Trabajo Práctico #1:

Conjunto de instrucciones MIPS

Julian Ferres, *Padrón Nro. 101483*
julianferres@gmail.com

Cecilia María Hortas, *Padrón Nro. 100687*
ceci.hortas@gmail.com

Matías Ezequiel Scakosky, *Padrón Nro. 99627*
scakosky@hotmail.com

2do. Cuatrimestre de 2018
66.20 Organización de Computadoras – Práctica Martes
Facultad de Ingeniería, Universidad de Buenos Aires

Abstract

Se propone como objetivo del Trabajo Práctico realizar un programa que utilice código en C y en Assembly MIPS para codificar y decodificar información en base64. Se utiliza el programa GXemul para simular el entorno de desarrollo, una máquina MIPS que corre una versión de NetBSD.

Contents

1 Introducción **3**

2 Documentación relevante al diseño e implementación del programa **3**
2.1 Diseño . . . 3
2.2 Detalle de implementación . . . 3
2.2.1 Subrutina encode . . . 4
2.2.2 Subrutina decode . . . 5
2.2.3 Función principal main . . . 5

3 Comandos para compilar el programa **5**

4 Corridas de prueba **6**

5 Código principal en C **6**
5.1 main.c . . . 6
5.2 base64.h . . . 8

6 Código implementado en MIPS32 **9**
6.1 code.s . . . 9
6.2 decode.s . . . 15

7 Conclusiones **23**

1 Introducción

Se propone escribir un programa cuya función principal se realice en C y desde allí se llame a subrutinas programadas en código MIPS32 que se encarguen de codificar o decodificar información de base64. Se cuentan con una serie de comandos básicos para el desarrollo del programa que serán detallados en el subtítulo de `Implementación`. El objetivo principal del programa consiste en realizar una acción que puede ser codificar o decodificar a partir de un archivo de entrada y generar un archivo de salida. En caso de no recibir los nombres de archivos se utiliza por defecto los streams standar **stdin** y **stdout**. Esto mismo ya fue realizado en el Trabajo Práctico 0 pero la novedad radica en implementar dichas funciones en MIPS32.

2 Documentación relevante al diseño e implementación del programa

A partir de la consigna se determina que los comandos que debe incluir el programa son:

`-h, --help Despliega el menú de ayuda`
`-V, --version Imprime la versión y cierra el programa`
`-i, --input Determina la ubicación del archivo de entrada`
`-o, --output Determina la ubicación del archivo de salida`
`-a, --action Determina la acción que ejecuta el programa: codificar (por defecto) o decodificar`

2.1 Diseño

El programa principal se encuentra desarrollado en la función `main`. Se detalla todo lo relativo al manejo de los comandos y se utiliza la libería `getopt.h`. Las acciones de codificar y decodificar se separan en subrutinas distintas y se implementan siguiendo una lógica similar. Ambas están en los archivos `code.S` y `decode.S` respectivamente.

Se puede encontrar el código del programa en el repositorio: `https://github.com/chortas/Orga-de-Compus-6620`[1]

2.2 Detalle de implementación

En esta sección se propone explicar la lógica que se adoptó para la implementación y enunciar los distintos problemas que se afrontaron y la solución elegida.

En un inicio se propuso la creación de subrutinas para modularizar el código: una para la lectura, otra para la escritura y otra para la búsqueda en la tabla base 64. Finalmente se descartó esta opción porque hacía más engorroso el desarrollo del código y hasta se podría considerar que complicaba algunas operaciones que de realizarse en el mismo archivo eran más autoexplicativas. De esta manera las subrutinas implementadas en MIP32 son las que se detallan en las siguientes secciones.

[1] Allí se encuentran tres carpetas: TP0, TP1 e Información. Todo lo detallado en este informe se encuentra en la carpeta TP1.

