

Trabajo Práctico 1:

Conjunto de instrucciones MIPS

Jimenez, Ruben, *Padrón Nro. 92.402*
rbnm.jimenez@gmail.com

Reyero, Felix, *Padrón Nro. 92.979*
felixcarp@gmail.com

Suárez, Emiliano, *Padrón Nro. 78.372*
emilianosuarez@gmail.com

Primera Entrega: 30/04/2015

1er. Cuatrimestre de 2015

66.20 Organización de Computadoras – Práctica Jueves
Facultad de Ingeniería, Universidad de Buenos Aires

Resumen

Se implementó una versión simplificada del programa **sha1** de UNIX.
Para nuestra implementación.

1. Introducción

Este Trabajo Práctico pretende familiarizarse con la programación en assembly y el concepto de ABI.

Para ello, implementaremos el algoritmo **sha1** de UNIX en código *Assembly*, mientras que la interpretación de argumentos del programa y la lectura de archivos será realizada en lenguaje *C*.

Además, se utilizará *GXemul* para simular una máquina *MIPS* corriendo una versión reciente del sistema operativo *NetBSD*.

El programa implementado, muestra por *stdout* un checksum generado a partir del contenido de los archivos pasados por parámetro. En caso de no especificarse algún archivo, se mostrarán un checksum a partir de lo ingresado por *stdin*.

2. Implementación

2.1. Arquitectura

En una primera versión se trabajó con una función *sha* íntegramente desarrollada en *Assembly*, con un stack mas grande. Pero tuvimos problemas con el *malloc* al intentar reservar memoria para los bloques a procesar por el algoritmo. Como una alternativa, probamos utilizar la función *malloc* de *C* desde *Assembly*, realizando *syscall* e incluso, utilizando la versión en *Assembly* de *mymalloc.S* que se encuentra en el grupo.

Con ninguna de estas opciones lograr que la función *sha* funcionara correctamente, por lo que decidimos modularizarla de la siguiente manera.

main.c

Cuerpo principal del programa, donde se realiza la lectura de los parámetros de los archivos.

Cada uno de estos archivos, es guardado en memoria dinámica con un tamaño máximo de 1000 bytes (utilizando *malloc*). En caso que el tamaño del archivo supere dicho tamaño, se reasigna memoria dinámica mediante la utilización de *realloc*, para disponer un bloque de memoria donde quepa el archivo (bloque de igual tamaño al tamaño del archivo).

relleno.S

Donde se encuentra la implementación en *Assembly* de la función *calcularRelleno*.

Dicha función cuenta con un stack donde se almacena la longitud original así como la longitudRelleno que es la longitud que incluye el relleno para poder determinar la cantidad de bloques finales a procesar.

También se almacena en el stack la variable *cantBloques* que determina la cantidad de bloques a procesar.

La dificultad de este método fue la de tratar longitudes que ocupan 64 bits, para ello se tuvo que manejar dos registros uno que tenga los 32 bits mas significativos y otro para los 32 restantes, usando el carry que produce la suma de la parte menos significativa para actualizar la parte mas significativa de la longitud final.

trozo.S

Donde se encuentra la implementación en *Assembly* de la función *cargarTrozos*.

Dicha función, además de los registros obligatorios, contiene en su stack dos “saved registers” *s0* y *s1* que almacenan los punteros hacia los vectores “trozos” y “bloques” y los registros para almacenar las variables “*i*” y “*mask*”. En el cuerpo de la función se hace uso de los 8 registros temporales para distintas asignaciones necesarias.

Luego de guardar los parámetros con los que viene la función, inicializar “*i*” y cargar la “*mask*” a un temporal, genera las 16 primeras palabras del vector de 80, trayendo byte a byte desde “bloques” y generando con operaciones lógicas (and, or) y movimientos (*sll*) el word final que se guarda en “trozos”.

Luego desde la posicion 16 hasta la 80 con operaciones lógicas y rotaciones (xor y rol) se genera el resto del contenido del vector operando con los 16 words anteriores.

algoritmo.S

Donde se encuentra la implementación en *Assembly* de la función *algoritmoSha1*.

Dicha función, además de los registros obligatorios, contiene en su stack seis “saved registers” s2..s7 que almacenan los punteros hacia los valores de “a,b,c,d,e,f” y los registros para almacenar las variables “i”, “k” y “temp”. En el cuerpo de la función se hace uso de los 8 registros temporales para distintas asignaciones necesarias.

Luego de guardar los parámetros con los que viene la función e inicializar “i” , se generan 4 ramas distintas de condicionales con procesos distintos (dependiendo el valor de “i” en el for) donde a través de operaciones lógicas (and, not, xor, or) generan un valor que se almacena en “f” y se carga una constante “k” determinada para que luego se terminen de generar los valores que devuelve la función y se asignen a sus respectivas variables.

