Trabajo práctico 1:
Conjunto de instrucciones MIPS

Daddario Rubén, *Padrón Nro. 83122*
ruben.daddario@gmail.com

Amura Federico, *Padrón Nro. 95202*
federicoamura@gmail.com

Grupo Nro. X - 2do. Cuatrimestre de 2014
66.20 Organización de Computadoras
Facultad de Ingeniería, Universidad de Buenos Aires

Jueves 27 de septiembre de 2014

Abstract

El presente informe se corresponde al trabajo práctico número 1 de la asignatura 66.20 Organización de Computadoras. El trabajo consiste en la implementacion de un programa que por consola verifica la integridad de los tags anidados en un archivo xml. Es decir, el programa verifica que todo tag abierto, tenga luego su correspondiente cerrado y que entre ellos esten bien anidados. La finalidad del trabajo es en realidad, realizar una parte de la programacion en Assembly para MIPS32.

Contents

1 Introducción

MIPS (Microprocessor without Interlocked Pipeline Stages) es un procesador con arquitectura RISC (reduced instruction set computer) desarrollado por MIPS Technologies. Este tipo de procesadores se encuentra en muchos sistemas embebidos y dispositivos de la actualidad como consolas de videojuegos, routers, etc. En particular, debido a que los diseñadores crearon un conjunto de instrucciones tan claro, los cursos sobre arquitectura de computadoras en unviersidades y escuelas técnicas a menudo se basan en la arquitectura MIPS.

Existen múltiples versiones de este procesador, siendo los más recientes MIPS32 y MIPS64, implementadas para 32 y 64 bits, respectivamente.

2 Desarrollo

El programa a elaborar es una implementación en ANSI C y Assembly para MIPS32 de un XML validator. Resumiendo, su funcion es la de validar la correcta utilizacion de los tags en un archivo XML, es decir, ver que todo tag sea abierto y cerrado y que esten correctamente anidados entre si.

3 Diseño y detalles de implementación

3.1 Recursos utilizados

Para resolver el problema planteado utilizamos la herramienta GXemul para emular un entorno MIPS y el sistema operativo NetBSD. También utilizamos code::blocks para la codificación de la aplicación en C y un repositorio SVN para compartir el proyecto. Para la codificacion en Assembly, se usaron procesadores de texto como el Gedit. Finalmente el programa fue compilado utilizando GCC.

Para implementar el programa utilizamos el lenguaje de programación C, en su totalidad para la version portable, y las partes especificas para MIPS32 fueron escritas en el Assembly correspondiente siguiendo el ABI de la catedra. Para la realización del informe utilizamos LaTeX.

3.2 Diseño

El programa se compone de múltiples archivos de cabecera .h con sus correspondientes archivos .c donde se realiza la implementación de los métodos utilizados.

```
tp1 [OPTION]...[FILE]...
```

Las opciones pueden ser:

- -h, –help Prints usage information
- -V, –version Prints version information
- -i, –input Path to input file (-i - for stdin)

3.3 Implementación

3.3.1 Portabilidad

Para proveer portabilidad se implementaron las funciones en lenguaje C para dar soporte genérico a aquellos entornos que carezcan de una versión más específica. Siempre cumpliendo con el estandar ANSI C.

3.3.2 Código y algoritmos

El programa se compone de un archivo main con el punto de entrada al ejecutable y desde aquí se invócan al resto de los archivos que definen las estructuras de datos necesarias para la ejecución. Sin ampliar demasiado ya que el foco del trabajo es la codificacion en Assembly, el programa se dividio en 5 partes.

- Main, encargado de la entrada del programa e invocar las demas partes
- Options Reader, cuya tarea es recibir el vector de parametros para la ejecucion del programa y procesarlo para que su utilizacion en el programa sea mas facil.
- Program Options, que sirve para imprimir la ayuda y liberar las opciones de ejecucion alojadas en memoria dinamica
- File reader, es la parte encargada de cargar el archivo en memoria (se supone que la memoria puede alojarlo)
- Validator, que incluye todas las funciones que se utilizaran para la validacion del archivo. Esta seccion fue luego dividida entre sus funciones para facilitar el paso a Assembly.

3.3.3 Desarrollo en Assembly

Primero se desarrollo la version en C ya que la version general es escrita totalmente en este lenguaje y la version especifica para MIPS32 contiene la mayoria del codigo tambien en el mismo. Como solo se tenia que desarrollar la funcion Validate y las que este utilice en Assembly, se tomaron ciertas medidas extra en su desarrollo. La mas notoria es que no esta toda contenida en un solo archivo .c si no que se encuentra en varios, cada uno incluyendo una sola funcion. Esta division tiene como finalidad facilitar el paso de esas funciones al codigo ensamblador, es una variante de la estrategia de division

y conquista. Gracias a esto podemos ir pasando de a una funcion por vez, como tambien trabajarlas aisladamente e incluso probarlas cuando una ya esta lista, sin tener que esperar al resto.

La codificacion en Assembly se hizo teniendo en cuenta el ABI de la catedra.