2.2.1 Subrutina encode

La lógica utilizada en esta subrutina se basa en leer x cantidad de bytes de un archivo específico o **stdin**, luego encodificarlos a caracteres de base 64 y escribirlos por el archivo de salida especificado o **stdout**. Debido a que la lógica ya se encontraba detallada en la función en C del tp0 se intentó seguirla y traducirla a las instrucciones de código MIPS. Sin embargo, no se pudo lograr una absoluta correspondencia debido a distintos problemas que surgieron a la hora de la implementación. En la función `encode` del tp0 se leía 1 byte, se encodificaba a base 64, se escribía el carácter encodificado y luego se leía el siguiente. Es decir:

- Lee un carácter
- Escribe un carácter
- Repite el ciclo si sigue habiendo carácteres para leer

Se intentó seguir esa lógica pero la lectura en un buffer de 1 byte resultó muy complicada de *debuggear*[2]. Algunos problemas a mencionar son la impresión de carácteres por parte del programa que no aparecían en GDB y viceversa (GDB imprimía carácteres que el programa no). Se intentó llegar a una solución para este problema pero ante la escasez de conocimiento sobre herramientas para debuggear código MIPS se eligió tomar un camino distinto al elegido en C.

Se procedió a leer de a 3 carácteres, debido a que es la unidad mínima que produce una cantidad de carácteres exactos sin padding (más específicamente 4). En consiguiente, se escribieron cuatro carácteres por ciclo. De esta manera, si se leían 3 carácteres se producían 4 carácteres en base 64. En caso de llegar a un EOF en alguna de estas tres lecturas se siguió con el procedimiento detallado para el agregado de paddings en el TP0. La lógica seguida en esta subrutina de MIPS varía ya que los pasos a seguir fueron:

- Leer caracteres de a 1 byte en código ASCII y colocarlo en un buffer de 3 posiciones
- Escribir 4 carácteres de a 1 byte en código B64 en un buffer de 4 posiciones
- Repetir el ciclo hasta encontrar un EOF

Tanto la lectura como la escritura de bytes se realizaron en distintos bloques reservados de memoria, un `buffer_entrada` y un `buffer_salida`. De esta manera se escribió en las posiciones adecuadas en el bloque de memoria reservado. Esto fue de gran utilidad debido a las firmas de las macros definidas en `<sys/syscall.h>` que se utilizaron para la lectura y la escritura requerían de punteros a datos. La firma de las funciones en C es: `ssize_t read(int fildes, void *buf, size_t nbyte);` (se leía de a 1 byte y se movía el puntero de lectura para guardar el resultado obtenido) y `ssize_t write(int fildes, const void *buf, size_t nbyte);` (se escribía de a 1 byte y se movía el puntero a escribir para realizar la escritura).

[2]Se utilizó el programa GDB para la realización de esta tarea. La misma es muy útil ya que permite ver el contenido de los registros y acceder a las distintas posiciones de memoria habilitadas por el programa que se corre

Es pertinente aclarar que no se detallan en este informe las distintas operaciones realizadas a los carácteres leídos para obtener los carácteres encodificados ya que se considera que eso excede los objetivos de este Trabajo Práctico. Esto mismo fue detallado en el informe del Trabajo Práctico 0 y no varió en la implementación de este.

2.2.2 Subrutina decode

En cuanto a esta subrutina los problemas surgidos fueron análogos a los de la función `encode`. Así, se eligió una solución similar a la detallada anteriormente: se implementó la lectura de 4 carácteres en base 64 y la decodificación a 3 carácteres en código ASCII.

- Leer 4 caracteres de a 1 byte en código B64 y colocarlos en un buffer de 4 posiciones
- Escribir 3 carácteres en código ASCII en un buffer de 3 posiciones
- Repetir el ciclo hasta encontrar un EOF

2.2.3 Función principal main

Se mantuvo la estructura elegida en el Trabajo Práctico 0 para la lectura de comandos. Igualmente, se tuvo que realizar alguna variación para la correspondencia con el código assembly MIPS. La misma consistió en obtener el file descriptor a partir del puntero a `FILE` a través de la función `int fileno(FILE *stream)` incluida en el header `stdio.h`. Además se tuvo que cambiar la firma de las funciones `encode` y `decode` del TP0 por las siguientes: `int encode(int fd, int wfd)` y `int decode(int fd, int wfd)` siendo `fd` el file descriptor del archivo de entrada para la lectura y `wfd` el file descriptor del archivo de salida para la escritura.

3 Comandos para compilar el programa

En esta sección se detallan los pasos para compilar el programa en NetBSD a partir del entorno proporcionado por GXemul.

