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 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.

El programa implementado satisface los siguientes requerimientos, que se detalla 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 retorna un buffer de 3 caracteres con el decode de 4 caracteres en base64. Se debe cumplir:

* Pre: el buffer input contiene 4 caracteres. El buffer output tiene por lo menos 3 caracteres
* Post: retorna un buffer de 3 byte con los caracteres en ASCII. retorna 0 si error 1 si ok.

- 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 en la función en donde recibe 3 caracteres en buffer y los convierte en 4 caracteres codificados en output. Se debe cumplir:

  * Pre: el buffer contiene length caracteres (1 a 3) y todos los caracteres son validos * Post: retorna un buffer de 4 byte con los caracteres en base64.

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 obtener un ejecutable, se creó un archivo `makefile` cuyo contenido es:

```
1 CC = gcc
2 CFLAGS = -o0 -g -Wall -Werror -pedantic -std=c99
3
4 OBJECTS = command.o encode.o file.c
5 EXEC = tp0
6
```

```
7  VALGRIND = valgrind --track-origins=yes --leak-check=full
8  VALGRIND-V = $(VALGRIND) -v
9
10 all: $(EXEC)
11
12 command.o: command.c command.h
13         $(CC) $(CFLAGS) -c command.c -o command.o
14 encode.o: encode.c encode.h
15         $(CC) $(CFLAGS) -c encode.c -o encode.o
16 file.o: file.c file.h
17         $(CC) $(CFLAGS) -c file.c -o file.o
18
19 $(EXEC): $(OBJECTS)
20         $(CC) $(CFLAGS) $(OBJECTS) main.c -o $(EXEC) -lm
21
22 run: $(EXEC)
23         ./$(EXEC)
24
25 valgrind: $(EXEC)
26         $(VALGRIND) ./$(EXEC)
27
28 valgrind-verb: $(EXEC)
29         $(VALGRIND-V) ./$(EXEC)
30
31 clean:
32         rm -f *.o $(EXEC)
33
```

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

```
$ make
```

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

```
$ ./tp0
```

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 | ./tp0 -a encode
```

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

Para el caso de hacerlo en el emulador GXemul que provee la cátedra, utilizando la máquina virtual que contiene el sistema operativo NetBSD, no se utilizó el archivo Makefile, la compilación se realizó con la herramienta `gcc`.

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

```
$ ./tp0 -h  o ./tp0 --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:
  tp0 -a encode -i ~/input -o ~/output
  tp0 -a decode
```

- Mensaje de version

```
$ ./tp0 -V  o ./tp0 --version
Version: 0.1

```

- Archivo de entrada no válido

```
$ ./tp0 -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:
  tp0 -a encode -i ~/input -o ~/output
  tp0 -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: 0.1\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_NOENCODE;
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)
211         opt->error = FileOpenForWrite(&opt->output, opt->output_route);
212
213     if(!opt->error){
214         opt->error = _CommandEncodeDecode(opt);
215         FileClose(&opt->input);
216         FileClose(&opt->output);
217     }
218     else {
219         FileClose(&opt->input);
220     }
221     return opt->error;
222 }
223
224 char _CommandEncodeDecode(CommandOptions *opt) {
225 /*  unsigned char buf_decoded[3];
226     unsigned char buf_encoded[4];
227     unsigned char count = 0;
```

```
228     if(opt->encode_opt == CMD_ENCODE){
229         while(!FileEofReached(&opt->input)){
230             memset(buf_decoded, 0, 3);
231             unsigned int read = FileRead(&opt->input, buf_decoded, 3);
232             if (read > 0) {
233                 Encode(buf_decoded, read, buf_encoded);
234                 FileWrite(&opt->output, buf_encoded, 4);
235                 ++count;
236                 if (count == 18) { // 19 * 4 = 76 bytes
237                     FileWrite(&opt->output, (unsigned char *) "\n", 1);
238                     count = 0;
239                 }
240 
241             }
242         }
243     }
244 
245     if (opt->encode_opt == CMD_DECODE) {
246         while (!FileEofReached(&opt->input) && !CommandHasError(opt)) {
247             unsigned int read = FileRead(&opt->input, buf_encoded, 4);
248             if (read > 0) {   // Solo es 0 si alcance el EOF
249                 if (read != 4) {  //Siempre debo leer 4 sino el formato es incorrecto
250                     fprintf(stderr, "Longitud de archivo no es multiplo de 4\n");
251                     CommandSetError(opt);
252                 } else {
253                     ++count;
254                     if (count == 18) { // 19 * 4 = 76 bytes
255                         unsigned char aux;
256                         FileRead(&opt->input, &aux, 1);
257                         count = 0;
258                     }
259                     if (Decode(buf_encoded, buf_decoded)) {
260                         char aux = 0;
261                         if (buf_encoded[2] == '=')
262                             ++aux;
263                         if (buf_encoded[3] == '=')
264                             ++aux;
265 
266                         FileWrite(&opt->output, buf_decoded, 3 - aux);
267                     } else {
268                         fprintf(stderr, "Caracteres invalidos en archivo codificado:
    ");
269                         unsigned int i;
270                         for (i = 0; i < 4; ++i)
271                             fprintf(stderr, "%c", buf_encoded[i]);
272                         CommandSetError(opt);
273                     }
274                 }
275             }
276 
277 
278         }
279     }
280         */
281     return opt->error;
282 }
283 
284 void CommandErrArg() {
```

