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 Re-entrega:	30-10-2018
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/`).

2. Introducción

Se desarrollo un programa con el propósito de familiarizarnos tanto con el conjunto de instrucciones MIPS como con el concepto de ABI. Se realiza una explicación de cómo se realizó el diseño y la explicación de las implementaciones realizadas, además de las pruebas hechas del programa.

Al final se presenta el código fuente de lo realizado.

3. 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

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

```

  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:

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

```

  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 .

  - base64_encode
    La función recibe por parámetro a *infd* y *outfd*, los *file descriptors* correspondientes a los archivos de entrada y salida pre-abiertos por ***main.c***. La función función realiza el encoding a base 64 en assembly ***MIPS***, cumpliendo con lo requerido por la ***ABI*** , de la entrada de datos, y vuelca el resultado en el archivo de salida. Ante un error, la función volverá 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.

  - base64_decode
    La función recibe por parámetro a *infd* y *outfd*, los *file descriptors* correspondientes a los archivos de entrada y salida pre-abiertos por ***main.c***. La función función realiza el decoding a base 64 en assembly ***MIPS***, cumpliendo con lo requerido por la ***ABI***, de la entrada de datos, y vuelca el resultado en el archivo de salida. Ante un error, la función volverá 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.

4. 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(default) 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.

5. Compilación del programa

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

```
$ ggcc -std=c99 -Wall -pedantic -Werror -O0 -ggdb -o tp1 main.c base64
```

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.

6. Pruebas realizadas

6.1. 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 "################################################################################"

echo

echo "Se guardaran los archivos resultantes de los tests en el directorio outputs"

if [ -d "./outputs" ]
then
   echo "El directorio outputs existe, por lo tanto se elimina su contenido."
   rm -r outputs/*
else
   echo "El directorio outputs no existe, por lo tanto se creara."
   mkdir outputs
fi
echo
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
```

```
41
42 echo "#################### FIN test ejercicio 1 archivo vacio sin -a
       ####################"
43 echo
       "################################################################################"

44 echo "#################### COMIENZA test ejercicio 2 stdin y stdout
       ####################"
45
46 echo -n Man | ./tp1 -a encode > ./outputs/outputEncode.txt
47 if diff -b ./outputs-aut/outputEncode-aut.txt ./outputs/outputEncode.txt; then echo
       "[OK]"; else
48         echo ERROR;
49 fi
50
51 echo "######################## FIN test ejercicio 2 stdin y stdout
       ########################"
52 echo
       "################################################################################"

53 echo "#---------# COMIENZA test ejercicio 3 stdin y stdout #--------#"
54
55 echo -n TWFu | ./tp1 -a decode > ./outputs/outputDecode.txt
56 if diff -b ./outputs-aut/outputDecode-aut.txt ./outputs/outputDecode.txt; then echo
       "[OK]"; else
57         echo ERROR;
58 fi
59
60 echo "######################## FIN test ejercicio 3 stdin y stdout
       ########################"
61 echo
       "################################################################################"

62 echo "#----------# COMIENZA test menu help (-h) #----------#"
63
64 ./tp1 -h > ./outputs/outputMenuH.txt
65
66 if diff -b ./outputs-aut/outputMenuHelp-aut.txt ./outputs/outputMenuH.txt; then echo
       "[OK]"; else
67         echo ERROR;
68 fi
69
70 echo "############################ FIN test menu version (-h)
       ############################"
71 echo
       "################################################################################"

72 echo "######################### COMIENZA test menu help (--help)
       #########################"
73
74 ./tp1 --help > ./outputs/outputMenuHelp.txt
75
76 if diff -b ./outputs-aut/outputMenuHelp-aut.txt ./outputs/outputMenuHelp.txt; then
       echo "[OK]"; else
77                 echo ERROR;
78 fi
79
```

```
80 echo "########################## FIN test menu version (--help)
       ##########################"
81 echo
       "################################################################################################"

82 echo "######################### COMIENZA test menu version (-V)
       ##########################"
83
84 ./tp1 -V > ./outputs/outputMenuV.txt
85
86 if diff -b ./outputs-aut/outputMenuVersion-aut.txt ./outputs/outputMenuV.txt; then
       echo "[OK]"; else
87                 echo ERROR;
88 fi
89 echo "############################ FIN test menu version (-V)
       ############################"
90 echo
       "################################################################################################"

91 echo "###################### COMIENZA test menu version (--version)
       ######################"
92
93 ./tp1 --version > ./outputs/outputMenuVersion.txt
94
95 if diff -b ./outputs-aut/outputMenuVersion-aut.txt ./outputs/outputMenuVersion.txt;
       then echo "[OK]"; else
96                 echo ERROR;
97 fi
98 echo "######################## FIN test menu version (--version)
       ########################"
99 echo
       "################################################################################################"

100 echo "###################### COMIENZA test ejercicio encode/decode
       ######################"
101
102 echo xyz | ./tp1 -a encode | ./tp1 -a decode | od -t c
103
104 echo "############################ FIN test ejercicio encode
       ############################"
105 echo
       "################################################################################################"

