_table, 64
260 encoding_table:
261         .byte   65
262         .byte   66
263         .byte   67
264         .byte   68
265         .byte   69
266         .byte   70
267         .byte   71
268         .byte   72
269         .byte   73
270         .byte   74
271         .byte   75
272         .byte   76
273         .byte   77
274         .byte   78
275         .byte   79
276         .byte   80
277         .byte   81
278         .byte   82
279         .byte   83
280         .byte   84
281         .byte   85
282         .byte   86
283         .byte   87
284         .byte   88
285         .byte   89
286         .byte   90
287         .byte   97
288         .byte   98
```

```
289         .byte   99
290         .byte   100
291         .byte   101
292         .byte   102
293         .byte   103
294         .byte   104
295         .byte   105
296         .byte   106
297         .byte   107
298         .byte   108
299         .byte   109
300         .byte   110
301         .byte   111
302         .byte   112
303         .byte   113
304         .byte   114
305         .byte   115
306         .byte   116
307         .byte   117
308         .byte   118
309         .byte   119
310         .byte   120
311         .byte   121
312         .byte   122
313         .byte   48
314         .byte   49
315         .byte   50
316         .byte   51
317         .byte   52
318         .byte   53
319         .byte   54
320         .byte   55
321         .byte   56
322         .byte   57
323         .byte   43
324         .byte   47
325
326         .type   encoding_table_size, @object
327         .size   encoding_table_size, 4
328 encoding_table_size:
329         .word   64
330
331         .text
332         .align  2
333         .globl  Encode
334         .ent    Encode
335
336         //////////////  Función Encode //////////////
337
338 Encode:
339         .frame  $fp,STACK_FRAME_ENCODE,ra               // vars= 8, regs= 2/0, args=
        0, extra= 8
340         //.mask 0x50000000,-4
341         //.fmask        0x00000000,0
342         .set    noreorder
343         .cpload t9
344         .set    reorder
345
```

```
346         // Creación del stack frame
347         subu    sp,sp,STACK_FRAME_ENCODE
348
349         .cprestore 0
350         sw      $fp,OFFSET_FP_ENCODE(sp)
351         sw      gp,OFFSET_GP_ENCODE(sp)
352
353         // De aquí al final de la función uso $fp en lugar de sp.
354         move    $fp,sp
355
356         // Guardo el primer parámetro *buffer
357         sw      a0,OFFSET_BUFFER_ENCODE($fp)
358         // Guardo el segundo parámetro 'length'(cantidad de caracteres)
359         sw      a1,OFFSET_LENGTH_ENCODE($fp)
360         // Guardo el puntero al array de salida(output)
361         sw      a2,OFFSET_OUTPUT_ENCODE($fp)
362
363         // Cargo en v0 el puntero al buffer.
364         lw      v0,OFFSET_BUFFER_ENCODE($fp)
365         // Cargo en v0 el 1er byte del buffer.
366         lbu     v0,0(v0)
367         // Guardo el 1er byte en el stack frame
368         sb      v0,OFFSET_B1($fp)
369         // Cargo nuevamente la dirección del buffer.
370         lw      v0,OFFSET_BUFFER_ENCODE($fp)
371         // Aumento en 1(1 byte) la dirección del buffer.
372         // Me muevo por el array del buffer.
373         addu    v0,v0,1
374         // Cargo el 2do byte del buffer.
375         lbu     v0,0(v0)
376         // Guardo el 2do byte en el stack frame.
377         sb      v0,OFFSET_B2($fp)
378         // Cargo nuevamente la dirección del buffer.
379         lw      v0,OFFSET_BUFFER_ENCODE($fp)
380         // Aumento en 2(2 byte) la dirección del buffer.
381         // Me muevo por el array del buffer.
382         addu    v0,v0,2
383         // Cargo el 2do byte del buffer.
384         lbu     v0,0(v0)
385         // Guardo el 3er byte en stack frame.
386         sb      v0,OFFSET_B3($fp)
387         // Cargo en v0 el 1er byte.