Las funciones *calcularRelleno*, *cargarTrozos* y *algoritmoSha1*, son llamadas desde la función *sha1* de *main.c*.

Los stacks de cada una de ellas que pueden observarse a continuación:

calcularRelleno	
Dir Mem	Valor
48	
44	ra
40	fp
36	gp
32	
28	longOrig
24	longOrig
20	longRelleno
16	longRelleno
12	a3
8	a2
4	a1
0	a0

cargarTrozos	
Dir Mem	Valor
36	
32	ra
28	fp
24	gp
20	mascara
16	i
12	a3
8	a2
4	a1
0	a0

algoritmoSha1	
Dir Mem	Valor
44	
40	ra
36	fp
32	gp
28	
24	temp
20	k
16	i
12	a3
8	a2
4	a1
0	a0

Registro	Valor
s0	bloques
s1	trozos
s2	a
s3	b
s4	c
s5	d
s6	e
s7	f

2.2. Diseño

Se desarrollo un programa que realiza la lectura a través del stdin o a través de archivos que se reciben por parámetro.

El comando acepta 2 parámetros para mostrar la Ayuda y la Versión del programa:

```
$ ./sha -h
$ ./sha --help
```

Para desplegar la ayuda del comando. Y los siguientes comandos para mostrar la versión:

```
$ ./sha -V
$ ./sha --version
```

Inicialmente el programa revisa la cadena de parametros ingresada y determina si el checksum debe generarse a partir de lo ingresado por *stdin* o a través del contenido del (o los) archivo(s).

Para esta última opción, se procesan los archivos de uno por vez, y para cada uno de ellos se genera el checksum a partir de sus datos.

Primero se obtiene el largo total del archivo, luego se llama a la función `calcularRelleno`, donde se obtiene una `longitudRelleno` que dividida por 8 determina la longitud del vector que se asigna en memoria dinámica para trabajar con el archivo completo.

Para completar el archivo, se llama a la función de C “`asignarDatos`” la cual se encarga de completar el vector anteriormente pedido con las características correspondientes que pide el preprocesamiento del algoritmo sha1 (a saber: el relleno consiste en un uno seguido de los ceros que sean necesarios, aunque el mensaje ya tenga la longitud deseada se debe efectuar el relleno, por lo que el número de bits de dicho relleno está en el rango de 1 a 512 bits, luego se le añade un bloque de 64 bits que represente la longitud del mensaje original antes de ser rellenado).

Es decir, la totalidad del archivo mas el relleno que lo completa, queda alojado en memoria dinámica. A continuación se procede con la función `sha1`.

Una vez dentro de la función `sha1`, se llama a la función “`calcularRelleno`”, donde se obtiene una `longitudRelleno` que, dividida por 512, determina la cantidad de bloques finales a procesar. Se inicializan las variables y por una decisión de diseño se decidió trabajar con un vector de 80 palabras en memoria dinámica porque nos parecía que el stack quedaba muy engorroso para trabajarlo.

Por esta elección, tuvimos problemas con la asignación de memoria dinámica desde assembly por lo cual se dividió (como se menciona en este informe) en partes la función.

A continuación se procede a trabajar con el bloque de datos en un `while` donde primero se carga el vector de 80 en “`cargarTrazos`” y luego el algoritmo propiamente dicho en “`algoritmosha1`” para finalmente devolver el valor en 160 bits.

2.3. Compilación

Se creó un archivo *Makefile* que permite compilar tanto la versión en *MIPS*, como en la versión implementada completamente en *C*.

El contenido de este archivo, puede verse a continuación:

```
1 CC=gcc -g -O0
2 CFLAGS=-Wall -lm
3 LDFLAGS=
4 OBJ_DIR=/
5 SOURCES=main.c
6 MIPS_FILES=algoritmo.S relleno.S trozo.S
7
8 OBJECTS=$(SOURCES:.c=.o)
9 EXECUTABLE=sha
10
11 mips:
12 $(CC) $(CFLAGS) $(SOURCES) $(MIPS_FILES) -o $(EXECUTABLE)
```

```

13 |
14 | c:
15 | $(CC) $(CFLAGS) $(SOURCES) -o $(EXECUTABLE)
16 |
17 | clean:
18 | rm -rf $(EXECUTABLE) $(OBJECTS)

```

../src/Makefile

Para compilarlo, se debe abrir una terminal en la carpeta donde están alojados los archivos fuentes (*src/*) y se ejecuta el siguiente comando:

```
../src$ make mips
```

Para compilar la versión en *Assembly* de *MIPS*.
O se puede utilizar esta opción:

```
../src$ make c
```

Para generar la versión en *C*.
Ambas opciones, generan el ejecutable *sha*.