4 Ejecuciones de prueba

En esta seccion se muestran algunos ejemplos de ejecucion del programa

comando utilizado: ./validate sample.xml

Figure 1: Salida de una ejecucion de prueba en NetBSD sobre MIPS32

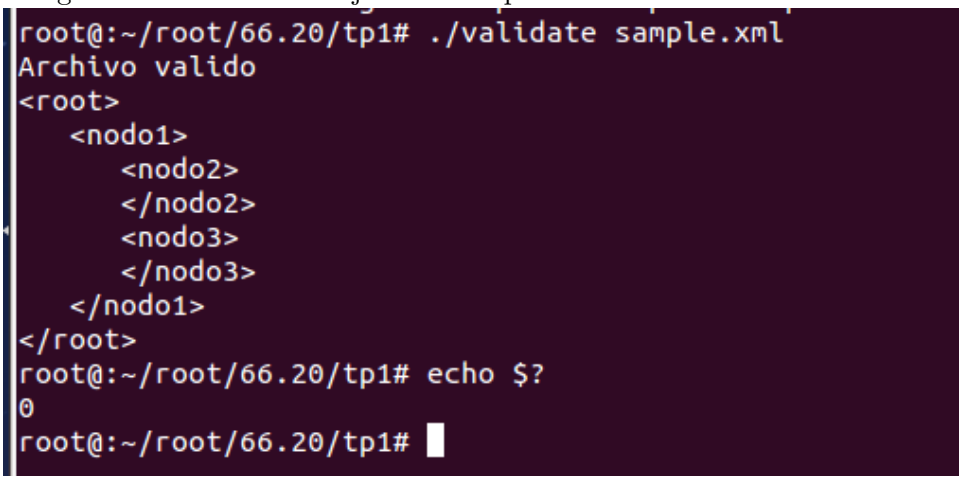
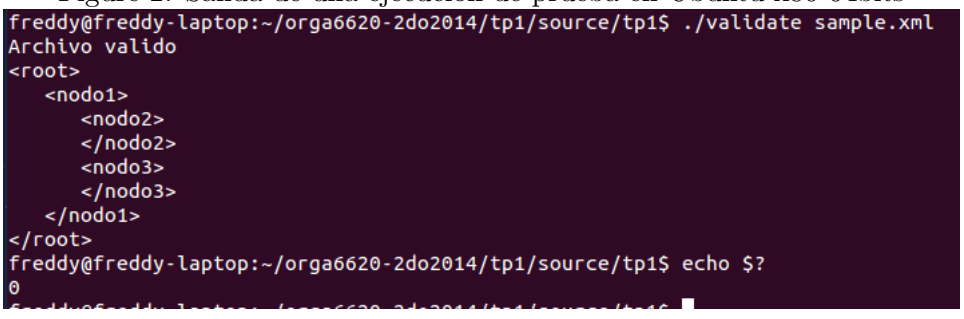


Figure 2: Salida de una ejecucion de prueba en Ubuntu x86 64bits



A continuación se muestra la ejecución con un archivo inválido

comando utilizado: ./validate invalid1.xml

Figure 3: Salida de una ejecucion de un archivo mal armado en Linux

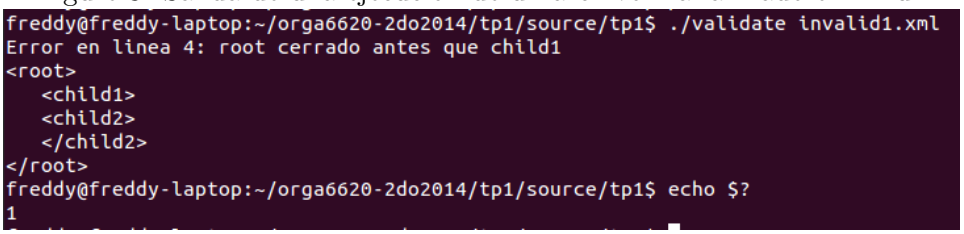
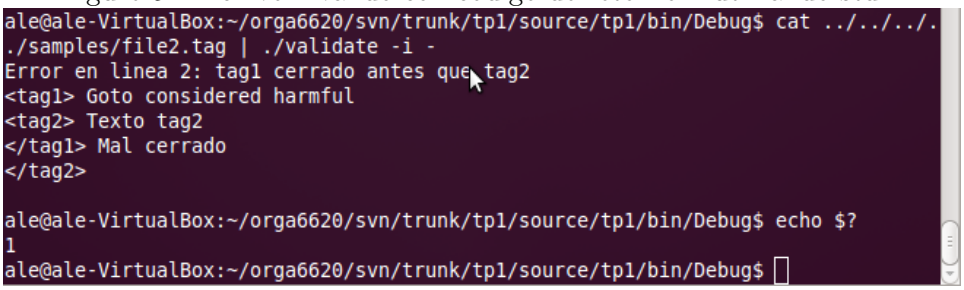


Figure 4: Archivo valido con codigo de retorno 0

Figure 5: Archivo invalido con codigo de retorno 1 utlizando stdin



5 Stacks

A modo de ayuda graficamos y mostramos un esquema de los stacks armados en las funciones assembly