388         lbu     v0,OFFSET_B1($fp)
389         // Muevo 2 'posiciones' hacia la derecha(shift 2).
390         srl     v0,v0,2
391         // Guardo el nuevo byte en una variable auxiliar.
392         sb      v0,OFFSET_B1_AUX($fp)
393         // Cargo en v1 el puntero al output.
394         lw      v1,OFFSET_OUTPUT_ENCODE($fp)
395         // Cargo en v0 el byte shifteado.
396         lbu     v0,OFFSET_B1_AUX($fp)
397         // Cargo en v0 el caracter(byte) de la tabla encoding(encoding_table)
398         lbu     v0,encoding_table(v0)
399         // Cargo en v0 el 1er byte de la dirección del output.
400         sb      v0,0(v1)
401         // Cargo en v0 el 1er byte del buffer nuevamente.
402         lbu     v0,OFFSET_B1($fp)
403         // Muevo 6 'posiciones' hacia la izquierda(shift 6).
```

```
404         sll     v0,v0,6
405         // Guardo el resultado del shift en el Stack Frame.
406         sb      v0,OFFSET_B2_AUX($fp)
407         // Cargo el byte sin signo shifteado.
408         lbu     v0,OFFSET_B2_AUX($fp)
409         // Muevo 2 'posiciones' hacia la derecha(shift 2).
410         srl     v0,v0,2
411         // Guardo el nuevo resultado del shift en el Stack Frame.
412         sb      v0,OFFSET_B2_AUX($fp)
413         // Cargo el 2do byte del buffer en v0.
414         lbu     v0,OFFSET_B2($fp)
415         // Hago un shift left de 4 posiciones.
416         srl     v0,v0,4
417         // Cargo en v1 el resultado(byte) del shift right 2.
418         lbu     v1,OFFSET_B2_AUX($fp)
419         // Hago un 'or' entre v1 y v0 para obtener el 2 indice de la tabla.
420         or      v0,v1,v0
421         //(*) Guardo en stack frame(12) el resultado del 'or' anterior.
422         sb      v0,OFFSET_B2_AUX($fp)
423         // Cargo en v0 el puntero al output.
424         lw      v0,OFFSET_OUTPUT_ENCODE($fp)
425         // Cargo en v1 la dirección del output + 1(1byte).
426         addu    v1,v0,1
427         // Cargo en v0 el ultimo resultado del shift(*)
428         lbu     v0,OFFSET_B2_AUX($fp)
429         // Cargo en v0 el caracter(byte) de la tabla encoding(encoding_table)
430         lbu     v0,encoding_table(v0)
431         // Salvo en el output array(output[1]) el valor del encoding_table
432         sb      v0,0(v1)
433         // Cargo en v0 el puntero al output.
434         lw      v0,OFFSET_OUTPUT_ENCODE($fp)
435         // Sumo 2 a la dirección del output(output[2]).
436         // Me desplazo dentro del output array.
437         addu    v1,v0,2
438         // Cargo en v0 el caracter ascii 61('=').
439         li      v0,EQUAL_CHAR                   // 0x3d
440         // Salvo en el output array(output[2]) el valor '='.
441         sb      v0,0(v1)
442         // Cargo en v0 el puntero al output.
443         lw      v0,OFFSET_OUTPUT_ENCODE($fp)
444         // Sumo 3 a la dirección del output(output[3]).
445         // Me desplazo dentro del output array.
446         addu    v1,v0,3
447         // Cargo en v0 el caracter ascii 61('=').
448         li      v0,EQUAL_CHAR                   // 0x3d
449         // Salvo en el output array(output[3]) el valor '='.
450         sb      v0,0(v1)
451         // Cargo en v1 el parametro length.
452         lw      v1,OFFSET_LENGTH_ENCODE($fp)
453         // Cargo en v0 el valor 3.
454         li      v0,3                    // 0x3
455         // Si el length == 3 salto a buffer_size_2.
456         bne     v1,v0,buffer_size_2
457         // Si el tamanio del buffer es 3 continuo NO salto.
458         // Cargo en v0 el 3er byte del buffer.