- Desde el directorio donde se instaló GXemul se corre el siguiente comando para bootear la imagen del disco patrón: `hostOS# ./gxemul -e 3max -d netbsd-pmax.img`
- Desde otra consola de linux se crea en el host OS con el usuario root un alias para la interfaz loopback (lo:0) con la IP 172.20.0.1 con el siguiente comando: `hostOS# ifconfig lo:0 172.20.0.1`
- Luego se ejecutan los siguientes comandos para la conexión contra la interfaz creada:

  ```
  hostOS# export TERM=xterm
  hostOS# ssh -p 2222 root@127.0.0.1
  ```

- Se transfieren los archivos a compilar a NetBSD con los siguientes comandos:

  `scp -P2222 -r TP1 root@127.0.0.1:/root/TP0NetBSD`

- Luego se ejecutan el siguiente comando para realizar la compilación del programa:

  `root@: /TP1NetBSD/TP1# gcc -g -Wall main.c r.S w.S getb64index.S encode.S decode.S -o tp1`

4 Corridas de prueba

Las pruebas realizadas se basaron en los ejemplos del enunciado. Se probaron los comandos básicos como `-h` y `-V` para probar que muestren el resultado esperado. Luego, para probar los comandos `-a action -i input -o output` se realizó lo siguiente:

- Se probaron ejemplos básicos de codificación como el "Man" propuesto por el enunciado.
- Se ejecutó el siguiente comando en la terminal para verificar que archivos de tamaño creciente codificaban y decodificaban correctamente: `n=1;while :; do head -c n </dev/urandom >/tmp/in.bin; ./tp1 -a encode -i /tmp/in.bin -o /tmp/out.b64; ./tp1 -a decode -i /tmp/out.b64 -o /tmp/out.bin; if diff /tmp/in.bin /tmp/out.bin; then :; else echo ERROR: n; break; fi; echo ok:  n; n=((n+1)); rm -f /tmp/in.bin /tmp/out.b64 /tmp/out.bin; done`

5 Código principal en C

5.1 main.c

```
#include <stdio.h>        #include <string.h>
#include <stdlib.h>
#include <getopt.h>
#include <stdbool.h>
#include <unistd.h>
#include "base64.h"

//Definición del menú de ayuda
const char HELP[] = "Usage:\n tp0 -h \n tp0 -V \n tp0 [options] \n
Options: \n -V, --version Print version and quit. \n -h, --help
Print this information. \n -i, --input Location of the input file. \n -a, --action
 Program action: encode (default) or decode. \n";

//Definición de la versión del programa
const char VERSION[] = "2018.10.13 \n";

//Defino tabla b64
```

```
char B64[64]= {'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O',
'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z', 'a', 'b', 'c', 'd', 'e', 'f', 'g',
'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u','v', 'w', 'x', 'y',
'z', '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', '+', '/'};

int main (int argc, char const *argv[]) {

        static struct option long_options[] = {
        {"version",  no_argument, 0,  0 },
        {"help",  no_argument, 0,  0 },
        {"input",  optional_argument, 0,  0 },
        {"output", optional_argument, 0,  0 },
        {"action",  optional_argument, 0, 0},
        {0,  0,   0,  0 }
        };

        int opt;
        int fd = 0; //stdin
        int wfd = 1; //stdout
        char* const* buffer = (char* const*) argv;
        int option_index = 0;
        bool isencode = true;
        while ((opt = getopt_long(argc, buffer, "Vha:i:o:", long_options,
        &option_index)) != -1) {


                switch (opt) {

                        case 'h':
                                write(1, HELP, strlen(HELP));
                                return 0;

                        case 'V':
                                write(1, VERSION, strlen(VERSION));
                                return 0;

                        case 'a':
                                if (! strcmp(optarg, "encode")) {
                                        continue;
                                }
                                if (! strcmp(optarg, "decode")) {
                                        isencode = false;
                                        continue;
                                }

                        case 'i':
                                if (! strcmp(optarg,"-")) continue;
                                FILE* fp = fopen(optarg, "r");

                                if(! fd) {
```

```
                                        fprintf(stderr, "File not found \n");
                                        return 1;
                                }
                                fd = fileno(fp);

                        case 'o':
                                if (! strcmp(optarg,"-")) continue;
                                FILE* wfp = fopen(optarg, "w");

                                if(! wfd) {
                                        fprintf(stderr, "File not found \n");
                                        return 1;
                                }
                                wfd = fileno(wfp);
                        case 0:
                                abort();
                }
        }

        int output;

        if(isencode){
                output = encode(fd, wfd);
        }

        else{
                output = decode(fd, wfd);
        }

        if (output){
                fprintf(stderr, "%s", errmsg[output]);
        }

        close(fd);
        close(wfd);
        return 0;
}
```