3. Casos de Prueba

Algunos de los casos de pruebas realizados, pueden observarse a continuación:

```

root@:/home/gxemul/tprub/div# sha1 hola
SHA1 (hola) = ef443fee4da6bfb41651930de7ad99f29ed9f079
root@:/home/gxemul/tprub/div# ./sha hola
ef443fee4da6bfb41651930de7ad99f29ed9f079

root@:/home/gxemul/tprub/div# sha1 vacio
SHA1 (vacio) = da39a3ee5e6b4b0d3255bfef95601890afd80709
root@:/home/gxemul/tprub/div# ./sha vacio
da39a3ee5e6b4b0d3255bfef95601890afd80709

root@:/home/gxemul/tprub/div# sha1 prueba-dog
SHA1 (prueba-dog) = 2fd4e1c67a2d28fced849ee1bb76e7391b93eb12
root@:/home/gxemul/tprub/div# ./sha prueba-dog
2fd4e1c67a2d28fced849ee1bb76e7391b93eb12

root@:/home/gxemul/tprub/div# sha1 prueba-cog
SHA1 (prueba-cog) = de9f2c7fd25e1b3afad3e85a0bd17d9b100db4b3
root@:/home/gxemul/tprub/div# ./sha prueba-cog
de9f2c7fd25e1b3afad3e85a0bd17d9b100db4b3

```

4. Conclusiones

El presente trabajo permitió la familiarización con las herramientas de compilación de código C y código assembly en un entorno que emula la arquitectura MIPS 32, asegurando la portabilidad del programa.

Además nos permitió conocer en detalle como se comporta el stack de una función en la programación en *Assembly*. Para esto último, fue de gran ayuda conocer previamente la implementación de las funciones en language C, para luego hacer la “traducción” a *Assembly* teniendo en cuenta la cantidad de argumentos, variables locales, tamaño de los datos, etc.

5. Apéndice

5.1. Código Fuente: main.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 #define ERROR_OPEN_FILE 10
6 #define MAX_COUNT_FILES 50
7 #define HASH_LENGTH 20
8 #define MAX_FILE_SIZE 1000
9
10 void printHelp();
11 void printVersion();
12 void printError(char* msgError, int codeError);
13 char* setFileSize(FILE* fp, long long int *length);
14
15 int readFromStdInput(int argumentCount);
16
17 long long int calcularRelleno(long long int longitudOriginal);
18 void cargarTrozos(char *bloque, unsigned int *Trozos);
19 void algoritmoShal(unsigned int *Trozos, unsigned *a, unsigned *b,
    unsigned *c, unsigned *d, unsigned *e);
20
21 int shal(unsigned char *resultado, char *nombre_archivo, unsigned
    long long longitudOriginal);
22 unsigned int leftrotate(unsigned int valor, int desplazamiento);
23 void asignarDatos(FILE* fp, unsigned char *bloques, long long int
    tamanoOriginal, long long int longitudRelleno);
24 void showChecksum(unsigned char *result);
25
26 int main(int argc, char* argv[]) {
27     int i = 0;
28     char* param;
29     char* files[MAX_COUNT_FILES];
30     unsigned char *result;
31     int fileCount = 0;
32     FILE* fp;
33     char *start;
34     long long int length;
35
36     if (readFromStdInput(argc)) {
37         int c;
38
39         fp = fopen("archivoAuxiliar.txt", "w+");
40         while ((c = fgetc(stdin)) != EOF) {
41             fputc(c, fp);
42         };
43
44         start = setFileSize(fp, &length);
45         long long int longitudRelleno = calcularRelleno(length);
46         char *bloques = malloc(longitudRelleno/8);
47
48         // se almacena el tamanoOriginal al final
49         asignarDatos(fp, bloques, length, longitudRelleno);
50         shal(result, bloques, length);
51         showChecksum(result);
52
53         fclose(fp);
54         remove("archivoAuxiliar.txt");
55     }
```

