

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

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 librería **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 MIPS32 son las que se detallan en las siguientes secciones.

¹Allí 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 caracteres 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 caracteres por parte del programa que no aparecían en GDB y viceversa (GDB imprimía caracteres 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 caracteres, debido a que es la unidad mínima que produce una cantidad de caracteres exactos sin padding (más específicamente 4). En consiguiente, se escribieron cuatro caracteres por ciclo. De esta manera, si se leían 3 caracteres se producían 4 caracteres 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 caracteres 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).

²Se 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 caracteres leídos para obtener los caracteres 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 caracteres en base 64 y la decodificación a 3 caracteres en código ASCII.

- Leer 4 caracteres de a 1 byte en código B64 y colocarlos en un buffer de 4 posiciones
- Escribir 3 caracteres 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; ./main -a encode -i  
/tmp/in.bin -o /tmp/out.b64; ./main -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, --versi

//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',
'W', 'X', 'Y', 'Z', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n',
'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");
                fd = fileno(fp);

                if(! fd) { fprintf(stderr, "File not found \n"); return 1;
                    continue;
                }

            case 'o':

```

```

        if (! strcmp(optarg, "-")) continue;
        FILE* wfp = fopen(optarg, "w");
        wfd = fileno(wfp);
        if(! wfd) { fprintf(stderr, "File not found \n"); return 1;
        break;

        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

6.2 decode.s

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 Measure 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 Encode 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>