5.2 base64.h

```
#ifndef BASE64_H
#define BASE64_H

int encode(int infd, int outfd);
int decode(int infd, int outfd);

char* errmsg[5]= {
"Todo OK",
"La lectura del archivo no fue exitosa",
"La escritura del archivo no fue exitosa",
```

```
"Cantidad de bytes inconsistente en el archivo a decodificar",
"Contenido inconsistente en el archivo a decodificar"
};

extern char B64[64];

#endif
```

6 Código implementado en MIPS32

6.1 code.s

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

#define FRAME_SZ 48

#define a1mask        0xFC
#define a2mask        0x03
#define b1mask        0xF0
#define b2mask        0x0F
#define c1mask 0xC0
#define c2mask 0X3F

.text
.abicalls
.align 2

.globl                encode
.ent                 encode

.set                 noreorder
.cpload                 t9
.set                 reorder

encode:
        sub         sp,sp, FRAME_SZ

        sw         $fp,(FRAME_SZ-16)(sp)
        sw         gp,(FRAME_SZ-12)(sp)
        sw         ra,(FRAME_SZ-8)(sp)
        move         $fp,sp

        sw t1,20(sp) #guardo t0 en el area de la pila de las variables locales


        sw a0, FRAME_SZ(sp) #Salvo el file descriptor de entrada en el arg building area d
        sw a1, (FRAME_SZ+4)(sp) #Salvo el file descriptor de salida en el arg building are
```

```
limpieza_buffer:
        la t0, buffer_entrada
        sb    zero, 0(t0)
        sb    zero, 1(t0)
        sb    zero, 2(t0)

        la t1, buffer_salida
        sb    zero, 0(t1)
        sb    zero, 1(t1)
        sb    zero, 2(t1)
        sb    zero, 3(t1)

lectura_inicial:
        la t0, buffer_entrada #aca podríamos llenar de cero para verificar que quede todo

        li t1, 0 #inicializo contador en 0
        sw t1, (20)(sp) #lo guardo en la pila

        lw        a0, (FRAME_SZ)(sp) #a0 -> file descriptor
        la        a1, buffer_entrada #a1 -> buffer de entrada
        li        a2, 1 #a2 -> leer 1 byte
        li        v0, SYS_read
        syscall

##Recupero t1 despues de la syscall
        lw t1, (20)(sp)

        bne a3, 0, error
        beq v0, 0, end #si leyo EOF que termine el programa

lectura_1:
        lw        a0, (FRAME_SZ)(sp) #a0 -> file descriptor
        la        a1, buffer_entrada + 1 #a1 -> buffer de entrada + 1
        li        a2, 1 #a2 -> leer 1 byte
        li        v0, SYS_read
        syscall

        ##Recupero t1 despues de la syscall
        lw t1, (20)(sp)

        beq v0, zero, primer_caracter #si termino de leer que vaya a ciclo

        bne a3, 0, error #agregar = FALTA

        addiu t1, t1, 1 #contador ++
        sw t1, (20)(sp) #lo guardo en la pila

lectura_2:
        lw        a0, (FRAME_SZ)(sp) #a0 -> file descriptor
        la        a1, buffer_entrada + 2 #a1 -> buffer de entrada + 1
```

```
        li        a2, 1 #a2 -> leer 1 byte
        li        v0, SYS_read
        syscall

        ##Recupero t1 despues de la syscall
        lw t1, (20)(sp)

        bne a3, 0, error #agregar = FALTA
        addiu t1, t1, 1 #contador ++
        sw t1, (20)(sp) #lo guardo en la pila

primer_caracter:
        la t0, buffer_entrada
        la t2, buffer_salida

        ## Codificación primera lectura

        lbu        t3, 0(t0)         # t3 = buffer_entrada[0] = buffer

        and t4, t3, a1mask # t4 = a1 = buffer & a1mask
        srl t4, t4, 2     # t4 = a1 = a1 >> 2

        and t5, t3, a2mask # t5 = a2 = buffer & a2mask
        sll t5, t5, 4 # t5 = a2 = a2 <<< 4

        sb t4, 0(t2)        # buffer_salida[0] = a1

        ## Pasar a B64

        la t0, B64        # t0 = direccion de la tabla

        lbu        t3, 0(t2)               # t3 = buffer_salida[0]
        addu t0, t0, t3
        lb        t0, 0(t0)               # t0 = B64[t3]
        sb        t0, 0(t2)               # buffer_salida[0] = t0