```

56         return 0;
57
58     } else if (argc >= 2) { // Parse arguments
59         param = *(argv + 1);
60         if ((strcmp(param, "-h") == 0) || (strcmp(param, "--help"
61             == 0) ) {
62             printHelp();
63         }
64         else if ((strcmp(param, "-V") == 0) || (strcmp(param, "--
65             version") == 0)) {
66             printVersion();
67         }
68     }
69
70     // Search for files
71     for(i = 1; i < argc; i++) {
72         if (*argv[i] != '-') {
73             files[fileCount++] = argv[i];
74         }
75     }
76
77     // Process each file
78     for(i = 0; i < fileCount; i++) {
79         result = malloc(HASH_LENGTH);
80
81         if (NULL == (fp = fopen(files[i], "r"))) {
82             fprintf(stderr, "Files '%s' doesn't exist.", files[i]);
83             exit(ERROR_OPEN_FILE);
84         }
85
86         start = setFileSize(fp, &length);
87         long long int longitudRelleno = calcularRelleno(length);
88         char *bloques = malloc(longitudRelleno/8);
89
90         fp = fopen(files[i], "r");
91
92         // se almacena el tamañoOriginal al final
93         asignarDatos(fp, bloques, length, longitudRelleno);
94         sha1(result, bloques, length);
95         fclose(fp);
96     }
97
98     showChecksum(result);
99
100     return 0;
101 }
102
103 char* setFileSize(FILE* fp, long long int *length) {
104     int character;
105     char *start;
106     int n = 0;
107
108     fseek(fp, 0, SEEK_END);
109     *length = ftell(fp);
110
111     fseek(fp, 0, SEEK_SET);
112
113     start = malloc(MAX_FILE_SIZE);
114
115     if ((*length) > MAX_FILE_SIZE) {
116         start = realloc(start, (*length));
117     }

```

```

116     *length *= 8; // devuelve el tamaño en bits
117
118
119     while ((character = fgetc(fp)) != EOF) {
120         start[n++] = (char)character;
121     }
122     start[n] = '\0';
123
124     return start;
125 }
126
127 void printHelp()
128 {
129     fprintf(stdout, "$ tp1 -h\n");
130     fprintf(stdout, "Usage:\n");
131     fprintf(stdout, "    tp1 -h\n");
132     fprintf(stdout, "    tp1 -V\n");
133     fprintf(stdout, "    tp1 [file ...]\n");
134     fprintf(stdout, "Options:\n");
135     fprintf(stdout, "    -V, --version    Print version and quit.\n");
136     ;
137     fprintf(stdout, "    -h, --help      Print this information and\n");
138     fprintf(stdout, "quit.\n\n");
139     fprintf(stdout, "Examples:\n");
140     fprintf(stdout, "    tp1 foo\n");
141     fprintf(stdout, "    echo \"hello\" | tp1\n");
142 }
143
144 int sha1(unsigned char *resultado, char *bloques, unsigned long
145          long longitudOriginal)
146 {
147     long long int longitudRelleno = calcularRelleno(
148         longitudOriginal);
149     // Preprocesamiento
150
151     // longArchivo/tamaño bloque    bloque = 512bits
152     long long int cantBloques = longitudRelleno/512;
153
154     // procesar el bloque en 4 rondas de 20 pasos cada ronda
155     // la memoria temporal cuenta con 5 registros ABCDE
156     unsigned A=0x67452301;
157     unsigned B=0xEFCDAB89;
158     unsigned C=0x98BADCFE;
159     unsigned D=0x10325476;
160     unsigned E=0xC3D2E1F0;
161
162     // big endian
163     unsigned int trozos[80];
164
165     int i;
166     unsigned a = 0;
167     unsigned b = 0;
168     unsigned c = 0;
169     unsigned d = 0;
170     unsigned e = 0;
171
172     while (cantBloques--)
173     {
174         cargarTrozos(bloques, trozos);
175
176         a = A;
177         b = B;

```

```

174         c = C;
175         d = D;
176         e = E;
177
178         algoritmoSha1(trozos, &a, &b, &c, &d, &e);
179
180         A += a;
181         B += b;
182         C += c;
183         D += d;
184         E += e;
185     }
186
187     // hh = (h0 leftshift 128) or (h1 leftshift 96) or (h2
188         //         leftshift 64)
189     //         or (h3 leftshift 32) or h4
190     for(i = 0; i < 4; i++)
191     {
192         resultado[i] = (A >> (24 - 8 * i));
193         resultado[i + 4] = (B >> (24 - 8 * i));
194         resultado[i + 8] = (C >> (24 - 8 * i));
195         resultado[i + 12] = (D >> (24 - 8 * i));
196         resultado[i + 16] = (E >> (24 - 8 * i));
197     }
198     return 0;
199 }
200
201 // un bloque tiene 64 bytes
202 void asignarDatos(FILE *fp, unsigned char *bloques, long long int
203     tamanioOriginal, long long int longitudRelleno)
204 {
205     char caracter;
206     int indice = 0;
207     int i = 0;
208     int cantBitsRelleno = 0;
209     char relleno = 0x80; // minimo relleno
210     char rellenoCero = 0x00;
211     unsigned mascara = 0x0000000000000000FF;
212
213     while( (caracter =getc(fp)) != EOF)
214     {
215         *(bloques + indice) = caracter;
216         indice++;
217     }
218
219     cantBitsRelleno = (longitudRelleno - tamanioOriginal);
220
221     // restamos 64 bits de tamanio y los 8 bits basicos de relleno;
222     cantBitsRelleno = cantBitsRelleno - 64 - 8;
223     *(bloques + indice) = relleno;
224     indice++;
225
226     for(i = 0; i < (cantBitsRelleno / 8); i++)
227     {
228         *(bloques + indice) = rellenoCero;
229         indice++;
230     }
231
232     for(i = 0; i < 8; i++)
233     {

```