459         lbu     v0,OFFSET_B3($fp)
460         // Hago un shift right de 6.
461         srl     v0,v0,6
```

```
462         // Guardo el nuevo byte en el stack frame.
463         sb      v0,OFFSET_B3_AUX($fp)
464         // Cargo el 2do byte del buffer en v0.
465         lbu     v0,OFFSET_B2($fp)
466         // Hago un shift left de 4.
467         sll     v0,v0,4
468         // Guardo en el stack frame(14) el nuevo valor.
469         sb      v0,OFFSET_B3_AUX_2($fp)
470         // Cargo en v0 el byte shifteado sin signo.
471         lbu     v0,OFFSET_B3_AUX_2($fp)
472         // Hago un shift rigth de 2.
473         srl     v0,v0,2
474         // Guardo en el stack frame(14) el valor shifteado.
475         sb      v0,OFFSET_B3_AUX_2($fp)
476         // Cargo en v1 el valor del SF(13)
477         lbu     v1,OFFSET_B3_AUX($fp)
478         // Idem en v0(13).
479         lbu     v0,OFFSET_B3_AUX_2($fp)
480         // Hago un 'or' y almaceno en v0.
481         or      v0,v1,v0
482         // Guardo en el stack frame(13) el resultado del 'or'.
483         sb      v0,OFFSET_B3_AUX($fp)
484         // Cargo en v0 el puntero al output.
485         lw      v0,OFFSET_OUTPUT_ENCODE($fp)
486         // Me desplazo por el vector 'output' en 2 posiciones(output[2]).
487         addu    v1,v0,2
488         // Cargo en v0 el resultado del 'or' anterior.
489         lbu     v0,OFFSET_B3_AUX($fp)
490         // Busco en la tabla de encoding el caracter que corresponde.
491         // Luego cargo el byte en v0.
492         lbu     v0,encoding_table(v0)
493         // Guardo el valor recuperado de la tabla encoding_table en el output[2].
494         sb      v0,0(v1)
495         // Cargo en v0 el 3er byte del buffer.
496         lbu     v0,OFFSET_B3($fp)
497         // Hago un shift left de 2.
498         sll     v0,v0,2
499         // Guardo en el stack frame el valor shifteado.
500         sb      v0,OFFSET_B4_AUX($fp)
501         // Cargo el byte sin signo shifteado.
502         lbu     v0,OFFSET_B4_AUX($fp)
503         // Hago un shift rigth de 2.
504         srl     v0,v0,2
505         // Guardo en el stack frame el valor shifteado.
506         sb      v0,OFFSET_B4_AUX($fp)
507         // Cargo en v0 el puntero al output.
508         lw      v0,OFFSET_OUTPUT_ENCODE($fp)
509         // Sumo 3 a la dirección del output(output[3]).
510         // Me desplazo dentro del output array.
511         addu    v1,v0,3
512         // Cargo en v0 el ultimo valor shifteado guardado.
513         lbu     v0,OFFSET_B4_AUX($fp)
514         // Busco en la tabla de encoding el caracter que corresponde.
515         // Luego cargo el byte en v0.
516         lbu     v0,encoding_table(v0)
517         // Guardo el valor recuperado de la tabla encoding_table en el output[3].
518         sb      v0,0(v1)
519         // Salto a return_encode
```

```
520         b       return_encode
521 buffer_size_2:
522         // Cargo en v1 el valor del parámetro length.
523         lw      v1,OFFSET_LENGTH_ENCODE($fp)
524         // Cargo en v0 el valor 2.
525         li      v0,2                    // 0x2
526         // Si length != 2 salgo de la función.
527         bne     v1,v0,return_encode
528         // Cargo en v0 el 3er byte del buffer.
529         lbu     v0,OFFSET_B3($fp)
530         // Hago un shift right de 6.
531         srl     v0,v0,6
532         // Guardo en el stack frame el ultimo valor shifteado.
533         sb      v0,OFFSET_B4_AUX($fp)
534         // Cargo el 2do byte del buffer en v0.