```
285     fprintf(stderr, "Invalid Arguments\n");
286     fprintf(stderr,"Options:\n");
287     fprintf(stderr,"  -V, --version    Print version and quit.\n");
288     fprintf(stderr,"  -h, --help       Print this information.\n");
289     fprintf(stderr,"  -i, --input      Location of the input file.\n");
290     fprintf(stderr,"  -o, --output     Location of the output file.\n");
291     fprintf(stderr,"  -a, --action     Program action: encode (default) or decode.\n"
        );
292     fprintf(stderr,"Examples:\n");
293     fprintf(stderr,"  tp0 -a encode -i ~/input -o ~/output\n");
294     fprintf(stderr,"  tp0 -a decode\n");
295 }
296
297 void FileCreate(File *file){
298     file->file = 0;
299     file->eof = 0;
300 }
301
302 char FileOpenForRead(File* file, const char *route ){
303     if(route == NULL) {
304         file->file = stdin;
305     } else {
306         file->file = fopen(route, "rb");
307         if (file->file == NULL) {
308             int err = errno;
309             fprintf(stderr, "File Open Error; %s\n", strerror(err));
310             return ERROR;
311         }
312     }
313     return OK;
314 }
315
316 char FileOpenForWrite(File* file, const char *route ) {
317     if(route == NULL) {
318         file->file = stdout;
319     } else {
320         file->file = fopen(route, "wb");
321         if (file->file == NULL) {
322             int err = errno;
323             fprintf(stderr, "File Open Error; %s\n", strerror(err));
324             return ERROR;
325         }
326     }
327     return OK;
328 }
329
330 int FileClose(File* file){
331     if(file->file == stdin || file->file == stdout)
332         return OK;
333
334     int result = fclose(file->file);
335     if (result == EOF){
336         int err = errno;
337         fprintf(stderr, "File Close Error; %s\n", strerror(err));
338         return ERROR;
339     }
340     return OK;
341 }
```

```
342
343 /*******************************
344  * FIN: DEFINICION DE FUNCIONES *
345  *******************************/
346
347 int main(int argc, char** argv) {
348     struct option arg_long[] = {
349             {"input",   required_argument,  NULL,  'i'},
350             {"output",  required_argument,  NULL,  'o'},
351             {"action",  required_argument,  NULL,  'a'},
352             {"help",    no_argument,        NULL,  'h'},
353             {"version", no_argument,        NULL,  'V'},
354     };
355     char arg_opt_str[] = "i:o:a:hV";
356     int arg_opt;
357     int arg_opt_idx = 0;
358     char should_finish = FALSE;
359
360     CommandOptions cmd_opt;
361     CommandCreate(&cmd_opt);
362
363     if(argc == 1)
364         CommandSetError(&cmd_opt);
365
366     while((arg_opt =
367                     getopt_long(argc, argv, arg_opt_str, arg_long, &arg_opt_idx)) !=
    -1 && !should_finish) {
368         switch(arg_opt){
369                 case 'i':
370                         CommandSetInput(&cmd_opt, optarg);
371                         break;
372                 case 'o':
373                 CommandSetOutput(&cmd_opt, optarg);
374                 break;
375                 case 'h':
376                         CommandHelp();
377                 should_finish = TRUE;
378                 break;
379                 case 'V':
380                         CommandVersion();
381                 should_finish = TRUE;
382                 break;
383                 case 'a':
384                     CommandSetEncodeOpt(&cmd_opt, optarg);
385                             break;
386                 default:
387                         CommandSetError(&cmd_opt);
388                         break;
389         }
390     }
391
392     if(should_finish)
393         return 0;
394
395     if(!CommandHasError(&cmd_opt)) {
396         CommandProcess(&cmd_opt);
397     } else {
398         CommandErrArg();
```