```

234         *(bloques+indice) = (tamanioOriginal>>(56-8*i)) & mascara;
235         indice++;
236     }
237 }
238 }
239
240 // devuelve el tamaño en bits
241 long long int calcularTamañoArchivo(char* nombreFile)
242 {
243     long long int tamaño =0;
244     FILE *fp = fopen(nombreFile,"r");
245     fseek(fp,0,SEEK_END);
246     tamaño = ftell(fp)*8;
247     return tamaño;
248 }
249
250
251 unsigned int leftrotate(unsigned int valor,int desplazamiento)
252 {
253     desplazamiento %=32;
254     unsigned retorno;
255     retorno = (valor << desplazamiento) | (valor >> (32 -
256         desplazamiento));
257     return retorno;
258 }
259
260 void printVersion()
261 {
262     fprintf(stdout, "Copyright (c) 2015\n");
263     fprintf(stdout, "Conjunto de instrucciones MIPS. v1.0.0\n\n");
264 }
265
266 void printError(char* msgError, int codeError)
267 {
268     fprintf(stderr, "%s\n", msgError);
269     exit(codeError);
270 }
271
272 void showChecksum(unsigned char *result) {
273     int i;
274     for(i = 0; i < 20; i++) {
275         unsigned char aux = result[i];
276         aux<<=4;
277         aux>>=4;
278         printf("%x", (result[i]>>4));
279         printf("%x",aux);
280     }
281     printf("\n");
282 }
283
284 int readFromStdInput(int argumentCount) {
285     return (int)(argumentCount < 2);
286 }

```

../src/main.c

5.2. Código Fuente: relleno.S

```

1
2 #include <mips/regdef.h>
3

```

```

4  .text
5  .align 2
6
7  .globl calcularRelleno
8  .ent  calcularRelleno
9
10 calcularRelleno:
11  .frame $fp, 48, ra
12  .set noreorder
13  .cpload t9
14  .set reorder
15  subu  sp, sp, 48
16  .cpstore 0
17
18
19  sw  ra, 44(sp)
20  sw  $fp, 40(sp)
21  sw  gp, 36(sp)
22  move $fp, sp
23
24  sw  a0, 48($fp)
25  sw  a1, 52($fp)
26  #sw a2, 56($fp)
27  #sw a3, 60($fp)
28
29  #obtencion del tamaño del relleno
30
31  #-----operaciones en 64 bits
32
33  li  t0, 0
34  li  t1, 0      #el tamaño debe venir en bits
35  move t0, a0    #t1|t0 = longArchivoRelleno
36  move t1, a1
37  li  t4, 512
38
39  bucle_relleno:
40  remu  t2, t0, t4  #calculo el valor de la longitud en modulo 512 (
41  resto)
42  beqz  t2, fin_bucle_relleno  #no es necesario considerar la parte
43  mas significativa
44  #suma de los 64 bits
45  addiu t2, t0, 8   #agrego 1 byte
46  sltu  t3, t2, t0  #(t3 =1) de carry si el resultado es mas chico
47  que el sumando
48  move  t0, t2     #volvamos el valor a t0
49  beqz  t3, bucle_relleno
50  addiu t1, t1, 1   #sumamos 1 a la parte mas significativa
51  b  bucle_relleno  #solucionar la suma en 64 bits
52
53  fin_bucle_relleno:
54  slti  t2, t0, 65  #si t0<65 entonces t2 =1 y agregamos 512 bits
55  beqz  t2, obtener_bloque
56  addiu t2, t0, 512  #agrega un bloque de 512 extra
57  sltu  t3, t2, t0
58  move  t0, t2
59  beqz  t3, obtener_bloque #si no hay acarreo no sumar al mas
60  significativo
61  addiu t1, t1, 1    #t1|t0 = longArchivoRelleno
62
63  obtener_bloque:

```

```

60 sw t0,16($fp)    #guardo la longitud relleno en 64 bits en el
    stack
61 sw t1,20($fp)
62 sw a0,24($fp)    #guardo la longitud original en 64 bits en el
    stack
63 sw a1,28($fp)
64
65 move v0,t0
66 move v1,t1
67
68 lw $fp, 40(sp)
69 lw ra, 44(sp)
70 lw gp, 36(sp)
71 addu sp, sp, 48
72
73 # Retorno.
74 #
75 j ra
76 .end calcularRelleno

```

../src/relleno.S

5.3. Código Fuente: trozo.S