535         lbu     v0,OFFSET_B2($fp)
536         // Hago un shift left de 4 posiciones.
537         sll     v0,v0,4
538         // Guardo en el stack frame nuevo valor shifteado.
539         sb      v0,OFFSET_B3_AUX_2($fp)
540         // Cargo en v0 el byte shifteado sin signo.
541         lbu     v0,OFFSET_B3_AUX_2($fp)
542         // Hago un shift right de 2 posiciones.
543         srl     v0,v0,2
544         // Guardo en el stack frame el valor shifteado.
545         sb      v0,OFFSET_B3_AUX_2($fp)
546         // Cargo en v1 uno de los valores shiftedos(b3aux).
547         lbu     v1,OFFSET_B4_AUX($fp)
548         // Cargo en v0 uno de los valores shiftedos(b3aux2).
549         lbu     v0,OFFSET_B3_AUX_2($fp)
550         // Hago un 'or' entre b3aux y b3aux2.
551         or      v0,v1,v0
552         // Guardo en el stack frame el resutado del 'or'.
553         sb      v0,OFFSET_B4_AUX($fp)
554         // Cargo en v0 el puntero al output.
555         lw      v0,OFFSET_OUTPUT_ENCODE($fp)
556         // Me desplazo dentro del output array y lo guardo en v1.
557         addu    v1,v0,2
558         // Cargo en v0 ultimo resultado del 'or'
559         lbu     v0,OFFSET_B4_AUX($fp)
560         // Busco en la tabla de encoding el caracter que corresponde.
561         // Luego cargo el byte en v0.
562         lbu     v0,encoding_table(v0)
563         // Guardo el valor recuperado de la tabla encoding_table en el output[2].
564         sb      v0,0(v1)
565 return_encode:
566         move    sp,$fp
567         lw      $fp,OFFSET_FP_ENCODE(sp)
568         // destruyo stack frame
569         addu    sp,sp,STACK_FRAME_ENCODE
570         j       ra
571         .end    Encode
572         //.size Encode, .-Encode
573
574         .globl  DecodeChar
575         .ent    DecodeChar
576
577         ////////////// Begin Función DecodeChar //////////////
```

```
578
579 DecodeChar:
580         // Reservo espacio para el stack frame de STACK_FRAME_DECODECHAR bytes
581         .frame  $fp,STACK_FRAME_DECODECHAR,ra           // vars= 8, regs= 2/0, args=
    0, extra= 8
582         //.mask 0x50000000,-4
583         //.fmask        0x00000000,0
584         .set    noreorder
585         .cpload t9
586         .set    reorder
587
588         // Creación del stack frame STACK_FRAME_DECODECHAR
589         subu    sp,sp,STACK_FRAME_DECODECHAR
590         .cprestore 0
591
592         // Guardo fp y gp en el stack frame
593         sw      $fp,OFFSET_FP_DECODECHAR(sp)
594         sw      gp,OFFSET_GP_DECODECHAR(sp)
595         // De aquí al final de la función uso $fp en lugar de sp.
596         move    $fp,sp
597
598         // Guardo en v0 el parámetro recibido: 'character'.
599         move    v0,a0
600         // Guardo en el stack frame 'character'.
601         sb      v0,OFFSET_CHARACTER_DECODECHAR($fp)
602         // Guardo en un '0' en el stack frame.
603         // Inicializo la variable 'i'.
604         sb      zero,OFFSET_I_DECODECHAR($fp)
605 condition_loop:
606         // Cargo en v0 el byte guardado anteriormente(0 o el nuevo valor de 'i').
607         lbu     v0,OFFSET_I_DECODECHAR($fp)
608         // Cargo en v1 el size del encoding_table(64).
609         lw      v1,encoding_table_size
610         // Si (i < encoding_table_size), guardo TRUE en v0, sino FALSE.
611         slt     v0,v0,v1
612         // Salto a condition_if si v0 != 0.
613         bne     v0,zero,condition_if
614         // Brancheo a condition_if_equal
615         b       condition_if_equal
616 condition_if:
617         // Cargo en v0 el valor de 'i'.