```
399             return 1;
400         }
401         return 0;
402 }
```

A.0.2. Header file base64.h

```
#ifndef TP1_BASE64_H
#define TP1_BASE64_H

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

#endif
```

A.0.3. Assembly base64.S

A.0.4. Assembly encode.S

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

        .file   1 "encode.c"
        #.section .mdebug.abi32
        #.previous
        #.abicalls
        .data
        .align  2
        .type   encoding_table, @object
        .size   encoding_table, 64
encoding_table:
        .byte   65
        .byte   66
        .byte   67
        .byte   68
        .byte   69
        .byte   70
        .byte   71
        .byte   72
        .byte   73
        .byte   74
        .byte   75
        .byte   76
        .byte   77
        .byte   78
        .byte   79
        .byte   80
        .byte   81
        .byte   82
        .byte   83
        .byte   84
        .byte   85
        .byte   86
        .byte   87
        .byte   88
        .byte   89
        .byte   90
        .byte   97
        .byte   98
        .byte   99
        .byte   100
        .byte   101
        .byte   102
        .byte   103
        .byte   104
        .byte   105
        .byte   106
        .byte   107
        .byte   108
        .byte   109
        .byte   110
        .byte   111
        .byte   112
        .byte   113
        .byte   114
```

```
57          .byte   115
58          .byte   116
59          .byte   117
60          .byte   118
61          .byte   119
62          .byte   120
63          .byte   121
64          .byte   122
65          .byte   48
66          .byte   49
67          .byte   50
68          .byte   51
69          .byte   52
70          .byte   53
71          .byte   54
72          .byte   55
73          .byte   56
74          .byte   57
75          .byte   43
76          .byte   47
77          .text
78          .align  2
79          .globl  Encode
80          .ent    Encode
81
82          ####### Función Encode #######
83
84  Encode:
85          .frame  $fp,24,ra               # vars= 8, regs= 2/0, args= 0, extra= 8
86          #.mask  0x50000000,-4
87          #.fmask 0x00000000,0
88          .set    noreorder
89          .cpload t9
90          .set    reorder
91
92          # Creación del stack frame
93          subu    sp,sp,24
94
95          .cprestore 0
96          sw      $fp,20(sp)
97          sw      gp,16(sp)
98
99          # De aquí al final de la función uso $fp en lugar de sp.
100         move    $fp,sp
101
102         # Guardo el primer parámetro *buffer
103         sw      a0,24($fp)
104         # Guardo el segundo parámetro length(cantidad de caracteres)
105         sw      a1,28($fp)
106         # Guardo el puntero al array de salida(output)
107         sw      a2,32($fp)
108
109         # Cargo en v0 el puntero al buffer.
110         lw      v0,24($fp)
111         # Cargo en v0 el 1er byte del buffer.
112         lbu     v0,0(v0)
113         # Guardo el 1er byte en el stack frame
114         sb      v0,8($fp)
```

```
115         # Cargo nuevamente la dirección del buffer.
116         lw      v0,24($fp)
117         # Aumento en 1(1 byte) la dirección del buffer.
118         # Me muevo por el array del buffer.
119         addu    v0,v0,1
120         # Cargo el 2do byte del buffer.
121         lbu     v0,0(v0)
122         # Guardo el 2do byte en el stack frame.
123         sb      v0,9($fp)
124         # Cargo nuevamente la dirección del buffer.
125         lw      v0,24($fp)
126         # Aumento en 2(2 byte) la dirección del buffer.
127         # Me muevo por el array del buffer.
128         addu    v0,v0,2
129         # Cargo el 2do byte del buffer.
130         lbu     v0,0(v0)
131         # Guardo el 3er byte en stack frame.
132         sb      v0,10($fp)
133         # Cargo en v0 el 1er byte.
134         lbu     v0,8($fp)
135         # Muevo 2 'posiciones' hacia la derecha(shift 2).
136         srl     v0,v0,2
137         # Guardo el nuevo byte en una variable auxiliar.
