Trabajo Práctico Nº 1: Conjunto de instrucciones MIPS

Martinez Ariel, *Padrón Nro. 88573*
arielcorreofiuba@gmail.com.ar

Nestor Huallpa, *Padrón Nro. 88614*
huallpa.nestor@gmail.com

Pablo Sivori, *Padrón Nro. 84026]*
sivori.daniel@gmail.com

Entrega: 16/05/2017

1er. Cuatrimestre de 2017
66.20 Organización de Computadoras
Facultad de Ingeniería, Universidad de Buenos Aires

Resumen

En el presente trabajo práctico se describirán todos los pasos y conclusiones relacionadas al desarrollo e implementación de la codificacion y decodificacion de datos formateados en base 64.

Índice

1. Introducción

El objetivo del presente trabajo práctico es implementar las funciones encode y decode de datos base 64 en código assembly. Para ello nos conectamos con el emulador gxemul para realizar la codificación en mips 32 y posteriormente poder realizar las pruebas pertinentes.

2. Implementación

2.1. Lenguaje

Como lenguaje de implementación se eligió ANSI C ya que el mismo permite una alta portabilidad entre diferentes plataformas. El desarrollo del programa se realizó usando un editor de texto (gedit,vim, kwrite) y compilando los archivos fuente con GCC que viene en linux. Para compilar, ejecutar el siguiente comando:

$ make

2.2. Descripción del programa

La función `main` se encuentra en `tp0.c` y se encarga de interpretar las opciones y argumentos. En caso de ser una opción, como ayuda o versión, se imprime el mensaje correspondiente y finaliza la ejecución. Cuando no es una opción de ayuda o versión, se procede a procesar los datos de entrada. La salida de estas funciones proveen un codigo de error que sirve como salida del programa. Los mensajes de versión y ayuda se imprimen por stdout y el programa finaliza devolviendo 0 (cero) al sistema. Los mensajes de error se imprimen por la salida de errores (stderr) y el programa finaliza devolviendo 1 (uno) al sistema.

2.2.1. Errores posibles

1. El procesamiento de la entrada estándar causó el agotamiento del heap.
2. La invocación del programa es incorrecta.
3. Alguno de los archivos es inexistente.
4. Se produjo un error en la lectura del archivo a decodificar.
5. Se produjo un error en la escritura del archivo, donde se encuentra el resultado de la decodificación.

Cuando se produce un error en la codificación o decodificación, se devuelve un código distinto de 0 el cual sirve como indice para ver la descripción del error, la cual se encuentra en el vector de errores msgerr. En caso de que se devuelva 0, en esta posición el vector msgerr contendrá el mensaje "No hubo errores".

2.3. Desarrollo de actividades

1. Se instaló en un linux un repositorio de fuentes (GIT) para que al dividir las tareas del TP se pudiese hacer una unión de los cambios ingresados por cada uno de los integrantes más fácilmente.

2. Cada persona del grupo se comprometió a que sus cambios en el código fuente y los cambios obtenidos del repositorio que pudiesen haber subido los otros integrantes del grupo, sean compilados los sistemas operativos Linux y el NetBSD, asegurando así portabilidad entre plataformas planteada en el enunciado.

3. Se estableció que todos los integrantes en mayor o menor medida, contribuyan en el desarrollo de todas las partes del código para que nadie quede en desconocimiento de lo que se hizo en cada sección. Una parte se dedico al desarrollo de la función encode, mientras que la otra parte se focalizó en el desarrollo de la función decode. De esta manera los integrantes del grupo realizaron la programación en codigo assembly de ambas funciones familiarizandose de esta manera con el lenguaje.

4. Se propuso antes de iniciar el desarrollo de cada función, diagramar los stacks frames de las funciones que utilizan el encode y decode, respetando la convención de la abi dada por la catedra a través de la bibliografía brindada por la misma.