106 echo "#################### COMIENZA test ejercicio longitud maxima 76
       ####################"
107
108 yes | head -c 1024 | ./tp1 -a encode > ./outputs/outputSize76.txt
109
110 if diff -b ./outputs-aut/outputSize76-aut.txt ./outputs/outputSize76.txt; then echo
       "[OK]"; else
111                 echo ERROR;
112 fi
113
114 echo "###################### FIN test ejercicio longitud maxima 76
       ######################"
115 echo
       "################################################################################################"
```

```
116 echo "######################## COMIENZA test ejercicio decode 1024
    ######################"
117
118 yes | head -c 1024 | ./tp1 -a encode | ./tp1 -a decode | wc -c > ./outputs/
    outputSize1024.txt
119
120 if diff -b ./outputs-aut/outputSize1024-aut.txt ./outputs/outputSize1024.txt; then
    echo "[OK]"; else
121                 echo ERROR;
122 fi
123
124 echo "########################## FIN test ejercicio decode 1024
    ##########################"
125 echo
    "#################################################################################"

126 echo "################### COMIENZA test ejercicio encode/decode random
    ###################"
127
128 n=1;
129 while :; do
130 #while [$n -lt 10]; do
131 head -c $n </dev/urandom >/tmp/in.bin;
132 ./tp1 -a encode -i /tmp/in.bin -o /tmp/out.b64;
133 ./tp1 -a decode -i /tmp/out.b64 -o /tmp/out.bin;
134 if diff /tmp/in.bin /tmp/out.bin; then :; else
135 echo ERROR: $n;
136 break;
137 fi
138 echo [OK]: $n;
139 n=`expr $n + 1`;
140 rm -f /tmp/in.bin /tmp/out.b64 /tmp/out.bin
141 done
142
143 echo "###################### FIN test ejercicio encode/decode random
    ######################"
144 echo
    "#################################################################################"


145
146 echo
147
148 echo
    "#################################################################################"

149 echo "############################ FIN Tests automaticos
    ##############################"
150 echo
    "#################################################################################"
```

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

6.2. Casos de prueba

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.

- ***Mensaje de ayuda***
  Ejecutando el comando:

```
$ ./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***
  Ejecutando el comando:

```
$ ./tp1 -V  o ./tp1 --version
```

```
Version: 0.2
```

- ***Archivo vacío***
  Ejecutando el comando:

```
$ ./tp1 -a encode -i ./outputs-aut/zero.txt -o ./outputs-aut/zero.txt.b64
```

- ***Codificación***
  Ejecutando el comando:

```
$ echo -n Man | ./tp1 -a encode
```

```
TWFu
```

- ***Decodificación***
  Ejecutando el comando:

```
$ echo -n TWFu | ./tp1 -a decode
```

```
Man
```

- ***Test de verificación para que el programa no genere líneas con más de 76 de longitud***
  Ejecutando el comando:

```
$ yes | head -c 1024 | ./tp1 -a encode

```

```
eQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkK
eQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkK
eQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkK
eQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkK
eQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkK
eQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkK
eQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkK
eQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkK
eQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkK
eQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkK
eQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkK
eQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkK
eQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkK
eQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkK
eQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkK
eQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkK
eQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkK
eQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkK
eQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5CnkKeQp5Cg==
```

- ***Verificamos que la cantidad de líneas sea 1024***
  Ejecutando el comando:

```
$ yes | head -c 1024 | ./tp1 -a encode | ./tp1 -a decode | wc -c