138         sb      v0,11($fp)
139         # Cargo en v1 el puntero al output.
140         lw      v1,32($fp)
141         # Cargo en v0 el byte shifteado.
142         lbu     v0,11($fp)
143         # Cargo en v0 el caracter(byte) de la tabla encoding(encoding_table)
144         lbu     v0,encoding_table(v0)
145         # Cargo en v0 el 1er byte de la dirección del output.
146         sb      v0,0(v1)
147         # Cargo en v0 el 1er byte del buffer nuevamente.
148         lbu     v0,8($fp)
149         # Muevo 6 'posiciones' hacia la izquierda(shift 6).
150         sll     v0,v0,6
151         # Guardo el resultado del shift en el Stack Frame.
152         sb      v0,12($fp)
153         # Cargo el byte sin signo shifteado.
154         lbu     v0,12($fp)
155         # Muevo 2 'posiciones' hacia la derecha(shift 2).
156         srl     v0,v0,2
157         # Guardo el nuevo resultado del shift en el Stack Frame.
158         sb      v0,12($fp)
159         # Cargo el 2do byte del buffer en v0.
160         lbu     v0,9($fp)
161         # Hago un shift left de 4 posiciones.
162         srl     v0,v0,4
163         # Cargo en v1 el resultado(byte) del shift right 2.
164         lbu     v1,12($fp)
165         # Hago un 'or' entre v1 y v0 para obtener el 2 indice de la tabla.
166         or      v0,v1,v0
167         #(*) Guardo en stack frame(12) el resultado del 'or' anterior.
168         sb      v0,12($fp)
169         # Cargo en v0 el puntero al output.
170         lw      v0,32($fp)
171         # Cargo en v1 la dirección del output + 1(1byte).
172         addu    v1,v0,1
```

```
173         # Cargo en v0 el ultimo resultado del shift(*)
174         lbu     v0,12($fp)
175         # Cargo en v0 el caracter(byte) de la tabla encoding(encoding_table)
176         lbu     v0,encoding_table(v0)
177         # Salvo en el output array(output[1]) el valor del encoding_table
178         sb      v0,0(v1)
179         # Cargo en v0 el puntero al output.
180         lw      v0,32($fp)
181         # Sumo 2 a la dirección del output(output[2]).
182         # Me desplazo dentro del output array.
183         addu    v1,v0,2
184         # Cargo en v0 el caracter ascii 61('=').
185         li      v0,61                   # 0x3d
186         # Salvo en el output array(output[2]) el valor '='.
187         sb      v0,0(v1)
188         # Cargo en v0 el puntero al output.
189         lw      v0,32($fp)
190         # Sumo 3 a la dirección del output(output[3]).
191         # Me desplazo dentro del output array.
192         addu    v1,v0,3
193         # Cargo en v0 el caracter ascii 61('=').
194         li      v0,61                   # 0x3d
195         # Salvo en el output array(output[3]) el valor '='.
196         sb      v0,0(v1)
197         # Cargo en v1 el parametro length.
198         lw      v1,28($fp)
199         # Cargo en v0 el valor 3.
200         li      v0,3                    # 0x3
201         # Si el length == 3 salto a buffer_size_2.
202         bne     v1,v0,buffer_size_2
203         # Si el tamanio del buffer es 3 continuo NO salto.
204         # Cargo en v0 el 3er byte del buffer.
205         lbu     v0,10($fp)
206         # Hago un shift right de 6.
207         srl     v0,v0,6
208         # Guardo el nuevo byte en el stack frame.
209         sb      v0,13($fp)
210         # Cargo el 2do byte del buffer en v0.
211         lbu     v0,9($fp)
212         # Hago un shift left de 4.
213         sll     v0,v0,4
214         # Guardo en el stack frame(14) el nuevo valor.
215         sb      v0,14($fp)
216         # Cargo en v0 el byte shifteado sin signo.
217         lbu     v0,14($fp)
218         # Hago un shift rigth de 2.
219         srl     v0,v0,2
220         # Guardo en el stack frame(14) el valor shifteado.
221         sb      v0,14($fp)
222         # Cargo en v1 el valor del SF(13)
223         lbu     v1,13($fp)
224         # Idem en v0(13).
225         lbu     v0,14($fp)
226         # Hago un 'or' y almaceno en v0.
227         or      v0,v1,v0