618         lbu     v0,OFFSET_I_DECODECHAR($fp)
619         // Cargo en v1 el byte contenido en encoding_table según el valor de 'i'.
620         // encoding_table[i]
621         lbu     v1,encoding_table(v0)
622         // Cargo en v0 'character'.
623         lb      v0,OFFSET_CHARACTER_DECODECHAR($fp)
624         // Salto a increase_index si el valor recuperado del vector encoding_table
625         // es distinto al valor pasado por parámetro(character).
626         bne     v1,v0,increase_index
627         // Cargo en v0 nuevamente el valor de 'i'.
628         lbu     v0,OFFSET_I_DECODECHAR($fp)
629
630         // Guardo en el stack frame(12) el valor de 'i'
631         //sw    v0,12($fp) //VER
632         sw v0,OFFSET_RETURN_DECODECHAR($fp)
633
634         // Brancheo a return_decode_index_or_zero
```

```
635         b       return_decode_index_or_zero
636 increase_index:
637         // Cargo en v0 nuevamente el valor de 'i'.
638         lbu     v0,OFFSET_I_DECODECHAR($fp)
639         // Sumo en 1 el valor de 'i'(i++).
640         addu    v0,v0,1
641         // Guardo el valor modificado en el stack frame.
642         sb      v0,OFFSET_I_DECODECHAR($fp)
643         // Salto a condition_loop
644         b       condition_loop
645 condition_if_equal:
646         // Cargo en v1 el byte(char) recibido como parámetro.
647         // parametro: character.
648         lb      v1,OFFSET_CHARACTER_DECODECHAR($fp)
649         // Cargo en v0 el inmediato EQUAL_CHAR=61(corresponde a el char '=').
650         li      v0,EQUAL_CHAR                   // 0x3d
651         // Salto a return_decode_error si el char recibido por parámetro no es igual
        a '='.
652         bne     v1,v0,return_decode_error
653         // Guardo un 0(DECODE_EQUAL) en el stack frame(12).
654         sw      zero,OFFSET_RETURN_DECODECHAR($fp)
655         // Salto a return_decode_index_or_zero.
656         b       return_decode_index_or_zero
657 return_decode_error:
658         // Cargo en v0 el inmediato DECODE_ERROR=100
659         li      v0,DECODE_ERROR                 // 0x64
660         // Guardo el DECODE_ERROR en el stack frame.
661         sw      v0,OFFSET_RETURN_DECODECHAR($fp)
662 return_decode_index_or_zero:
663         // Cargo en v0 el valor retornado por DecodeChar
664         lw      v0,OFFSET_RETURN_DECODECHAR($fp)
665
666         move    sp,$fp
667         // Restauro fp
668         lw      $fp,OFFSET_FP_DECODECHAR(sp)
669         // Destruyo el stack frame
670         addu    sp,sp,STACK_FRAME_DECODECHAR
671         // Regreso el control a la función llamante.
672         j       ra
673         .end    DecodeChar
674         //.size DecodeChar, .-DecodeChar
675
676         ///////////////  End Función DecodeChar ///////////////
677
678         ///////////////  Begin Función Decode ///////////////
679
680         .align  2
681         .globl  Decode
682         .ent    Decode
683 Decode:
684         .frame  $fp,STACK_FRAME_DECODE,ra               // vars= 24, regs= 4/0, args=
        16, extra= 8
685         //.mask 0xd0010000,-4
686         //.fmask        0x00000000,0
687         .set    noreorder
688         .cpload t9
689         .set    reorder
690
```

```
691         // Creación del stack frame
692         subu    sp,sp,STACK_FRAME_DECODE
693         .cprestore 16
694
695         sw      ra,OFFSET_RA_DECODE(sp)
696         sw      $fp,OFFSET_FP_DECODE(sp)
697         sw      gp,OFFSET_GP_DECODE(sp)
698         sw      s0,OFFSET_S0_DECODE(sp)
699
700         // De aquí al final de la función uso $fp en lugar de sp.