```

```
    1024
```

7. 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("  tp1 -a encode -i ~/input -o ~/output\n");
167     printf("  tp1 -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,"  tp1 -a encode -i ~/input -o ~/output\n");
255     fprintf(stderr,"  tp1 -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     while((arg_opt =
325                     getopt_long(argc, argv, arg_opt_str, arg_long, &arg_opt_idx)) !=
    -1 && !should_finish) {
326         switch(arg_opt){
327                 case 'i':
328                 if(strcmp(optarg, "-") != 0)
329                         CommandSetInput(&cmd_opt, optarg);
330                 break;
331                 case 'o':
332                 if(strcmp(optarg, "-") != 0)
333                         CommandSetOutput(&cmd_opt, optarg);
334                 break;
335                 case 'h':
336                         CommandHelp();
337                 should_finish = TRUE;
338                 break;
339                 case 'V':
340                         CommandVersion();
341                 should_finish = TRUE;
```

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

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 28

#define OFFSET_RA_DECODECHAR 24
#define OFFSET_FP_DECODECHAR 20
#define OFFSET_GP_DECODECHAR 16
#define OFFSET_RETURN_DECODECHAR 8
#define OFFSET_I_DECODECHAR 4
#define OFFSET_CHARACTER_DECODECHAR 0

#define STACK_FRAME_DECODE 32//64

#define OFFSET_BUFFER_OUTPUT_ENCODE 36//68
#define OFFSET_BUFFER_INPUT_ENCODE 32//64
#define OFFSET_RA_DECODE 24//28//60
#define OFFSET_FP_DECODE 20//24//56
#define OFFSET_GP_DECODE 16//20//52
//#define OFFSET_S0_DECODE 16//48
#define OFFSET_CHAR1_AUX_ENCODE 13//37
#define OFFSET_CHAR0_AUX_ENCODE 12//36
#define OFFSET_I_DECODE 8//32
#define OFFSET_CHARS3_ENCODE 7//27
#define OFFSET_CHARS2_ENCODE 6//26
#define OFFSET_CHARS1_ENCODE 5//25
#define OFFSET_CHARS0_ENCODE 4//24
#define OFFSET_RETURN_ENCODE 0//20
```

```
57
58          .data
59          .align  2
60  sep:    .ascii  "\n"
61  pad:    .ascii  "="
62          .globl errmsg
63  errmsg:
64          .word base64_ok, base64_err1, base64_err2, base64_err3
65          .size errmsg, 16
66  base64_ok:
67          .asciiz "OK"
68  base64_err1:
69          .asciiz "I/O Error"
70  base64_err2:
71          .asciiz "File no es multiplo de 4"
72  base64_err3:
73          .asciiz "File contiene caracteres invalidos"
74          .text
75          .align  2
76          .globl  base64_encode
77          .ent    base64_encode
78  base64_encode:
79          // debugging info: descripcion del stack frame
80          .frame  $fp, 40, ra     // $fp: registro usado como frame pointer
81                                  // 32: tamañodel stack frame
82                                  // ra: registro que almacena el return address
83          // bloque para codigo PIC
84          .set    noreorder       // apaga reordenamiento de instrucciones
85          .cpload t9              // directiva usada para codigo PIC
86          .set    reorder         // enciende reordenamiento de instrucciones
87          // creo stack frame
88          subu    sp, sp, 40      // 4 (SRA) + 2 (LTA) + 4 (ABA)
89          // directiva para codigo PIC
90          .cprestore 24           // inserta aqui "sw gp, 24(sp)",
91                                  // mas "lw gp, 24(sp)" luego de cada jal.
92          // salvado de callee-saved regs en SRA
93          sw      $fp, 28(sp)
94          sw      ra, 32(sp)
95          // de aqui al fin de la funcion uso $fp en lugar de sp.
96          move    $fp, sp
97          // salvo 1er arg (siempre)
98          sw      a0, 40($fp)     // a0 contiene file input
99          sw      a1, 44($fp)     // a1 contiene file output
100         li      s1, 0           // count = 0
101
102         //Limpio input para read
103 base64_encode_loop:
104         sw      zero, 20($fp)   //input = 0
105
106         //Leo archivo
107         lw      a0, 40($fp)
108         addi    a1, $fp, 20
109         li      a2, 3
110         li      v0, SYS_read
111         syscall
112         beqz    v0, base64_encode_return_ok     //Si no lei nada finalizo
113         blt     v0, 0, base64_encode_io_error
114         //Paso parametros y llamo a Encode
```

```
115         addi    a0, $fp, 20
116         move    a1, v0
117         addi    a2, $fp, 16
118         la      t9, Encode
119         jal     ra, t9
120
121         //Grabo en file
122         lw      a0, 44($fp)     // File descriptor out
123         addi    a1, $fp, 16     // Apunto a buffer out
124         li      a2, 4           // length = 4
125         li      v0, SYS_write
126         syscall
127         addi s1, s1, 1          // count++
128         bne s1, 18, base64_encode_loop // Si count = 18 agrego un salto
129         lw      a0, 44($fp)     // file out
130         la      a1, sep         // sep = '\n'
131         li      a2, 1           // length = 4