228         # Guardo en el stack frame(13) el resultado del 'or'.
229         sb      v0,13($fp)
230         # Cargo en v0 el puntero al output.
```

```
231         lw      v0,32($fp)
232         # Me desplazo por el vector 'output' en 2 posiciones(output[2]).
233         addu    v1,v0,2
234         # Cargo en v0 el resultado del 'or' anterior.
235         lbu     v0,13($fp)
236         # Busco en la tabla de encoding el caracter que corresponde.
237         # Luego cargo el byte en v0.
238         lbu     v0,encoding_table(v0)
239         # Guardo el valor recuperado de la tabla encoding_table en el output[2].
240         sb      v0,0(v1)
241         # Cargo en v0 el 3er byte del buffer.
242         lbu     v0,10($fp)
243         # Hago un shift left de 2.
244         sll     v0,v0,2
245         # Guardo en el stack frame el valor shifteado.
246         sb      v0,15($fp)
247         # Cargo el byte sin signo shifteado.
248         lbu     v0,15($fp)
249         # Hago un shift rigth de 2.
250         srl     v0,v0,2
251         # Guardo en el stack frame el valor shifteado.
252         sb      v0,15($fp)
253         # Cargo en v0 el puntero al output.
254         lw      v0,32($fp)
255         # Sumo 3 a la dirección del output(output[3]).
256         # Me desplazo dentro del output array.
257         addu    v1,v0,3
258         # Cargo en v0 el ultimo valor shifteado guardado.
259         lbu     v0,15($fp)
260         # Busco en la tabla de encoding el caracter que corresponde.
261         # Luego cargo el byte en v0.
262         lbu     v0,encoding_table(v0)
263         # Guardo el valor recuperado de la tabla encoding_table en el output[3].
264         sb      v0,0(v1)
265         # Salto a return_encode
266         b       return_encode
267 buffer_size_2:
268         # Cargo en v1 el valor del parámetro length.
269         lw      v1,28($fp)
270         # Cargo en v0 el valor 2.
271         li      v0,2                    # 0x2
272         # Si length != 2 salgo de la función.
273         bne     v1,v0,return_encode
274         # Cargo en v0 el 3er byte del buffer.
275         lbu     v0,10($fp)
276         # Hago un shift right de 6.
277         srl     v0,v0,6
278         # Guardo en el stack frame el ultimo valor shifteado.
279         sb      v0,15($fp)
280         # Cargo el 2do byte del buffer en v0.
281         lbu     v0,9($fp)
282         # Hago un shift left de 4 posiciones.
283         sll     v0,v0,4
284         # Guardo en el stack frame nuevo valor shifteado.
285         sb      v0,14($fp)
286         # Cargo en v0 el byte shifteado sin signo.
287         lbu     v0,14($fp)
288         # Hago un shift right de 2 posiciones.
```

```
289         srl     v0,v0,2
290         # Guardo en el stack frame el valor shifteado.
291         sb      v0,14($fp)
292         # Cargo en v1 uno de los valores shiftedos(b3aux).
293         lbu     v1,15($fp)
294         # Cargo en v0 uno de los valores shiftedos(b3aux2).
295         lbu     v0,14($fp)
296         # Hago un 'or' entre b3aux y b3aux2.
297         or      v0,v1,v0
298         # Guardo en el stack frame el resutado del 'or'.
299         sb      v0,15($fp)
300         # Cargo en v0 el puntero al output.
301         lw      v0,32($fp)
302         # Me desplazo dentro del output array y lo guardo en v1.
303         addu    v1,v0,2
304         # Cargo en v0 ultimo resultado del 'or'
305         lbu     v0,15($fp)
306         # Busco en la tabla de encoding el caracter que corresponde.
307         # Luego cargo el byte en v0.
308         lbu     v0,encoding_table(v0)
309         # Guardo el valor recuperado de la tabla encoding_table en el output[2].
310         sb      v0,0(v1)
311 return_encode:
312         move    sp,$fp
313         lw      $fp,20(sp)
314         # destruyo stack frame
315         addu    sp,sp,24
316         j       ra
317         .end    Encode
318         .size   Encode, .-Encode
319         #.ident "GCC: (GNU) 3.3.3 (NetBSD nb3 20040520)"
```