701         move    $fp,sp
702
703         // Guardo en el stack frame los parámetros recibidos.
704         // a0=puntero a buffer_input
705         sw      a0,OFFSET_BUFFER_INPUT_ENCODE($fp)
706         // Guardo en el stack frame los parámetros recibidos.
707         // a1=puntero a buffer_output
708         sw      a1,OFFSET_BUFFER_OUTPUT_ENCODE($fp)
709         // Guardo un 0 en el stack frame(OFFSET_I_DECODE). Inicializo 'i'.
710         sw      zero,OFFSET_I_DECODE($fp)
711 loop_decode_char:
712         // Cargo en v0 el valor de 'i' guardado en el stack frame.
713         lw      v0,OFFSET_I_DECODE($fp)
714         // Si (i < SIZE_DECODE_CHAR), guardo TRUE en v0, sino FALSE.
715         sltu    v0,v0,SIZE_DECODE_CHAR
716         // Salto a if_decode_char si sigo dentro del bucle.
717         bne     v0,zero,if_decode_char
718         // Salto a main_shift
719         b       main_shift
720 if_decode_char:
721         // Cargo en v1 el valor de 'i'.
722         lw      v1,OFFSET_I_DECODE($fp)
723         // Cargo en v0 el valor de fp + OFFSET_CHARS0_ENCODE ???
724         addu    v0,$fp,OFFSET_CHARS0_ENCODE
725         // Cargo en s0 el valor de buf_input[i]
726         addu    s0,v0,v1
727         // Cargo en v1 el puntero a buf_input
728         lw      v1,OFFSET_BUFFER_INPUT_ENCODE($fp)
729         // Cargo en v0 el valor de 'i'.
730         lw      v0,OFFSET_I_DECODE($fp)
731         // Me desplazo por el vector(buf_input[i])
732         addu    v0,v1,v0
733         // Cargo en v0 el valor del buf_input[i](1 byte).
734         lb      v0,0(v0)
735         // Asigna el valor del byte a a0 antes de llamar a la función.
736         move    a0,v0
737         // Carga en t9 la direccion de la funcion DecodeChar.
738         la      t9,DecodeChar
739         // Hace el llamado a la función.
740         jal     ra,t9
741         // Guardo en s0 el resultado de la función.
742         // El valor regresa en el registro v0
743         sb      v0,0(s0)
744         // Cargo en v1 el valor de 'i'.
745         lw      v1,OFFSET_I_DECODE($fp)
746         // Cargo en v0 el valor de fp + OFFSET_CHARS_ENCODE ???
747         addu    v0,$fp,OFFSET_CHARS0_ENCODE
748         // Cargo en v0 el valor de chars[i](direccion).
```

```
749         addu    v0,v0,v1
750         // Cargo en v1 el byte apuntado.
751         lbu     v1,0(v0)
752         // Cargo en v0 el DECODE_ERROR
753         li      v0,DECODE_ERROR                 // 0x64
754         // Si chars[i] != DECODE_ERROR salto a increase_index_decode
755         bne     v1,v0,increase_index_decode
756         // Guarda en el stack frame un 0.
757         sw      zero,OFFSET_RETURN_ENCODE($fp)
758         // Si chars[i] == DECODE_ERROR retorno un 0.
759         b       return_zero
760 increase_index_decode:
761         // Cargo en v0 el valor de 'i'.
762         lw      v0,OFFSET_I_DECODE($fp)
763         // Sumo en 1 el valor de 'i'(i++).
764         addu    v0,v0,1
765         // Guardo el valor modificado en el stack frame.
766         sw      v0,OFFSET_I_DECODE($fp)
767         // Salto a loop_decode_char
768         b       loop_decode_char
769 main_shift:
770         // Cargo en v0 la dirección de chars[0]
771         lbu     v0,OFFSET_CHARS0_ENCODE($fp)
772         // Hago un shift left logical de SHIFT_2 y lo asigno a v0.
773         sll     v0,v0,SHIFT_2
774         // Guardo el valor en el stack frame.
775         sb      v0,OFFSET_CHAR0_AUX_ENCODE($fp)
776         // Cargo el valor de chars[1] en v0.