132         li      v0, SYS_write
133         syscall
134         li      s1, 0
135         j base64_encode_loop
136
137 base64_encode_return_ok:        // return;
138         li      v0, 0
139         j base64_encode_return
140 base64_encode_io_error:
141         li      v0, 1
142         // restauro callee-saved regs
143 base64_encode_return:
144         lw      gp, 24(sp)
145         lw      $fp, 28(sp)
146         lw      ra, 32(sp)
147         // destruyo stack frame
148         addu    sp, sp, 40
149         // vuelvo a funcion llamante
150         jr      ra
151         .end    base64_encode
152         .size   base64_encode, .-base64_encode
153
154         .globl  base64_decode
155         .ent    base64_decode
156 base64_decode:
157         // debugging info: descripcion del stack frame
158         .frame  $fp, 56, ra     // $fp: registro usado como frame pointer
159                                 // 56: tamañodel stack frame
160                                 // ra: registro que almacena el return address
161         // bloque para codigo PIC
162         .set    noreorder       // apaga reordenamiento de instrucciones
163         .cpload t9              // directiva usada para codigo PIC
164         .set    reorder         // enciende reordenamiento de instrucciones
165         // creo stack frame
166         subu    sp, sp, 56      // 8 (SRA) + 2 (LTA) + 4 (ABA)
167         // directiva para codigo PIC
168         .cprestore 44           // inserta aqui "sw gp, 24(sp)",
169                                 // mas "lw gp, 24(sp)" luego de cada jal.
170         // salvado de callee-saved regs en SRA
171         sw      $fp, 48(sp)
172         sw      ra, 52(sp)
```

```
173         sw      s1, 24(sp)
174         sw      s2, 28(sp)
175         sw      s3, 32(sp)
176         sw      s4, 36(sp)
177         sw      s5, 40(sp)
178         // de aqui al fin de la funcion uso $fp en lugar de sp.
179         move    $fp, sp
180         // salvo 1er arg (siempre)
181         sw      a0, 56($fp)     // a0 contiene file input
182         sw      a1, 60($fp)     // a1 contiene file output
183         li      s1, 0           // count = 0
184         la      s5, pad
185         lbu     s5, 0(s5)
186         //Limpio input para read
187 base64_decode_loop:
188         sw      zero, 20($fp)   //input = 0
189
190         //Leo archivo
191         lw      a0, 56($fp)
192         addi    a1, $fp, 20
193         li      a2, 4
194         li      v0, SYS_read
195         syscall
196         beqz    v0, base64_decode_return_ok     //Si no lei nada finalizo
197         blt     v0, 0, base64_decode_ioerror
198         blt     v0, 4, base64_decode_nomult
199         //Controlo si hay padding
200         li      s3, 0                           //s3 = cant de padding a borrar
201         lbu     s2, 23($fp)                     //s2 aux control padding
202         bne     s2, s5, ctl1
203         addi    s3, s3, 1
204 ctl1:
205         lbu     s2, 22($fp)                     //s2 aux control padding
206         bne     s2, s5, ctl2
207         addi    s3, s3, 1
208 ctl2:
209         //Controlo salto de linea
210         addi s1, s1, 1          // count++
211         bne s1, 18, not_sep     // Si count = 18 elimino un caracter
212         lw      a0, 56($fp)     // file in
213         addi    a1, $fp, 16     // grabo en out buffer, luego se pisa
214         li      a2, 1           // length = 1
215         li      v0, SYS_read
216         syscall
217         li      s1, 0
218         //Paso parametros y llamo a Decode
219 not_sep:
220         addi    a0, $fp, 20
221         addi    a1, $fp, 16
222         la      t9, Decode
223         jal     ra, t9
224
225         //Chequeo error
226         beq     v0, DECODE_ERROR, base64_decode_decode_err
227
228         //Grabo en file
229         lw      a0, 60($fp)     // File descriptor out
230         addi    a1, $fp, 16     // Apunto a buffer out
```

```
231         li      s4, 3
232         subu    a2, s4, s3      // a2 = 3 - cant de padding
233         li      v0, SYS_write
234         syscall
235         j base64_decode_loop
236 base64_decode_return_ok:
237         li v0, 0
238         j base64_decode_return
239 base64_decode_ioerror:
240         li v0, 1
241         j base64_decode_return
242 base64_decode_nomult:
243         li v0, 2
244         j base64_decode_return
245 base64_decode_decode_err:
246         li v0, 3
247 base64_decode_return:   // return;
248         // restauro callee-saved regs
249         lw      gp, 44(sp)
250         lw      $fp, 48(sp)
251         lw      ra, 52(sp)
252         lw      s1, 24(sp)
253         lw      s2, 28(sp)
254         lw      s3, 32(sp)
255         lw      s4, 36(sp)
256         lw      s5, 40(sp)
257         // destruyo stack frame
258         addu    sp, sp, 56
259         // vuelvo a funcion llamante
260         jr      ra
261         .end    base64_decode
262         .size   base64_decode, .-base64_decode
263