A.0.5. Assembly decode.S

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

#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

        .file   1 "decode.c"
        #.section .mdebug.abi32
        #.previous
        #.abicalls
        .data
        .align  2
        .type   encoding_table, @object
        .size   encoding_table, 64
encoding_table:
        .byte   65
        .byte   66
        .byte   67
        .byte   68
        .byte   69
        .byte   70
        .byte   71
        .byte   72
        .byte   73
        .byte   74
        .byte   75
        .byte   76
        .byte   77
        .byte   78
        .byte   79
        .byte   80
        .byte   81
        .byte   82
        .byte   83
        .byte   84
        .byte   85
        .byte   86
        .byte   87
        .byte   88
        .byte   89
        .byte   90
        .byte   97
        .byte   98
        .byte   99
        .byte   100
        .byte   101
        .byte   102
        .byte   103
        .byte   104
        .byte   105
        .byte   106
```

```
        .byte   107
        .byte   108
        .byte   109
        .byte   110
        .byte   111
        .byte   112
        .byte   113
        .byte   114
        .byte   115
        .byte   116
        .byte   117
        .byte   118
        .byte   119
        .byte   120
        .byte   121
        .byte   122
        .byte   48
        .byte   49
        .byte   50
        .byte   51
        .byte   52
        .byte   53
        .byte   54
        .byte   55
        .byte   56
        .byte   57
        .byte   43
        .byte   47
        .align  2
        .type   encoding_table_size, @object
        .size   encoding_table_size, 4

encoding_table_size:
        .word   64
        .text
        .align  2
        .globl  DecodeChar
        .ent    DecodeChar

        ####### Función DecodeChar #######

DecodeChar:
        .frame  $fp,24,ra               # vars= 8, regs= 2/0, args= 0, extra= 8
        #.mask  0x50000000,-4
        #.fmask 0x00000000,0
        .set    noreorder
        .cpload t9
        .set    reorder

        # Creación del stack frame
        subu    sp,sp,24
        .cprestore 0

        # Guardo fp y gp en el stack frame
        sw      $fp,20(sp)
        sw      gp,16(sp)
        # De aquí al final de la función uso $fp en lugar de sp.
        move    $fp,sp
```

```
115
116         # Guardo en v0 el parámetro recibido: character.
117         move    v0,a0
118         # Guardo en el stack frame 'character'.
119         sb      v0,8($fp)
120         # Guardo en un '0' en el stack frame.
121         # Inicializo la variable 'i'.
122         sb      zero,9($fp)
123 condition_loop:
124         # Cargo en v0 el byte guardado anteriormente(0 o el nuevo valor de 'i').
125         lbu     v0,9($fp)
126         # Cargo en v1 el size del encoding_table(64).
127         lw      v1,encoding_table_size
128         # Si (i < encoding_table_size), guardo TRUE en v0, sino FALSE.
129         slt     v0,v0,v1
130         # Salto a condition_if si v0 != 0.
131         bne     v0,zero,condition_if
132         # Brancheo a condition_if_equal
133         b       condition_if_equal
134 condition_if:
135         # Cargo en v0 el valor de 'i'.
136         lbu     v0,9($fp)
137         # Cargo en v1 el byte contenido en encoding_table según el valor de 'i'.
138         # encoding_table[i]
139         lbu     v1,encoding_table(v0)
140         # Cargo en v0 'character'.
141         lb      v0,8($fp)
142         # Salto a increase_index si el valor recuperado del vector encoding_table
143         # es distinto al valor pasado por parámetro(character).
144         bne     v1,v0,increase_index
145         # Cargo en v0 nuevamente el valor de 'i'.
146         lbu     v0,9($fp)
147         # Guardo en el stack frame(12) el valor de 'i'
148         sw      v0,12($fp)
149         # Brancheo a return_decode_index_or_zero
150         b       return_decode_index_or_zero
151 increase_index:
152         # Cargo en v0 nuevamente el valor de 'i'.
153         lbu     v0,9($fp)
154         # Sumo en 1 el valor de 'i'(i++).
155         addu    v0,v0,1
156         # Guardo el valor modificado en el stack frame.
157         sb      v0,9($fp)
158         # Salto a condition_loop
159         b       condition_loop
160 condition_if_equal:
161         # Cargo en v1 el byte(char) recibido como parámetro.
162         # parametro: character.
163         lb      v1,8($fp)
164         # Cargo en v0 el inmediato 61(corresponde a el char '=').
165         li      v0,61                   # 0x3d
166         # Salto a return_decode_error si el char recibido por parámetro no es igual a
    '='.
167         bne     v1,v0,return_decode_error
168         # Guardo un 0(DECODE_EQUAL) en el stack frame(12).
169         sw      zero,12($fp)
170         # Salto a return_decode_index_or_zero.
171         b       return_decode_index_or_zero
```