```

1 #
2 #s0 puntero a bloques
3 #s1 puntero a trozos
4 #
5
6
7 #include <mips/regdef.h>
8
9 .text
10 .align 2
11
12 .globl cargarTrozos
13 .ent cargarTrozos
14
15 cargarTrozos:
16 .frame $fp, 48, ra
17 .set noreorder
18 .cpload t9
19 .set reorder
20 subu sp, sp, 40
21 .cpstore 0
22
23 sw gp,24(sp)
24 sw $fp,28(sp)
25 sw ra,32(sp)
26 move $fp, sp
27
28 #guardo los parametros con que vine #void cargarTrozos(char *
    bloques,int *indice,unsigned int *trozos)
29
30 sw a0, 40($fp)    #ptero a bloques
31 sw a1, 44($fp)    # ptero a trozos
32 sw a2, 48($fp)    # nada
33 sw a3, 52($fp)    # nada
34
35 #cargo los parametros
36 move s0,a0

```

```

37 | move s1,a1
38 |
39 | sw zero,16($fp) # i=0
40 | li t0,255 # mascara 0x000000ff
41 | sw t0,20($fp) #mascara en 28
42 |
43 | FOR16PRIMEROS:
44 | #traigo los datos
45 | lw t0,16($fp) #t0 cargo i
46 | lw t5,20($fp) #t5 = mascara
47 |
48 | #comienzo
49 |
50 | lbu t6,0(s0) #t6 cargo el byte al que apunta (t4 = bloques +
    | indice), lo carga de 0 a 7
51 | and t6,t6,t5 #t6 con el and de la mascara, queda 00 00 00
    | ALGO
52 | sll t6,t6,8 #lo nuevo 8 lugares, quedaria 00 00 ALGO 00
53 |
54 | addiu s0,s0,1 #(*(bloques+indice+1)
55 | lbu t7,0(s0) #t7 cargo el byte al que apunta (t4 = bloques +
    | indice + 1), lo carga de 0 a 7
56 | and t7,t7,t5 #t7 con el and de la mascara, queda 00 00 00
    | ALGO
57 | or t6,t6,t7 #suma logica de t6 y t7, quedaria 00 00 ALGO ALGO
58 | sll t6,t6,8 #nuevo quedaria 00 ALGO ALGO 00
59 |
60 | addiu s0,s0,1 #(*(bloques+indice+2) (ya habia sumado 1
    | antes)
61 | lbu t7,0(s0) #t7 cargo el byte al que apunta (t4 = bloques +
    | indice + 2), lo carga de 0 a 7
62 | and t7,t7,t5 #t7 con el and de la mascara, queda 00 00 00
    | ALGO
63 | or t6,t6,t7 #suma logica de t6 y t7, quedaria 00 ALGO ALGO
    | ALGO
64 | sll t6,t6,8 #nuevo quedaria ALGO ALGO ALGO 00
65 |
66 | addiu s0,s0,1 #(*(bloques+indice+3) (ya habia sumado 1
    | antes)
67 | lbu t7,0(s0) #t6 cargo el byte al que apunta (t4 = bloques +
    | indice + 3), lo carga de 0 a 7
68 | and t7,t7,t5 #t6 con el and de la mascara, queda 00 00 00
    | ALGO
69 | or t6,t6,t7 #suma logica de t6 y t7, quedaria ALGO ALGO ALGO
    | ALGO
70 |
71 | #GUARDO
72 | sw t6, 0(s1) #guardo t6 en (t2 = trozos + i*4)
73 |
74 | #AUMENTO VARIABLES
75 |
76 |
77 | addiu t0,t0,1 #i = i + 1
78 | sw t0,16($fp) #guardo
79 |
80 | addiu s1,s1,4 # avanza a la sig palabra
81 | addiu s0,s0,1 # avanza al sig byte
82 |
83 | addiu t7,t0,-16
84 | beqz t7, FOR16TO80
85 |
86 | b FOR16PRIMEROS

```