264         //.file 1 "encode.c"
265         //.section .mdebug.abi32
266         //.previous
267         //.abicalls
268         .data
269         .align  2
270         .type   encoding_table, @object
271         .size   encoding_table, 64
272 encoding_table:
273         .byte   65
274         .byte   66
275         .byte   67
276         .byte   68
277         .byte   69
278         .byte   70
279         .byte   71
280         .byte   72
281         .byte   73
282         .byte   74
283         .byte   75
284         .byte   76
285         .byte   77
286         .byte   78
287         .byte   79
288         .byte   80
```

```
289         .byte   81
290         .byte   82
291         .byte   83
292         .byte   84
293         .byte   85
294         .byte   86
295         .byte   87
296         .byte   88
297         .byte   89
298         .byte   90
299         .byte   97
300         .byte   98
301         .byte   99
302         .byte   100
303         .byte   101
304         .byte   102
305         .byte   103
306         .byte   104
307         .byte   105
308         .byte   106
309         .byte   107
310         .byte   108
311         .byte   109
312         .byte   110
313         .byte   111
314         .byte   112
315         .byte   113
316         .byte   114
317         .byte   115
318         .byte   116
319         .byte   117
320         .byte   118
321         .byte   119
322         .byte   120
323         .byte   121
324         .byte   122
325         .byte   48
326         .byte   49
327         .byte   50
328         .byte   51
329         .byte   52
330         .byte   53
331         .byte   54
332         .byte   55
333         .byte   56
334         .byte   57
335         .byte   43
336         .byte   47
337
338         .type   encoding_table_size, @object
339         .size   encoding_table_size, 4
340 encoding_table_size:
341         .word   64
342
343         .text
344         .align  2
345         .globl  Encode
346         .ent    Encode
```

```
347
348         ///////////////  Función Encode ///////////////
349
350 Encode:
351         .frame  $fp,STACK_FRAME_ENCODE,ra                // vars= 8, regs= 2/0, args=
    0, extra= 8
352         //.mask 0x50000000,-4
353         //.fmask        0x00000000,0
354         .set    noreorder
355         .cpload t9
356         .set    reorder
357
358         // Creación del stack frame
359         subu    sp,sp,STACK_FRAME_ENCODE
360
361         .cprestore 0
362         sw      $fp,OFFSET_FP_ENCODE(sp)
363         sw      gp,OFFSET_GP_ENCODE(sp)
364
365         // De aquí al final de la función uso $fp en lugar de sp.
366         move    $fp,sp
367
368         // Guardo el primer parámetro *buffer
369         sw      a0,OFFSET_BUFFER_ENCODE($fp)
370         // Guardo el segundo parámetro 'length'(cantidad de caracteres)
371         sw      a1,OFFSET_LENGTH_ENCODE($fp)
372         // Guardo el puntero al array de salida(output)
373         sw      a2,OFFSET_OUTPUT_ENCODE($fp)
374
375         // Cargo en v0 el puntero al buffer.
376         lw      v0,OFFSET_BUFFER_ENCODE($fp)
377         // Cargo en v0 el 1er byte del buffer.
378         lbu     v0,0(v0)
379         // Guardo el 1er byte en el stack frame
380         sb      v0,OFFSET_B1($fp)
381         // Cargo nuevamente la dirección del buffer.
382         lw      v0,OFFSET_BUFFER_ENCODE($fp)
383         // Aumento en 1(1 byte) la dirección del buffer.
384         // Me muevo por el array del buffer.
385         addu    v0,v0,1
386         // Cargo el 2do byte del buffer.
387         lbu     v0,0(v0)
388         // Guardo el 2do byte en el stack frame.
389         sb      v0,OFFSET_B2($fp)
390         // Cargo nuevamente la dirección del buffer.
391         lw      v0,OFFSET_BUFFER_ENCODE($fp)
392         // Aumento en 2(2 byte) la dirección del buffer.
393         // Me muevo por el array del buffer.
394         addu    v0,v0,2
395         // Cargo el 2do byte del buffer.
396         lbu     v0,0(v0)
397         // Guardo el 3er byte en stack frame.
398         sb      v0,OFFSET_B3($fp)
399         // Cargo en v0 el 1er byte.
400         lbu     v0,OFFSET_B1($fp)
401         // Muevo 2 'posiciones' hacia la derecha(shift 2).
402         srl     v0,v0,2
403         // Guardo el nuevo byte en una variable auxiliar.
```

```
404         sb      v0,OFFSET_B1_AUX($fp)
405         // Cargo en v1 el puntero al output.
406         lw      v1,OFFSET_OUTPUT_ENCODE($fp)
407         // Cargo en v0 el byte shifteado.
408         lbu     v0,OFFSET_B1_AUX($fp)
409         // Cargo en v0 el caracter(byte) de la tabla encoding(encoding_table)
410         lbu     v0,encoding_table(v0)
411         // Cargo en v0 el 1er byte de la dirección del output.
412         sb      v0,0(v1)
413         // Cargo en v0 el 1er byte del buffer nuevamente.
414         lbu     v0,OFFSET_B1($fp)
415         // Muevo 6 'posiciones' hacia la izquierda(shift 6).
416         sll     v0,v0,6