        beq t1, zero, escribir_doble_padding #si el contador esta en 1 es porque no tenía

segundo_caracter:

        la t0, buffer_entrada
        la t2, buffer_salida

        ## Codificación segunda lectura

        lbu        t3, 1(t0)         # t3 = buffer_entrada[1] = buffer

        and t4, t3, b1mask # t4 = b1 = buffer & b1mask
        srl t4, t4, 4     # t4 = b1 = b1 >> 4
        or t4, t4, t5 # t4 = b1 = b1 | a2
```

```
        and t5, t3, b2mask # t5 = b2 = buffer & b2mask
        sll t5, t5, 2 # t5 = b2 = b2 <<< 2

        sb t4, 1(t2)        # buffer_salida[1] = b1

        ## Pasar a b64

        la t0, B64        # t0 = direccion de la tabla

        lbu        t3, 1(t2)               # t3 = buffer_salida[1]
        addu t0, t0, t3
        lb        t0, 0(t0)               # t0 = B64[t3]
        sb        t0, 1(t2)               # buffer_salida[1] = t0

        beq v0, zero, escribir_simple_padding

tercer_cuarto_caracter:

        la t0, buffer_entrada
        la t2, buffer_salida

        ## Codificación tercera lectura

        lbu        t3, 2(t0)         # t3 = buffer_entrada[2] = buffer

        and t4, t3, c1mask # t4 = c1 = buffer & c1mask
        srl t4, t4, 6     # t4 = c1 = c1 >> 6
        or t4, t4, t5 # t4 = c1 = c1 | b2

        and t5, t3, c2mask # t5 = c2 = buffer & c2mask

        sb t4, 2(t2)        # buffer_salida[2] = c1
        sb t5, 3(t2)    # buffer_salida[3] = c2

        ## Pasar a b64

        la        t0, B64        # t0 = direccion de la tabla

        lbu        t3, 2(t2)               # t3 = buffer_salida[2]
        addu        t0, t0, t3
        lb        t0, 0(t0)               # t0 = B64[t3]
        sb        t0, 2(t2)               # buffer_salida[2] = t0

        la        t0, B64        # t0 = direccion de la tabla

        lbu        t3, 3(t2)               # t3 = buffer_salida[3]
        addu        t0, t0, t3
        lb        t0, 0(t0)               # t0 = B64[t3]
        sb        t0, 3(t2)               # buffer_salida[3] = t0
```

```
        b write_1

escribir_doble_padding:

        la t2, buffer_salida

        sb t5, 1(t2)        # buffer_salida[1] = a2

        ## Pasar a B64
        la t0, B64        # t0 = direccion de la tabla

        lbu        t3, 1(t2)                # t3 = buffer_salida[1]
        addu t0, t0, t3
        lb        t0, 0(t0)                # t0 = B64[t3]
        sb        t0, 1(t2)                # buffer_salida[1] = t0

        ## Escribir dos iguales
        li t0, 64

        li t0, '=' # Guardo = en t0
        sb t0, 2(t2) #buffer_salida[2] = t0
        sb t0, 3(t2) #buffer_salida[3] = t0

        b write_1

escribir_simple_padding:

        la t2, buffer_salida

        sb t5, 2(t2)        # buffer_salida[2] = b2

        ## Pasar a B64
        la t0, B64        # t0 = direccion de la tabla

        lbu        t3, 2(t2)                # t3 = buffer_salida[1]
        addu t0, t0, t3
        lb        t0, 0(t0)                # t0 = B64[t3]
        sb        t0, 2(t2)                # buffer_salida[2] = t0