```

87
88
89 FOR16TO80:
90     #traigo los datos
91     lw  t0,16($fp)    #t0 cargo i
92
93     addiu  s1,s1,-12    #t3 apunta a trozos(i - 3).. 3 lugares = 4
94             bytesx3 = 12 bytes
95     lw  t4,0(s1)    #t4 traigo la palabra a la q apunta t3
96     addiu  s1,s1,12
97
98     addiu  s1,s1,-32    #t5 apunta a trozos(i - 8)
99     lw  t6,0(s1)    #t6 traigo la palabra a la que apunta t5
100     addiu  s1,s1,32
101
102     xor    t4,t4,t6    #t4 = *(trozos + (i-3)) ^ *(trozos + (i-8))
103
104     addiu  s1,s1,-56    #t5 apunta a trozos(i - 14)
105     lw  t6,0(s1)    #t6 traigo la palabra a la que apunta t5
106     addiu  s1,s1,56
107
108     xor    t4,t4,t6    #t4 = *(trozos + (i-3)) ^ *(trozos + (i-8)) ^
109             *(trozos + (i-14))
110
111     addiu  s1,s1,-64    #t5 apunta a trozos(i - 16)
112     lw  t6,0(s1)    #t6 traigo la palabra a la que apunta t5
113     addiu  s1,s1,64
114
115     xor    t4,t4,t6    #t4 = *(trozos + (i-3)) ^ *(trozos + (i-8)) ^
116             *(trozos + (i-14)) ^ *(trozos + (i-16));
117
118     rol  t4,t4,1    #leftrotate 1 Åi?
119
120     sw  t4,0(s1)    #guardo t4 en t2 = trozos + i*4 es decir apunta a
121             trozos[i]
122
123     addiu  t0,t0,1    #i = i + 1
124     sw  t0,16($fp)    #guardo
125     addiu  s1,s1,4    #t2 = avanzo a la siguiente palabra
126
127     addiu  t7,t0,-80
128     beqz  t7,FinDelFor
129
130     b FOR16TO80
131
132 FinDelFor:
133
134     lw  $fp,28(sp)
135     lw  ra, 32(sp)
136     lw  gp, 24(sp)
137     addu  sp, sp, 40
138
139     # Retorno.
140     #
141     j  ra
142     .end  cargarTrozos

```

../src/trozo.S

5.4. Código Fuente: algoritmo.S

```

1 #
2 #a0 puntero a trozos
3 #a1 a
4 #a2 b
5 #a3 c
6 #a4 d
7 #a5 e
8 #
9
10
11
12 #include <mips/regdef.h>
13
14 .text
15 .align 2
16
17 .globl algoritmoSha1
18 .ent algoritmoSha1
19
20 algoritmoSha1:
21 .frame $fp, 48, ra
22 .set noreorder
23 .cload t9
24 .set reorder
25 subu sp, sp, 48
26 .cprestore 0
27
28
29 sw gp,32(sp)
30 sw $fp,36(sp)
31 sw ra,40(sp)
32 move $fp, sp
33
34 sw a0, 48($fp) # trozos
35 sw a1, 52($fp) # a
36 sw a2, 56($fp) # b
37 sw a3, 60($fp) # c
38
39 la t0,4(a3) # dir
40 la t1,8(a3) # dir
41
42 sw t0, 64($fp) # d
43 sw t1, 68($fp) # e
44
45
46 #-----algoritmo-----
47
48 #
49 #s2 = a
50 #s3 = b
51 #s4 = c
52 #s5 = d
53 #s6 = e
54 #s7 = f (funcion)
55 #
56 #carga variables en s0..s7
57
58 move s1,a0
59 lw s2,0(a1)
60 lw s3,0(a2)
61 lw s4,0(a3)
62 lw s5,4(a3)

```

```

63 lw s6,8(a3)
64
65 #inicializar i de vuelta
66 sw zero,16($fp) # i = 0
67 lw t6,16($fp) # traigo i, t6 = i
68
69 PROCESO0A19:
70 lw t6,16($fp) # traigo i, t6 = i
71 move s7,zero
72
73 and s7,s3,s4 #en f(s7) <— (b & c)
74 not t0,s3 #t0 niego b
75 and t1,t0,s5 #en t1 <— ((~b) & d)
76 xor s7,s7,t1 #f(s7) <— (b & c) ^ ((~b) & d)
77
78 lw t2,ctek1 #carga en t2, k1
79 sw t2,20($fp) #es donde esta k en el stack
80
81
82 jal ASIGNACIONTEMPORAL
83
84
85 addiu t6,t6,1
86 sw t6,16($fp)
87
88 addiu t7,t6,-20
89 beqz t7,PROCESO20A39
90
91 b PROCESO0A19
92
93
94 PROCESO20A39:
95 lw t6,16($fp) # traigo i, t6 = i
96 move s7,zero
97
98 xor s7,s3,s4 #f <— b ^ c
99 xor s7,s7,s5 #f <— b ^ c ^ d
100
101 lw t0,ctek2 #carga en t0, k2
102 sw t0,20($fp) #es donde esta k en el stack
103
104
105 jal ASIGNACIONTEMPORAL
106
107
108 addiu t6,t6,1
109 sw t6,16($fp)
110
111 addiu t7,t6,-40
112 beqz t7,PROCESO40A59
113
114 b PROCESO20A39
115
116 PROCESO40A59:
117 lw t6,16($fp) # traigo i, t6 = i
118 move s7,zero
119
120 and s7,s3,s4 #en f(s7) <— (b & c)
121 and t0,s3,s5 #en t0 <— (b & d)
122
123 or s7,s7,t0 #f <— (b & c) | (b & d)
124

```