777         lbu     v0,OFFSET_CHARS1_ENCODE($fp)
778         // Hago un shift left logical de SHIFT_2 y lo asigno a v0.
779         srl     v0,v0,SHIFT_4
780         // Guardo en el stack frame el valor shifteado.
781         sb      v0,OFFSET_CHAR1_AUX_ENCODE($fp)
782         // Cargo en v1 char1_aux(chars[0] luego de ser shifteado).
783         lbu     v1,OFFSET_CHAR0_AUX_ENCODE($fp)
784         // Cargo en v0 char2_aux(chars[1] luego de ser shifteado).
785         lbu     v0,OFFSET_CHAR1_AUX_ENCODE($fp)
786         // Hago un or de v1 y v0 y lo asigno a v0.
787         or      v0,v1,v0
788         // Guardo en valor en el stack frame.
789         sb      v0,OFFSET_CHAR0_AUX_ENCODE($fp)
790         // Cargo en v1 el puntero al buffer_output.
791         lw      v1,OFFSET_BUFFER_OUTPUT_ENCODE($fp)
792         // Cargo en v0 char1_aux(chars[0] luego de ser shifteado).
793         lbu     v0,OFFSET_CHAR0_AUX_ENCODE($fp)
794         // Guardo en el vector buffer_output el valor de char1_aux.
795         sb      v0,0(v1)
796         // Cargo el valor de chars[1] en v0.
797         lbu     v0,OFFSET_CHARS1_ENCODE($fp)
798         // Hago un shift left de 4 posiciones y lo guardo en v0.
799         sll     v0,v0,SHIFT_4
800         // Guardo en el stack frame el valor shifteado.
801         sb      v0,OFFSET_CHAR0_AUX_ENCODE($fp)
802         // Cargo en v0 chars[2].
803         lbu     v0,OFFSET_CHARS2_ENCODE($fp)
804         // Hago un shift rigth de 2 de chars[2] y lo guardo en v0.
805         srl     v0,v0,SHIFT_2
806         // Guardo en stack frame el valor shifteado.
```

```
807         sb      v0,OFFSET_CHAR1_AUX_ENCODE($fp)
808         // Cargo en v1 y v0 los valores shifteados anteriormente.
809         lbu     v1,OFFSET_CHAR1_AUX_ENCODE($fp)
810         lbu     v0,OFFSET_CHAR0_AUX_ENCODE($fp)
811         // Hago un or de v1 y v0 y lo asigno a v0.
812         or      v0,v1,v0
813         // Vuelvo a guardar en el stack frame el resultado del or.
814         // (**)
815         sb      v0,OFFSET_CHAR1_AUX_ENCODE($fp)
816         // Cargo en v0 el puntero al buffer_output.
817         lw      v0,OFFSET_BUFFER_OUTPUT_ENCODE($fp)
818         // Sumo 1 al puntero para desplazarme dentro del vector.
819         // Luego asigno el resultado a v1.
820         addu    v1,v0,1
821         // Cargo en v0 el resultado de (**).
822         lbu     v0,OFFSET_CHAR1_AUX_ENCODE($fp)
823         // Guardo en el vector buffer_output el valor (**).
824         sb      v0,0(v1)
825         // Cargo en v0 chars[2]
826         lbu     v0,OFFSET_CHARS2_ENCODE($fp)
827         // Hago un shift left de 6.
828         sll     v0,v0,SHIFT_6
829         // Guardo en el stack frame el valor shifteado.
830         // (***)
831         sb      v0,OFFSET_CHAR0_AUX_ENCODE($fp)
832         // Cargo en v0 el puntero al buffer_output.
833         lw      v0,OFFSET_BUFFER_OUTPUT_ENCODE($fp)
834         // Sumo 2 al puntero para desplazarme dentro del vector buffer_output.
835         // Luego asigno el resultado a a0.
836         addu    a0,v0,2
837         // Cargo en v1 el ultimo valor shifteado (***).