```
172 return_decode_error:
173         # Cargo en v0 el inmediato DECODE_ERROR=100
174         li      v0,DECODE_ERROR                 # 0x64
175         # Guardo el DECODE_ERROR en el stack frame.
176         sw      v0,12($fp)
177 return_decode_index_or_zero:
178         # Cargo en v0 el valor retornado por DecodeChar
179         lw      v0,12($fp)
180
181         move    sp,$fp
182         # Restauro fp
183         lw      $fp,20(sp)
184         # Destruyo el stack frame
185         addu    sp,sp,24
186         # Regreso el control a la función llamante.
187         j       ra
188         .end    DecodeChar
189         #.size  DecodeChar, .-DecodeChar
190         # ??? FALTA EL .text ???
191
192         ####### Función Decode #######
193
194         .align  2
195         .globl  Decode
196         .ent    Decode
197 Decode:
198         .frame  $fp,64,ra               # vars= 24, regs= 4/0, args= 16, extra= 8
199         #.mask  0xd0010000,-4
200         #.fmask 0x00000000,0
201         .set    noreorder
202         .cpload t9
203         .set    reorder
204
205         # Creación del stack frame
206         subu    sp,sp,64
207         .cprestore 16
208
209         sw      ra,60(sp)
210         sw      $fp,56(sp)
211         sw      gp,52(sp)
212         sw      s0,48(sp)
213
214         # De aquí al final de la función uso $fp en lugar de sp.
215         move    $fp,sp
216
217         # Guardo en el stack frame los parámetros recibidos.
218         # a0=puntero a buffer_input
219         sw      a0,64($fp)
220         # Guardo en el stack frame los parámetros recibidos.
221         # a1=puntero a buffer_output
222         sw      a1,68($fp)
223         # Guardo un 0 en el stack frame(32). Inicializo 'i'.
224         sw      zero,32($fp)
225 loop_decode_char:
226         # Cargo en v0 el valor de 'i' guardado en el stack frame.
227         lw      v0,32($fp)
228         # Si (i < SIZE_DECODE_CHAR), guardo TRUE en v0, sino FALSE.
229         sltu    v0,v0,SIZE_DECODE_CHAR
```

```
230         # Salto a if_decode_char si sigo dentro del bucle.
231         bne     v0,zero,if_decode_char
232         # Salto a main_shift
233         b       main_shift
234 if_decode_char:
235         # Cargo en v1 el valor de 'i'.
236         lw      v1,32($fp)
237         # Cargo en v0 el valor de fp + 24 ???
238         addu    v0,$fp,24
239         # Cargo en s0 el valor de buf_input[i]
240         addu    s0,v0,v1
241         # Cargo en v1 el puntero a buf_input
242         lw      v1,64($fp)
243         # Cargo en v0 el valor de 'i'.
244         lw      v0,32($fp)
245         # Me desplazo por el vector(buf_input[i])
246         addu    v0,v1,v0
247         # Cargo en v0 el valor del buf_input[i](1 byte).
248         lb      v0,0(v0)
249         # Asigna el valor del byte a a0 antes de llamar a la función.
250         move    a0,v0
251         # Carga en t9 la direccion de la funcion DecodeChar.
252         la      t9,DecodeChar
253         # Hace el llamado a la función.
254         jal     ra,t9
255         # Guardo en s0 el resultado de la función.
256         # El valor regresa en el registro v0
257         sb      v0,0(s0)
258         # Cargo en v1 el valor de 'i'.
259         lw      v1,32($fp)
260         # Cargo en v0 el valor de fp + 24 ???
261         addu    v0,$fp,24
262         # Cargo en v0 el valor de chars[i](direccion).
263         addu    v0,v0,v1
264         # Cargo en v1 el byte apuntado.
265         lbu     v1,0(v0)
266         # Cargo en v0 el DECODE_ERROR
267         li      v0,DECODE_ERROR                 # 0x64
268         # Si chars[i] != DECODE_ERROR salto a increase_index_decode
269         bne     v1,v0,increase_index_decode
270         # Guarda en el stack frame un 0.
271         sw      zero,40($fp)
272         # Si chars[i] == DECODE_ERROR retorno un 0.
273         b       return_zero
274 increase_index_decode:
275         # Cargo en v0 el valor de 'i'.
276         lw      v0,32($fp)
277         # Sumo en 1 el valor de 'i'(i++).
278         addu    v0,v0,1
279         # Guardo el valor modificado en el stack frame.
280         sw      v0,32($fp)
281         # Salto a loop_decode_char
282         b       loop_decode_char
283 main_shift:
284         # Cargo en v0 la dirección de chars[0]
285         lbu     v0,24($fp)
286         # Hago un shift left logical de SHIFT_2 y lo asigno a v0.
287         sll     v0,v0,SHIFT_2
```