5. Solo se desarrollo el código assembly de las funciones encode y decode, lo cual para poder generar el resto del codigo assembly a partir del código fuente, dentro de NETBSD, se utilizó gcc con la siguiente opción:

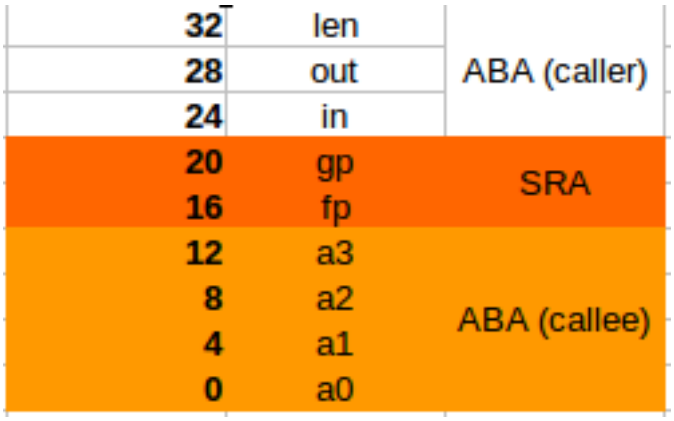
   gcc -S main.c

6. Para crear el presente informe se debe utilizar el comando make en el directorio informe.

2.4. Stack Frame de funciones

Función Encode

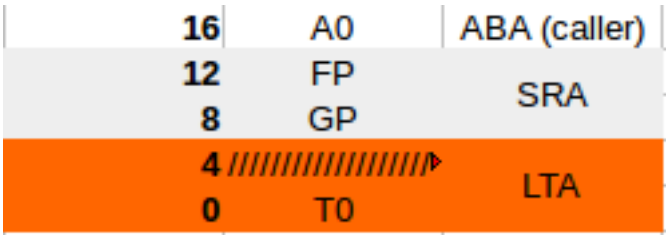
Para la función encode, se utilizo la función bloqueToBase64 cuyo stack frame es el siguiente:



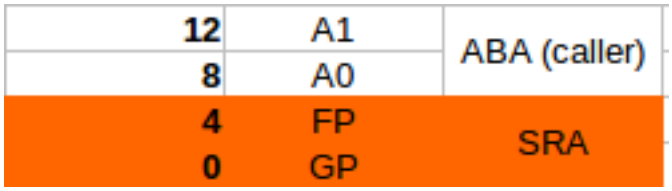
La función principal para realizar la codificación del mensaje a base 64 tiene el siguiente stack frame:

Función Decode

Para la función decode, se utilizo la función int search_b64(char caracter), la cual devuelve la posición en el vector de caracteres base64, del caracter codificado en base 64, utilizado para el proceso de decodificación. El stack frame utilizado para dicha función es:



También se utilizó la función read characters, la cual lee los caracteres de un archivo con codificación base 64. Su stack frame es:



Finalmente se muestra el stack frame de la función decode:

76	A0	ABA (caller)
72	A1	
68	////////////////////	SRA
64	RA	
60	FP	
56	GP	
52	BUFFER_WRITE	LTA
48		
44		
40		
36	T4	
32	T3	
26	T2	
24	T1	
20	T0	
12	A3	ABA (callee)
8	A2	
4	A1	
0	A0	

2.5. Casos de prueba

1. Codificamos **Man** ingresando por stdin y devolviendo por stdout
   TWFuCg==
   Para esta prueba ejecutamos por consola, estando posicionados en la carpeta pruebas, el siguiente comando:
   echo Man — ../tp0 -a encode

2. Codificamos **Man** ingresando por stdin. Luego decodificamos por stdout obteniendo como resultado **Man**.
   Para esta prueba ejecutamos por consola, estando posicionados en la carpeta pruebas, el siguiente comando:
   echo -n Man — ../tp0 — ../tp0 -a decode

3. Verificamos bit a bit, obteniendo por stdout
   0000000 x y z \n
   0000004
   Para esta prueba ejecutamos por consola, estando posicionados en la carpeta pruebas, el siguiente comando:
   echo xyz — ../tp0 — ../tp0 -a decode — od -t c

4. Codificamos 1024 bytes y chequeamos que no haya más de 76 unidades de longitud.
   El resultado es una secuencia de palabras eQp5CnkK que se repiten 9 veces por linea, teniendo un total de 17 lineas.
   En la última línea, la secuencia finaliza con las palabras eQp5Cg==
   Para esta prueba ejecutamos por consola, estando posicionados en la carpeta pruebas, el siguiente comando:
   yes — head -c 1024 — ../tp0 -a encode