417         // Guardo el resultado del shift en el Stack Frame.
418         sb      v0,OFFSET_B2_AUX($fp)
419         // Cargo el byte sin signo shifteado.
420         lbu     v0,OFFSET_B2_AUX($fp)
421         // Muevo 2 'posiciones' hacia la derecha(shift 2).
422         srl     v0,v0,2
423         // Guardo el nuevo resultado del shift en el Stack Frame.
424         sb      v0,OFFSET_B2_AUX($fp)
425         // Cargo el 2do byte del buffer en v0.
426         lbu     v0,OFFSET_B2($fp)
427         // Hago un shift left de 4 posiciones.
428         srl     v0,v0,4
429         // Cargo en v1 el resultado(byte) del shift right 2.
430         lbu     v1,OFFSET_B2_AUX($fp)
431         // Hago un 'or' entre v1 y v0 para obtener el 2 indice de la tabla.
432         or      v0,v1,v0
433         //(*) Guardo en stack frame(12) el resultado del 'or' anterior.
434         sb      v0,OFFSET_B2_AUX($fp)
435         // Cargo en v0 el puntero al output.
436         lw      v0,OFFSET_OUTPUT_ENCODE($fp)
437         // Cargo en v1 la dirección del output + 1(1byte).
438         addu    v1,v0,1
439         // Cargo en v0 el ultimo resultado del shift(*)
440         lbu     v0,OFFSET_B2_AUX($fp)
441         // Cargo en v0 el caracter(byte) de la tabla encoding(encoding_table)
442         lbu     v0,encoding_table(v0)
443         // Salvo en el output array(output[1]) el valor del encoding_table
444         sb      v0,0(v1)
445         // Cargo en v0 el puntero al output.
446         lw      v0,OFFSET_OUTPUT_ENCODE($fp)
447         // Sumo 2 a la dirección del output(output[2]).
448         // Me desplazo dentro del output array.
449         addu    v1,v0,2
450         // Cargo en v0 el caracter ascii 61('=').
451         li      v0,EQUAL_CHAR                   // 0x3d
452         // Salvo en el output array(output[2]) el valor '='.
453         sb      v0,0(v1)
454         // Cargo en v0 el puntero al output.
455         lw      v0,OFFSET_OUTPUT_ENCODE($fp)
456         // Sumo 3 a la dirección del output(output[3]).
457         // Me desplazo dentro del output array.
458         addu    v1,v0,3
459         // Cargo en v0 el caracter ascii 61('=').
460         li      v0,EQUAL_CHAR                   // 0x3d
461         // Salvo en el output array(output[3]) el valor '='.
```

```
462         sb      v0,0(v1)
463         // Cargo en v1 el parametro length.
464         lw      v1,OFFSET_LENGTH_ENCODE($fp)
465         // Cargo en v0 el valor 3.
466         li      v0,3                    // 0x3
467         // Si el length == 3 salto a buffer_size_2.
468         bne     v1,v0,buffer_size_2
469         // Si el tamanio del buffer es 3 continuo NO salto.
470         // Cargo en v0 el 3er byte del buffer.
471         lbu     v0,OFFSET_B3($fp)
472         // Hago un shift right de 6.
473         srl     v0,v0,6
474         // Guardo el nuevo byte en el stack frame.
475         sb      v0,OFFSET_B3_AUX($fp)
476         // Cargo el 2do byte del buffer en v0.
477         lbu     v0,OFFSET_B2($fp)
478         // Hago un shift left de 4.
479         sll     v0,v0,4
480         // Guardo en el stack frame(14) el nuevo valor.
481         sb      v0,OFFSET_B3_AUX_2($fp)
482         // Cargo en v0 el byte shifteado sin signo.
483         lbu     v0,OFFSET_B3_AUX_2($fp)
484         // Hago un shift rigth de 2.
485         srl     v0,v0,2
486         // Guardo en el stack frame(14) el valor shifteado.
487         sb      v0,OFFSET_B3_AUX_2($fp)
488         // Cargo en v1 el valor del SF(13)
489         lbu     v1,OFFSET_B3_AUX($fp)
490         // Idem en v0(13).
491         lbu     v0,OFFSET_B3_AUX_2($fp)
492         // Hago un 'or' y almaceno en v0.
493         or      v0,v1,v0
494         // Guardo en el stack frame(13) el resultado del 'or'.
495         sb      v0,OFFSET_B3_AUX($fp)
496         // Cargo en v0 el puntero al output.
497         lw      v0,OFFSET_OUTPUT_ENCODE($fp)
498         // Me desplazo por el vector 'output' en 2 posiciones(output[2]).
499         addu    v1,v0,2
500         // Cargo en v0 el resultado del 'or' anterior.
501         lbu     v0,OFFSET_B3_AUX($fp)
502         // Busco en la tabla de encoding el caracter que corresponde.
503         // Luego cargo el byte en v0.
504         lbu     v0,encoding_table(v0)
505         // Guardo el valor recuperado de la tabla encoding_table en el output[2].
506         sb      v0,0(v1)
507         // Cargo en v0 el 3er byte del buffer.
508         lbu     v0,OFFSET_B3($fp)
509         // Hago un shift left de 2.
510         sll     v0,v0,2
511         // Guardo en el stack frame el valor shifteado.
512         sb      v0,OFFSET_B4_AUX($fp)
513         // Cargo el byte sin signo shifteado.
514         lbu     v0,OFFSET_B4_AUX($fp)
515         // Hago un shift rigth de 2.
516         srl     v0,v0,2
517         // Guardo en el stack frame el valor shifteado.
518         sb      v0,OFFSET_B4_AUX($fp)
519         // Cargo en v0 el puntero al output.
```