        ## Escribir un igual
        li t0, '=' # Guardo = en t0
        sb t0, 3(t2) #buffer_salida[3] = t0

write_1:

        lw        a0, (FRAME_SZ + 4)(sp) #a0 -> wfd
        la        a1, buffer_salida #a1-> direccion del primer caracter

        li        a2, 1 #a2-> 1 byte
```

```
        li        v0, SYS_write
        syscall

        bne a3, 0, error

write_2:

        lw        a0, (FRAME_SZ + 4)(sp) #a0 -> wfd
        la        a1, buffer_salida + 1 #a1-> direccion del segundo caracter

        li        a2, 1 #a2-> 1 byte
        li        v0, SYS_write
        syscall

        bne a3, 0, error

write_3:
        lw        a0, (FRAME_SZ + 4)(sp) #a0 -> wfd
        la        a1, buffer_salida + 2 #a1-> direccion del tercer caracter

        li        a2, 1 #a2-> 1 byte
        li        v0, SYS_write
        syscall

        bne a3, 0, error

write_4:

        lw        a0, (FRAME_SZ + 4)(sp) #a0 -> wfd
        la        a1, buffer_salida + 3 #a1-> direccion del cuarto caracter

        li        a2, 1 #a2-> 1 byte
        li        v0, SYS_write
        syscall

        beq a3, 0, lectura_inicial

error:

        li a0, 1
        li v0, SYS_exit
        syscall

end:

        li v0, 0
        lw        a1,  (FRAME_SZ +  4)(sp)
        lw        a0,  (FRAME_SZ     )(sp)