5. Verificamos que la cantidad de bytes decodificados sea 1024.
   1024
   Para esta prueba ejecutamos por consola, estando posicionados en la carpeta pruebas, el siguiente comando:
   yes — head -c 1024 — ../tp0 -a encode — ../tp0 -a decode — wc -c

6. Codificamos el contenido del archivo de **entrada.txt**, el cual contiene la palabra **Man**. Guardamos la salida en el archivo **salida.txt**, el cual contiene como resultado de la codificación **TWFuCg==**. Luego decodificamos la salida de este archivo y lo mandamos a otro archivo de entrada (**entrada2.txt**). Vemos que en este último archivo se encuentra la palabra que se había pasado como entrada en el primer archivo, verificandose de esta manera la codificación y decodificación.
   Para esta prueba ejecutamos por consola, estando posicionados en la carpeta pruebas, los siguientes comandos:
   ../tp0 -i entrada.txt -o salida.txt -a encode
   ../tp0 -i salida.txt -o entrada2.txt -a decode

7. Corremos el script **test_enunciado.sh**. El programa verifica con diff la codificación, mostrando las diferencias en caso de error. Finalmente esperamos superar la centena de corridas para cortar su ejecución haciendo ctrl+c.

3. El código fuente, en lenguaje C

```
/**
*    6620 - Organizacion del computador
*    Trabajo Practico 0
*    Alumnos:
*            88614 - Nestor Huallpa
*            88573 - Ariel Martinez
*            84026 - Pablo Sivori
*/

#include <getopt.h>
#include "base64.h"
#include <unistd.h>

static char* ENCODE = "encode";
static char* DECODE = "decode";

typedef struct {
        char* accion;
        char* entrada;
        char* salida;
} Parametro;

Parametro manejarArgumentosEntrada(int argc, char** argv)
{
        int siguiente_opcion;
        int option_index;

    /* Una cadena que lista las opciones cortas validas */
    const char* const op_cortas = "hva:i:o:"; /* "hva::i:o:" */

    /* Una estructura de varios arrays describiendo los valores largos */
    const struct option op_largas[] =
        {
        { "help",       no_argument,            0,  'h'},
        { "version",    no_argument,            0,  'V'},
                { "action",     required_argument,  0,  'a'}, /*optional_argument*/
                { "input",      required_argument,  0,  'i'},
                { "output",     required_argument,  0,  'o'},
        { 0, 0, 0, 0 }
    };


        Parametro parametro;
        parametro.accion        = ENCODE;
        parametro.entrada       = "";
        parametro.salida        = "";

    while (1)
    {
        siguiente_opcion = getopt_long (argc, argv, op_cortas, op_largas, &option_index);

        if (siguiente_opcion == -1)
                break;

        switch (siguiente_opcion)
        {
            case 'h' :

                printf("Usage:\n");
                printf("\ttp0 -h\n");
                printf("\ttp0 -V\n");
                printf("\ttp0 [ options ]\n");

                printf("Options:\n");
                printf("\t-V, --version       Print version and quit.\n");
                printf("\t-h, --help          Print this information.\n");
                printf("\t-i, --input         Location of the input file.\n");
                printf("\t-o, --output        Location of the output file.\n");
                printf("\t-a, --action        Program action: encode (default) or decode.\n");

                printf("Examples:\n");
                printf("\ttp0 -a encode -i ~/input -o ~/output\n");
                printf("\ttp0 -a encode\n");
                exit(0);
                    break;

            case 'v' :
                printf("Tp0: Version 0.1: Grupo: Nestor Huallpa, Ariel Martinez, Pablo Sivori \n");
                exit(0);
                break;

            case 'a' :
                if ( optarg )
                        parametro.accion = optarg;
                break;

            case 'i' :
                if ( optarg )
                                        parametro.entrada = optarg;
                break;

            case 'o' :
                if ( optarg )
                                        parametro.salida = optarg;
                break;
```

```
            }
        }

        return parametro;

    }

    int main(int argc, char** argv) {
            Parametro p = manejarArgumentosEntrada(argc, argv);

            int isEntradaArchivo = strcmp(p.entrada,"");
            int isSalidaArchivo = strcmp(p.salida,"");
            FILE* archivoEntrada = (isEntradaArchivo!=0)?fopen(p.entrada, "rb"):stdin; //Si la entrada esta vacia lee st
            FILE* archivoSalida = (isSalidaArchivo!=0)?fopen ( p.salida, "w" ):stdout; //Si la salida esta vacia escribe

            int returnCode = 0;

            if (archivoEntrada == NULL) {
                    fprintf(stderr, "ERROR: NO EXISTE LA ENTRADA.\n");
                    exit (1);
            }

            if ( strcmp(p.accion, ENCODE) == 0 )
            {
                    /* Codificar entrada */
            int fileDescriptorEntrada = fileno(archivoEntrada);
            int fileDescriptorSalida = fileno(archivoSalida);
                    returnCode = base64_encode(fileDescriptorEntrada, fileDescriptorSalida);

            } else if ( strcmp(p.accion, DECODE) == 0 ) {

                    /* Decodificar entrada */
            int infd = fileno(archivoEntrada);
            int outfd = fileno(archivoSalida);
                    returnCode = decodificar(infd, outfd);

            } else {
                    fprintf(stderr, "ERROR: SE DEBE INGRESAR UN ARGUMENTO CORRECTO PARA LA OPCION i.\n");
                    exit(1);
            }

        /* Cierro los archivos de entrada y salida si no son stdin y stdout */
        if(isEntradaArchivo!=0){
                    fclose(archivoEntrada);
            }
            if(isSalidaArchivo!=0){
                    fclose  ( archivoSalida );
            }

            if(returnCode!=0){
                    exit(1);
            }

            return returnCode;
    }
```