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

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

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

```
692
693         ///////////////  Begin Función Decode ///////////////
694
695         .align  2
696         .globl  Decode
697         .ent    Decode
698 Decode:
699         .frame  $fp,STACK_FRAME_DECODE,ra                // vars= 24, regs= 4/0, args=
        16, extra= 8
700         //.mask 0xd0010000,-4
701         //.fmask        0x00000000,0
702         .set    noreorder
703         .cpload t9
704         .set    reorder
705
706         // Creación del stack frame
707         subu    sp,sp,STACK_FRAME_DECODE
708         .cprestore 16
709
710         sw      ra,OFFSET_RA_DECODE(sp)
711         sw      $fp,OFFSET_FP_DECODE(sp)
712         sw      gp,OFFSET_GP_DECODE(sp)
713         //sw    s0,OFFSET_S0_DECODE(sp)
714
715         // De aquí al final de la función uso $fp en lugar de sp.
716         move    $fp,sp
717
718         // Guardo en el stack frame los parámetros recibidos.
719         // a0=puntero a buffer_input
720         sw      a0,OFFSET_BUFFER_INPUT_ENCODE($fp)
721         // Guardo en el stack frame los parámetros recibidos.
722         // a1=puntero a buffer_output
723         sw      a1,OFFSET_BUFFER_OUTPUT_ENCODE($fp)
724         // Guardo un 0 en el stack frame(OFFSET_I_DECODE). Inicializo 'i'.
725         sw      zero,OFFSET_I_DECODE($fp)
726 loop_decode_char:
727         // Cargo en v0 el valor de 'i' guardado en el stack frame.
728         lw      v0,OFFSET_I_DECODE($fp)
729         // Si (i < SIZE_DECODE_CHAR), guardo TRUE en v0, sino FALSE.
730         sltu    v0,v0,SIZE_DECODE_CHAR
731         // Salto a if_decode_char si sigo dentro del bucle.
732         bne     v0,zero,if_decode_char
733         // Salto a main_shift
734         b       main_shift
735 if_decode_char:
736         // Cargo en v1 el valor de 'i'.
737         lw      v1,OFFSET_I_DECODE($fp)
738         // Cargo en v0 el valor de fp + OFFSET_CHARS0_ENCODE ???
739         addu    v0,$fp,OFFSET_CHARS0_ENCODE
740         // Cargo en s0 el valor de buf_input[i]
741         addu    s0,v0,v1
742         // Cargo en v1 el puntero a buf_input
743         lw      v1,OFFSET_BUFFER_INPUT_ENCODE($fp)
744         // Cargo en v0 el valor de 'i'.
745         lw      v0,OFFSET_I_DECODE($fp)
746         // Me desplazo por el vector(buf_input[i])
747         addu    v0,v1,v0
748         // Cargo en v0 el valor del buf_input[i](1 byte).
```

```
749         lb      v0,0(v0)
750         // Asigna el valor del byte a a0 antes de llamar a la función.
751         move    a0,v0
752         // Carga en t9 la direccion de la funcion DecodeChar.
753         la      t9,DecodeChar
754         // Hace el llamado a la función.
755         jal     ra,t9
756         // Guardo en s0 el resultado de la función.
757         // El valor regresa en el registro v0
758         sb      v0,0(s0)
759         // Cargo en v1 el valor de 'i'.
760         lw      v1,OFFSET_I_DECODE($fp)
761         // Cargo en v0 el valor de fp + OFFSET_CHARS_ENCODE ???
762         addu    v0,$fp,OFFSET_CHARS0_ENCODE
763         // Cargo en v0 el valor de chars[i](direccion).
764         addu    v0,v0,v1
765         // Cargo en v1 el byte apuntado.
766         lbu     v1,0(v0)
767         // Cargo en v0 el DECODE_ERROR
768         li      v0,DECODE_ERROR                 // 0x64
769         // Si chars[i] != DECODE_ERROR salto a increase_index_decode
770         bne     v1,v0,increase_index_decode
771         // Guarda en el stack frame un 0.
772         sw      zero,OFFSET_RETURN_ENCODE($fp)
773         // Si chars[i] == DECODE_ERROR retorno un 0.
774         b       return_zero
775 increase_index_decode:
776         // Cargo en v0 el valor de 'i'.
777         lw      v0,OFFSET_I_DECODE($fp)
778         // Sumo en 1 el valor de 'i'(i++).
779         addu    v0,v0,1
780         // Guardo el valor modificado en el stack frame.
781         sw      v0,OFFSET_I_DECODE($fp)
782         // Salto a loop_decode_char
783         b       loop_decode_char
784 main_shift:
785         // Cargo en v0 la dirección de chars[0]
786         lbu     v0,OFFSET_CHARS0_ENCODE($fp)
787         // Hago un shift left logical de SHIFT_2 y lo asigno a v0.
788         sll     v0,v0,SHIFT_2
789         // Guardo el valor en el stack frame.
790         sb      v0,OFFSET_CHAR0_AUX_ENCODE($fp)
791         // Cargo el valor de chars[1] en v0.
792         lbu     v0,OFFSET_CHARS1_ENCODE($fp)
793         // Hago un shift left logical de SHIFT_2 y lo asigno a v0.
794         srl     v0,v0,SHIFT_4