```
288         # Guardo el valor en el stack frame.
289         sb      v0,36($fp)
290         # Cargo el valor de chars[1] en v0.
291         lbu     v0,25($fp)
292         # Hago un shift left logical de SHIFT_2 y lo asigno a v0.
293         srl     v0,v0,SHIFT_4
294         # Guardo en el stack frame el valor shifteado.
295         sb      v0,37($fp)
296         # Cargo en v1 char1_aux(chars[0] luego de ser shifteado).
297         lbu     v1,36($fp)
298         # Cargo en v0 char2_aux(chars[1] luego de ser shifteado).
299         lbu     v0,37($fp)
300         # Hago un or de v1 y v0 y lo asigno a v0.
301         or      v0,v1,v0
302         # Guardo en valor en el stack frame.
303         sb      v0,36($fp)
304         # Cargo en v1 el puntero al buffer_output.
305         lw      v1,68($fp)
306         # Cargo en v0 char1_aux(chars[0] luego de ser shifteado).
307         lbu     v0,36($fp)
308         # Guardo en el vector buffer_output el valor de char1_aux.
309         sb      v0,0(v1)
310         # Cargo el valor de chars[1] en v0.
311         lbu     v0,25($fp)
312         # Hago un shift left de 4 posiciones y lo guardo en v0.
313         sll     v0,v0,SHIFT_4
314         # Guardo en el stack frame el valor shifteado.
315         sb      v0,36($fp)
316         # Cargo en v0 chars[2].
317         lbu     v0,26($fp)
318         # Hago un shift rigth de 2 de chars[2] y lo guardo en v0.
319         srl     v0,v0,SHIFT_2
320         # Guardo en stack frame el valor shifteado.
321         sb      v0,37($fp)
322         # Cargo en v1 y v0 los valores shifteados anteriormente.
323         lbu     v1,37($fp)
324         lbu     v0,36($fp)
325         # Hago un or de v1 y v0 y lo asigno a v0.
326         or      v0,v1,v0
327         # Vuelvo a guardar en el stack frame el resultado del or.
328         # (**)
329         sb      v0,37($fp)
330         # Cargo en v0 el puntero al buffer_output.
331         lw      v0,68($fp)
332         # Sumo 1 al puntero para desplazarme dentro del vector.
333         # Luego asigno el resultado a v1.
334         addu    v1,v0,1
335         # Cargo en v0 el resultado de (**).
336         lbu     v0,37($fp)
337         # Guardo en el vector buffer_output el valor (**).
338         sb      v0,0(v1)
339         # Cargo en v0 chars[2]
340         lbu     v0,26($fp)
341         # Hago un shift left de 6.
342         sll     v0,v0,SHIFT_6
343         # Guardo en el stack frame el valor shifteado.
344         # (***)
345         sb      v0,36($fp)
```

```
346         # Cargo en v0 el puntero al buffer_output.
347         lw      v0,68($fp)
348         # Sumo 2 al puntero para desplazarme dentro del vector buffer_output.
349         # Luego asigno el resultado a a0.
350         addu    a0,v0,2
351         # Cargo en v1 el ultimo valor shifteado (***).
352         lbu     v1,36($fp)
353         # Cargo en v0 chars[3]
354         lbu     v0,27($fp)
355         # Hago un or de v1 y v0 y lo asigno a v0.
356         or      v0,v1,v0
357         # Guardo en el vector buffer_output el resultado del or.
358         sb      v0,0(a0)
359         # Cargo en v0 el inmediato 1(RETURNO_OK).
360         li      v0,RETURNO_OK                   # 0x1
361         # Guardo en el stack frame el valor de retorno.
362         sw      v0,40($fp)
363 return_zero:
364         # Cargo en v0 el valor salvado en el stack frame(0).
365         lw      v0,40($fp)
366         move    sp,$fp
367
368         # Restauro ra,fp y gp.
369         lw      ra,60(sp)
370         lw      $fp,56(sp)
371         lw      s0,48(sp)
372
373         # Destruyo el stack frame.
374         addu    sp,sp,64
375         # Devuelvo el control a la función llamante.
376         j       ra
377
378         .end    Decode
379         #.size  Decode, .-Decode
380         #.ident "GCC: (GNU) 3.3.3 (NetBSD nb3 20040520)"
```