4. El código MIPS32 generado por el compilado

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

#########FUNCION DECODE############

#STATICS VAR DEFINITIONS FUNCTION DECODIFICAR
#define SF_SIZE_DECODIFICAR         60
#define SF_DECODIFICAR_GP_POS       44
#define SF_DECODIFICAR_FP_POS       48
#define SF_DECODIFICAR_RA_POS       52


#STATICS VAR DEFINITIONS FUNCTION READ CHARACTERS
#define SF_SIZE_READCH               8
#define SF_READCH_GP_POS             0
#define SF_READCH_FP_POS             4

#define BUFFER_SIZE                  4

.data
.align 2
#index:  .byte 1
buffer: .space 4
buffer_write: .space 4

.text
.abicalls
.align 2
.global decodificar
.ent decodificar

decodificar:
        .frame          $fp, SF_SIZE_DECODIFICAR, ra
        .set            noreorder
        .cpload         t9
        .set            reorder
        subu            sp, sp, SF_SIZE_DECODIFICAR
        sw              ra, SF_DECODIFICAR_RA_POS(sp)
        #sw             gp, SF_DECODIFICAR_GP_POS(sp)
        sw              $fp, SF_DECODIFICAR_FP_POS(sp)
        .cprestore      SF_DECODIFICAR_GP_POS
        move            $fp, sp

        sw              a0, 60($fp)
        sw              a1, 64($fp)
```

5. Conclusiones

1. La realización completa del TP llevó cierta dificultad al tener que realizarlo en el contexto solicitado: alta portabilidad, desarrollo en C, funciones en assembly respetando la convención de la ABI e informe hecho en LaTeX.

2. En el primer caso la dificultad radicaba en tener configurado y funcionando el GXEmul dentro de un Linux, y lograr que en ambos casos el programa compile y corra sin problemas.

3. Tuvimos que invertir tiempo para leer la bibliografía dada por la catedra para respetar la convención de la ABI para el desarrollo en assembly de las funciones solicitadas.

4. En cuanto al trabajo grupal en si mismo, no hubo inconvenientes de ningún tipo ya que al ser el grupo relativamente chico y tener conocimiento del manejo del versionado de un proyecto ante cambios ingresado por los integrantes (por medio del GIT), la introducción de modificaciones y correcciones fué fluida.

6. Enunciado del trabajo practico

Universidad de Buenos Aires, F.I.U.B.A.
66.20 Organización de Computadoras
Trabajo práctico 1: conjunto de instrucciones MIPS
`$Date: 2017/04/23 22:25:51 $`

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

- Entrega: 2/5/2017;
- Vencimiento: 16/5/2017.

Referencias

[1] Enunciado del primer trabajo práctico (TP0), primer cuatrimestre de 2017 (`http://groups.yahoo.com/groups/orga-comp/files/TPs/`).

[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/`).