795         // Guardo en el stack frame el valor shifteado.
796         sb      v0,OFFSET_CHAR1_AUX_ENCODE($fp)
797         // Cargo en v1 char1_aux(chars[0] luego de ser shifteado).
798         lbu     v1,OFFSET_CHAR0_AUX_ENCODE($fp)
799         // Cargo en v0 char2_aux(chars[1] luego de ser shifteado).
800         lbu     v0,OFFSET_CHAR1_AUX_ENCODE($fp)
801         // Hago un or de v1 y v0 y lo asigno a v0.
802         or      v0,v1,v0
803         // Guardo en valor en el stack frame.
804         sb      v0,OFFSET_CHAR0_AUX_ENCODE($fp)
805         // Cargo en v1 el puntero al buffer_output.
806         lw      v1,OFFSET_BUFFER_OUTPUT_ENCODE($fp)
```

```
807         // Cargo en v0 char1_aux(chars[0] luego de ser shifteado).
808         lbu     v0,OFFSET_CHAR0_AUX_ENCODE($fp)
809         // Guardo en el vector buffer_output el valor de char1_aux.
810         sb      v0,0(v1)
811         // Cargo el valor de chars[1] en v0.
812         lbu     v0,OFFSET_CHARS1_ENCODE($fp)
813         // Hago un shift left de 4 posiciones y lo guardo en v0.
814         sll     v0,v0,SHIFT_4
815         // Guardo en el stack frame el valor shifteado.
816         sb      v0,OFFSET_CHAR0_AUX_ENCODE($fp)
817         // Cargo en v0 chars[2].
818         lbu     v0,OFFSET_CHARS2_ENCODE($fp)
819         // Hago un shift rigth de 2 de chars[2] y lo guardo en v0.
820         srl     v0,v0,SHIFT_2
821         // Guardo en stack frame el valor shifteado.
822         sb      v0,OFFSET_CHAR1_AUX_ENCODE($fp)
823         // Cargo en v1 y v0 los valores shifteados anteriormente.
824         lbu     v1,OFFSET_CHAR1_AUX_ENCODE($fp)
825         lbu     v0,OFFSET_CHAR0_AUX_ENCODE($fp)
826         // Hago un or de v1 y v0 y lo asigno a v0.
827         or      v0,v1,v0
828         // Vuelvo a guardar en el stack frame el resultado del or.
829         // (**)
830         sb      v0,OFFSET_CHAR1_AUX_ENCODE($fp)
831         // Cargo en v0 el puntero al buffer_output.
832         lw      v0,OFFSET_BUFFER_OUTPUT_ENCODE($fp)
833         // Sumo 1 al puntero para desplazarme dentro del vector.
834         // Luego asigno el resultado a v1.
835         addu    v1,v0,1
836         // Cargo en v0 el resultado de (**).
837         lbu     v0,OFFSET_CHAR1_AUX_ENCODE($fp)
838         // Guardo en el vector buffer_output el valor (**).
839         sb      v0,0(v1)
840         // Cargo en v0 chars[2]
841         lbu     v0,OFFSET_CHARS2_ENCODE($fp)
842         // Hago un shift left de 6.
843         sll     v0,v0,SHIFT_6
844         // Guardo en el stack frame el valor shifteado.
845         // (***)
846         sb      v0,OFFSET_CHAR0_AUX_ENCODE($fp)
847         // Cargo en v0 el puntero al buffer_output.
848         lw      v0,OFFSET_BUFFER_OUTPUT_ENCODE($fp)
849         // Sumo 2 al puntero para desplazarme dentro del vector buffer_output.
850         // Luego asigno el resultado a a0.
851         addu    a0,v0,2
852         // Cargo en v1 el ultimo valor shifteado (***).
853         lbu     v1,OFFSET_CHAR0_AUX_ENCODE($fp)
854         // Cargo en v0 chars[3]
855         lbu     v0,OFFSET_CHARS3_ENCODE($fp)
856         // Hago un or de v1 y v0 y lo asigno a v0.
857         or      v0,v1,v0
858         // Guardo en el vector buffer_output el resultado del or.
859         sb      v0,0(a0)
860         // Cargo en v0 el inmediato 1(RETURNO_OK).
861         li      v0,RETURNO_OK                   // 0x1
862         // Guardo en el stack frame el valor de retorno.
863         sw      v0,OFFSET_RETURN_ENCODE($fp)
864 return_zero:
```

```
865         // Cargo en v0 el valor salvado en el stack frame(0).
866         lw      v0,OFFSET_RETURN_ENCODE($fp)
867         move    sp,$fp
868
869         // Restauro ra,fp y gp.
870         lw      ra,OFFSET_RA_DECODE(sp)
871         lw      $fp,OFFSET_FP_DECODE(sp)
872         //lw    s0,OFFSET_S0_DECODE(sp)
873
874         // Destruyo el stack frame.
875         addu    sp,sp,STACK_FRAME_DECODE
876         // Devuelvo el control a la función llamante.
877         j       ra
878
879         .end    Decode
880         .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**
32	character	**ABA (caller)**
28	////////////////////	**SRA**
24	ra	
20	fp	
16	gp	
12	////////////////////	**LTA**
8	return	
4	i	
0	character	

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**
36	*buffer_output	**ABA (caller)**
32	*buffer_input	
28	////////////////////////	**SRA**
24	ra	
20	fp	
16	gp	
15	////////////////////////	**LTA**
14	////////////////////////	
13	char1_aux	
12	char0_aux	
8	i	
7	chars3	
6	chars2	
5	chars1	
4	chars0	
0	return	

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