838         lbu     v1,OFFSET_CHAR0_AUX_ENCODE($fp)
839         // Cargo en v0 chars[3]
840         lbu     v0,OFFSET_CHARS3_ENCODE($fp)
841         // Hago un or de v1 y v0 y lo asigno a v0.
842         or      v0,v1,v0
843         // Guardo en el vector buffer_output el resultado del or.
844         sb      v0,0(a0)
845         // Cargo en v0 el inmediato 1(RETURNO_OK).
846         li      v0,RETURNO_OK                   // 0x1
847         // Guardo en el stack frame el valor de retorno.
848         sw      v0,OFFSET_RETURN_ENCODE($fp)
849 return_zero:
850         // Cargo en v0 el valor salvado en el stack frame(0).
851         lw      v0,OFFSET_RETURN_ENCODE($fp)
852         move    sp,$fp
853
854         // Restauro ra,fp y gp.
855         lw      ra,OFFSET_RA_DECODE(sp)
856         lw      $fp,OFFSET_FP_DECODE(sp)
857         lw      s0,OFFSET_S0_DECODE(sp)
858
859         // Destruyo el stack frame.
860         addu    sp,sp,STACK_FRAME_DECODE
861         // Devuelvo el control a la función llamante.
862         j       ra
863
864         .end    Decode
```

```
865         .size   Decode, .-Decode
```

B. Stack frame

B.1. Stack frame base_64decode

int base64_decode(int infd, int outfd)		
Offset	**Contents**	**Type reserved area**
68	outfd	**ABA (caller)**
64	infd	
60	////////////////////////	**SRA**
56	ra	
52	fp	
48	gp	
39	////////////////////////	**LTA**
38	OUT_BUFFER	
37		
36		
32	IN_BUFFER	
28		
27		
26		
12		**ABA (callee)**
8		
4		
0		

Figura 1: Stack frame base64decode

B.2. Stack frame base_64encode

int base64_encode(int infd, int outfd)		
Offset	**Contents**	**Type reserved area**
68	outfd	**ABA (caller)**
64	infd	
60	////////////////////////	**SRA**
56	ra	
52	fp	
48	gp	
39	////////////////////////	**LTA**
38	IN_BUFFER	
37		
36		
32	OUT_BUFFER	
28		
27		
26		
12		**ABA (callee)**
8		
4		
0		

Figura 2: Stack frame base64encode

B.3. Stack frame decodeChar

unsigned char DecodeChar(char character)		
Offset	**Contents**	**Type reserved area**
36	character	**ABA (caller)**
32	fp	**SRA**
28	gp	
24	return	**LTA**
20	i	
16	character	
12	a3	**ABA(callee)**
8	a2	
4	a1	
0	a0	

Figura 3: Stack frame decodeChar

B.4. Stack frame decode

unsigned char Decode(unsigned char *buf_input, unsigned char *buf_output)		
Offset	**Contents**	**Type reserved area**
68	*buffer_output	**ABA (caller)**
64	*buffer_input	
60	ra	**SRA**
56	fp	
52	gp	
48	s0	
39	///////////////////////	**LTA**
38	///////////////////////	
37	char1_aux	
36	char0_aux	
32	i	
28	///////////////////////	
27	chars3	
26	chars2	
25	chars1	
24	chars0	
20	return	
16	///////////////////////	
12	a3	**ABA (callee)**
8	a2	
4	a1	
0	a0	

Figura 4: Stack frame decode

B.5. Stack frame encode

void Encode(const unsigned char* buffer, unsigned int length, unsigned char* output)		
Offset	**Contents**	**Type reserved area**
24	*output	**ABA (caller)**
20	length	
16	*buffer	
12	fp	**SRA**
8	gp	
7	OFFSET_B4_AUX	**LTA**
6	OFFSET_B3_AUX_2	
5	OFFSET_B3_AUX	
4	OFFSET_B2_AUX	
3	OFFSET_B1_AUX	
2	OFFSET_B3	
1	OFFSET_B2	
0	OFFSET_B1	

Figura 5: Stack frame encode