        lw $fp,(FRAME_SZ-16)(sp)
```

```
        lw gp,(FRAME_SZ-12)(sp)
        lw ra,(FRAME_SZ-8)(sp) #duda

        addu sp,sp,FRAME_SZ #Libero el stackFrame

        jr ra

.end encode

.data
.align 2

buffer_entrada:
        .space 3
        .byte        0, 0, 0
buffer_salida:
        .space 4
        .byte        0, 0, 0, 0
```

6.2 decode.s

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

#define FRAME_SZ 32

#define mask1d        0x30
#define mask2d        0x3C
#define mask3d        0x3F

.data
.text
.abicalls
.align 2

.globl               decode
.ent               decode

.set               noreorder
.cpload               t9
.set               reorder

decode:

        subu         sp,  sp, FRAME_SZ

        sw      gp,  (FRAME_SZ - 24)(sp)
        sw             $fp, (FRAME_SZ - 20)(sp)
        sw             ra,  (FRAME_SZ - 16)(sp)
        sw      s2,  (FRAME_SZ - 12)(sp)
```

```
        sw      s3,  (FRAME_SZ - 8)(sp)

        move    $fp, sp

        sw              a0,  (FRAME_SZ     )(sp)
        sw              a1,  (FRAME_SZ +  4)(sp)

##LECTURAS

primer_lectura:

        lw a0, (FRAME_SZ)(sp)
        la a1, in_buffer
        li a2, 1
        li v0, SYS_read
        syscall

        beq a3, zero, primer_validacion_lectura

        b error_lectura

primer_validacion_lectura:

        li s2, 0 #La cuenta de los bytes leidos
        addu s2, s2, v0 #Suma 1 si leyo algun byte, 0 si no

        beq s2, 1, segunda_lectura #Si s2 es 0, leimos un EOF

        b salida_exitosa

segunda_lectura:

        lw a0, (FRAME_SZ)(sp)
        la a1, in_buffer + 1
        li a2, 1
        li v0, SYS_read
        syscall #Leo segundo byte

        beq a3, zero, segunda_validacion_lectura

        b error_lectura

segunda_validacion_lectura:

        addu s2, s2, v0 #Suma 1 si leyo algun byte, 0 si no

        beq s2, 2, tercera_lectura #Si s2 es 1, leimos un EOF

        b cantidad_de_bytes_inconsistente
```

```
tercera_lectura:

        lw a0, (FRAME_SZ)(sp)
        la a1, in_buffer + 2
        li a2, 1
        li v0, SYS_read
        syscall

        beq a3, zero, tercer_validacion_lectura

        b error_lectura

tercer_validacion_lectura:

        addu s2, s2, v0 #Suma 1 si leyo algun byte, 0 si no

        beq s2, 3, cuarta_lectura #Si s2 es 2, leimos un EOF

        b cantidad_de_bytes_inconsistente

cuarta_lectura:

        lw a0, (FRAME_SZ)(sp)
        la a1, in_buffer + 3
        li a2, 1
        li v0, SYS_read
        syscall

        beq a3, zero, cuarta_validacion_lectura

        b error_lectura

cuarta_validacion_lectura:

        addu s2, s2, v0 #Suma 1 si leyo algun byte, 0 si no

        beq s2, 4, inspeccion_buffer_leido #Si s2 es 3, leimos un EOF

        b cantidad_de_bytes_inconsistente

inspeccion_buffer_leido:

        la t0, in_buffer #t0 = direccion in_buffer

        ## Los iguales solo pueden estar en la tercer lectura
        ## O solo la cuarta lectura, luego devuelvo contenido inconsistente si no es asi

        lbu t1, 0(t0) # t1 = contenido de in_buffer[0]
        li t2, '=' #Cargo el caracter '=' para comparar despues
        beq t1, t2, contenido_inconsistente
```

```
        lbu t1, 1(t0) # t1 = contenido de in_buffer[1]
        li t2, '=' #Cargo el caracter '=' para comparar despues
        beq t1, t2, contenido_inconsistente

        lbu t1, 2(t0) # t1 = contenido de in_buffer[2]
        beq t1, t2, chequeo_doble_igual

        lbu t1, 3(t0) # t1 = contenido de in_buffer[3]
        beq t1, t2, validacion_fin_archivo_1 #El archivo deberia terminar aca

        li t3, 0
        j decoding

chequeo_doble_igual:

        lbu t1, 3(t0) # t1 = contenido de in_buffer[3]
        li t2, '='
        beq t1, t2, validacion_fin_archivo_2 #Sin errores (se leyó **==)

        b contenido_inconsistente

validacion_fin_archivo_1:

        lw a0, (FRAME_SZ)(sp)
        la a1, in_buffer + 3
        li a2, 1
        li v0, SYS_read
        syscall

        bne a3, zero, error_lectura

        beq v0,0 , contador_simple_igual #Suma 1 si leyo algun byte, 0 si no

        b contenido_inconsistente

contador_simple_igual:

        li s3, 1 # Caso ***=
        j decoding

validacion_fin_archivo_2:

        lw a0, (FRAME_SZ)(sp)
        la a1, in_buffer + 3
        li a2, 1
        li v0, SYS_read
        syscall

        bne a3, zero, error_lectura
```

```
        beq v0,0 , contador_doble_igual

        b contenido_inconsistente

contador_doble_igual:

        li s3, 2 # Caso **==
        j decoding

##RUTINA QUE BUSCA EL NUMERO DE INDICE DE B64 PARA EL CARACTER A DECODIFICAR #Le llega por

geti64index:

        li t0,0 #cargo un 0 en t0 para usarlo de contador

        la t1,B64 #esto va a fallar, t1 -> direccion de inicio del array

        addu t2, t0, t1 #Posicion absoluta del indice

ciclob64index:

        beq t0, 64 , contenido_inconsistente

        addu t2, t0, t1
        lb t3, 0(t2) #accedo a elemento i del array y lo guardo en t3

        beq t3,t4, indiceb64encontrado

        addi t0,t0,1

        b ciclob64index

indiceb64encontrado:

        j ra

        ##DECODE

decoding:

        b primer_decode

primer_decode:

        lb t4, in_buffer
        la ra, segundo_decode

        b geti64index
```

```
segundo_decode:

        sb t0, partial_decode_buffer
        lb t4, in_buffer+1
        la ra, tercer_decode

        b geti64index

tercer_decode:

        sb t0, partial_decode_buffer+1
        lb t4, in_buffer+2
        li t5, '=' #Tengo que ver si mi tercer leida no fue un =
        beq t4, t5, rearmado_de_bytes

        la ra, cuarto_decode

        b geti64index

cuarto_decode:

        sb t0, partial_decode_buffer+2
        lb t4, in_buffer+3
        li t5, '='
        beq t4, t5, rearmado_de_bytes

        la ra, finalizo_decode

        b geti64index

finalizo_decode:

        sb t0, partial_decode_buffer+3

rearmado_de_bytes:

        lb t0, partial_decode_buffer
        lb t1, partial_decode_buffer+1
        lb t2, partial_decode_buffer+2
        lb t3, partial_decode_buffer+3
        la t4, out_buffer

        sll t0, 2 #t0 << 2
        li t5, 0 #Limpio t5 por las dudas
        and t5, t1, mask1d #En t5 tengo los 2 LSB que faltan para out_buffer[0]
        srl t5, 4 #Muevo esos dos bits a las 2 LSB positions
        or t0, t0, t5
        sb t0, 0(t4) # output[0] = t0

        beq s3, 2, primer_escritura #Caso **== --> x
```

```
        sll t1, 4 #Los 4 bits que sobraron de t1 << 4
        li t5, 0 #Limpio t5 por las dudas
        and t5, t2 , mask2d #En t5 tengo los 4 LSB que faltan para out_buffer[1]
        srl t5, 2 #Muevo esos dos bits a las 4 LSB positions
        or t1, t1, t5
        sb t1, 1(t4) # output[1] = t1

        beq s3, 1, primer_escritura #Caso ***= --> xx

        sll t2, 6 #Los 2 bits que sobraron de t2 << 6
        or t2, t2, t3
        sb t2, 2(t4) # output[2] = t2

##ESCRITURA

primer_escritura:

        lw        a0, (FRAME_SZ + 4)(sp)
        la         a1, out_buffer
        li         a2, 1
        li         v0, SYS_write
        syscall

        beq a3, 0, primer_validacion_escritura

        b error_escritura

primer_validacion_escritura:

        ##Si lei dos iguales, solo deberia escribir un byte
        ##En este ciclo, luego, habria que salir sin errores

        li t0, 2
        beq t0, s3, salida_exitosa

segunda_escritura:

        lw        a0, (FRAME_SZ + 4)(sp)
        la         a1, out_buffer + 1
        li         a2, 1
        li         v0, SYS_write
        syscall

        beq a3, 0, segunda_validacion_escritura

        b error_escritura

segunda_validacion_escritura:
```

```
        ## Si lei un solo igual, solo deberia escribir dos bytes
        ## En este ciclo, luego, habria que salir sin errores

        li t0, 1
        beq t0, s3, salida_exitosa

tercer_escritura:

        lw        a0, (FRAME_SZ + 4)(sp)
        la         a1, out_buffer + 2
        li         a2, 1
        li         v0, SYS_write
        syscall

        beq a3, 0, continuo_ciclo

        b error_escritura

continuo_ciclo:

        j primer_lectura

##RETORNOS

salida_exitosa:

        li v0, 0
        j end

error_lectura:

        li v0, 1
        j end

error_escritura:

        li v0, 2
        j end

cantidad_de_bytes_inconsistente:

        li v0, 3
        j end

contenido_inconsistente:

        li v0, 4 # Error de contenido inconsistente
        j end

end:
```

```
        lw  a1,  (FRAME_SZ +  4)(sp)
        lw          a0,  (FRAME_SZ           )(sp)

        lw          gp,  (FRAME_SZ - 24)(sp)
        lw          $fp, (FRAME_SZ - 20)(sp)
        lw          ra,  (FRAME_SZ - 16)(sp)
        lw  s3,  (FRAME_SZ - 12)(sp)
        lw  s2,  (FRAME_SZ -  8)(sp)

        addu          sp,  sp, FRAME_SZ

        jr          ra
.end decode

.data
.align 2

partial_decode_buffer:
        .space  4
        .byte         0, 0, 0, 0
in_buffer:
        .space  4
        .byte         0, 0, 0, 0
out_buffer:
        .space  3
        .byte         0, 0, 0
```