```

125 and    t0,s4,s5      #en t0 <— (c & d)
126
127 or     s7,s7,t0      #f <— (b & c) | (b & d) | (c & d)
128
129 lw     t0,ctek3      #carga en t0, k3
130 sw     t0,20($fp)    #es donde esta k en el stack
131
132
133 jal    ASIGNACIONTEMPORAL
134
135
136 addiu   t6,t6,1
137 sw     t6,16($fp)
138
139 addiu   t7,t6,-60
140 beqz    t7,PROCESO60A79
141
142 b       PROCESO40A59
143
144 PROCESO60A79:
145 lw     t6,16($fp)    # traigo i, t6 = i
146 move    s7,zero
147
148 xor     s7,s3,s4      #f <— b ^ c
149 xor     s7,s7,s5      #f <— b ^ c ^ d
150
151 lw     t0,ctek4      #carga en t0, k4
152 sw     t0,20($fp)    #es donde esta k en el stack
153
154
155 jal    ASIGNACIONTEMPORAL
156
157 addiu   t6,t6,1
158 sw     t6,16($fp)
159
160 addiu   t7,t6,-80
161 beqz    t7,return_algoritmo
162
163 b       PROCESO60A79
164
165
166 ASIGNACIONTEMPORAL:
167
168 #ESTO FINALIZA EL FOR DE 80 CON EL ALGORITMO PER SE
169
170 rol     t0,s2,5       #en t0 leftrotate a 5
171 addu    t0,t0,s7       #t0 = t0 + f
172 addu    t0,t0,s6       #t0 = t0 + f + e
173 lw     t1,20($fp)     #t1 cargo k
174 addu    t0,t0,t1       #t0 = t0 + f + e + k
175
176 lw     t1,16($fp)     #t1 cargo i
177 sll     t2,t1,2        #t2 = i*4
178 addu    t3,s1,t2       #t3 = trozos + i*4 es decir apunta a trozos[i]
179 lw     t4,0(t3)       #t4 traigo la palabra a la q apunta t3
180
181 addu    t0,t0,t4       #t0 = t0 + f + e + k + trozos[i]
182
183 sw     t0,24($fp)     #guardo t0 en temp (24($fp))
184 #sw     s5,0(s6)       #e = d;
185 move    s6,s5
186 #sw     s4,0(s5)       #d = c;

```

```

187     move s5,s4
188     rol s4,s3,30    #c = leftrotate(b ,30);
189     #sw s2,0(s3)    #b = a;
190     move s3,s2
191     #sw t0,0(s2)    #a = temp;
192     move s2,t0
193
194     #AUMENTO I
195     #addiu t1,t1,1    #i = i + 1
196     #sw t1,16($fp)    #guardo
197
198     jalr ra
199
200 return_algoritmo:
201 #-----FIN algoritmo
202
203
204     lw a0, 48($fp)    # trozos
205     lw a1, 52($fp)    # a
206     lw a2, 56($fp)    # b
207     lw a3, 60($fp)    # c
208     lw t6, 64($fp)    # d
209     lw t7, 68($fp)    # e
210
211     sw s2,0(a1)
212     sw s3,0(a2)
213     sw s4,0(a3)
214     sw s5,0(t6)
215     sw s6,0(t7)
216
217
218     lw $fp,36(sp)
219     lw ra, 40(sp)
220     lw gp, 32(sp)
221     addu sp, sp, 48
222
223     # Retorno.
224     #
225     j ra
226     .end algoritmoSha1
227
228
229
230     .data # comienza zona de datos
231
232     cteA: .word 0x67452301
233     cteB: .word 0xEFCDAB89
234     cteC: .word 0x98BADCFE
235     cteD: .word 0x10325476
236     cteE: .word 0xC3D2E1F0
237     ctek1: .word 0x5A827999;
238     ctek2: .word 0x6ED9EBA1;
239     ctek3: .word 0x8F1BBCDC;
240     ctek4: .word 0xCA62C1D6;

```

../src/algoritmo.S

5.5. Bibliografía

Referencias

- [1] *US Secure Hash Algorithm 1 (SHA1)*
<http://tools.ietf.org/html/rfc3174>
- [2] *SHA1 Message Digest Algorithm Overview*
<http://www.herongyang.com/Cryptography/SHA1-Message-Digest-Algorithm-Overview.html>
- [3] *SHA 1*
<http://en.wikipedia.org/wiki/SHA-1>
- [4] *MIPS Assembly Language Guide*
http://www.cs.uni.edu/~fienup/cs041s08/lectures/lec20_MIPS.pdf
- [5] *MIPS Instruction Reference*
<http://www.mrc.uidaho.edu/mrc/people/jff/digital/MIPSir.html>