7 Conclusiones

En conclusión, se cumplió el objetivo del Trabajo Práctico ya que se desarrolló el programa detallado por el enunciado. Los comandos del programa se ejecutan como detalla el comando `-h` y tras ser sometidos a distintas pruebas se concluye que funcionan según lo esperado. Así mismo fue posible la implementación del código en MIPS32 y se pudo utilizar el programa GXemul para simular un entorno de desarrollo de una maquina MIPS corriendo el sistema operativo NetBSD para ejecutarlo. Los distintos problemas que surgieron pudieron solucionarse pero muchas veces se generó una complicación muy grande a la hora de realizar un seguimiento del contenido de los registros. No se pudo obtener una solución a la lectura de un buffer de 1 byte pero no se consideró que eso genere una solución más ineficaz debido a que igualmente el programa se desarrolló según los parámetros esperados.

References

[1] GXemul, http://gavare.se/gxemul/.

[2] J. L. Hennessy and D. A. Patterson, “Computer Architecture. A Quantitative Approach,” 3ra Edición, Morgan Kaufmann Publishers, 2000.

[3] J. Larus and T. Ball, "Rewriting Executable Files to Mesure Program Behavior," Tech. Report 1083, Univ. of Wisconsin, 1992.

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

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

[6] Base64 Enconde and Decode - Online, https://www.base64encode.org/

[7] Getopt Long Option Example (The GNU C library), https://www.gnu.org/

[8] MIPS RISC Processor, http://www.sco.com/developers/devspecs/mipsabi.pdf