Figure 6: Stack funcion validate.S

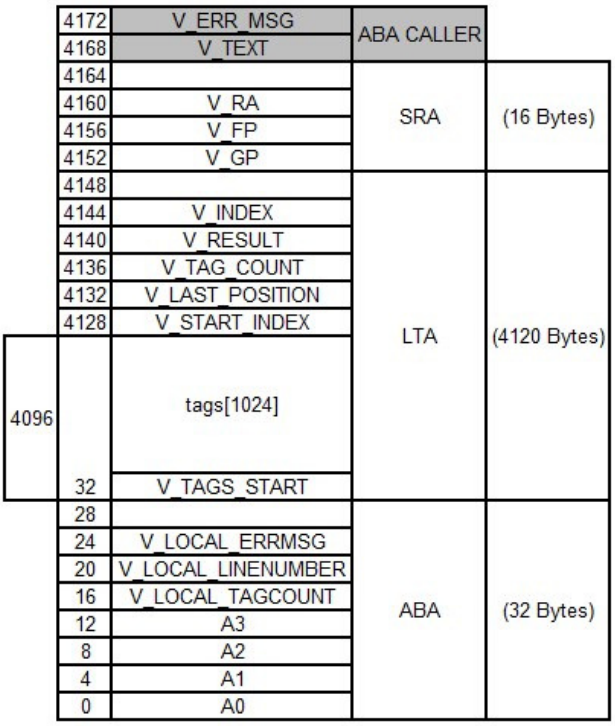


Figure 7: Stack funcion isOpenTag.S

Figure 8: Stack funcion findTag.S

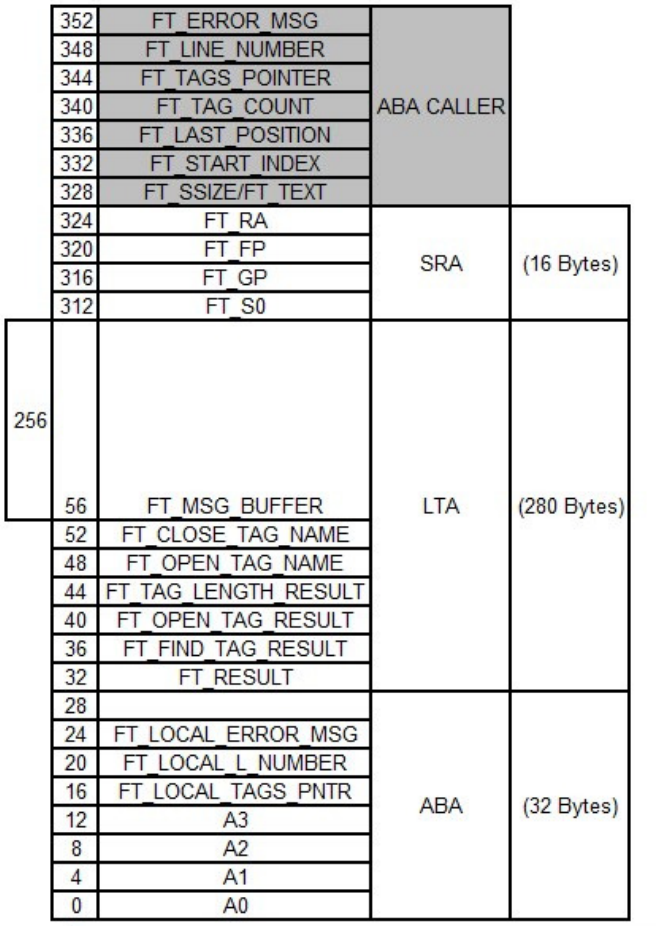


Figure 9: Stack funcion getTagName.S

48	GTN_TAG_POINTER	ABA CALLER	
44		SRA	(16 bytes)
40	GTN_RA		
36	GTN_FP		
32	GTN_GP		
28		LTA	(16 bytes)
24	GTN_NAME_START		
20	GTN_NAME_LENGTH		
16	GTN_TAGNAME		
12		ABA	(16 bytes)
8	A2		
4	A1		
0	A0		

Figure 10: Stack funcion findTagInner.S

76	FTI_LINE_NUMBER	ABA CALLER	
72	FTI_LAST_POS		
68	FTI_START_INDEX		
64	FTI_TEXT		
60		SRA	(16 Bytes)
56	FTI_RA		
52	FTI_FP		
48	FTI_GP		
44			(40 bytes)
40	FTI_RESULT		
36	FTI_TOTAL_LENGTH		
32	FTI_TAG_NAME		
28	FTI_TAG_START_POSITION		
24	FTI_TEXT_LENGTH		
20	FTI_CURRENT_POS		
16	FTI_FOUND		
12		ABA	(16 Bytes)
8	A2		
4	A1		
0	A0		

6 Compilacion

La compilacion de la version generica, siendo totalmente en C, no varia de un programa comun, se hace simplemente con un archivo Makefile, que envia las ordenes de compilacion para cada archivo y luego la de vinculacion para generar el ejecutable.

En el caso de la version con codigo Assembly, el proceso varia en dos cuestiones. Primero, los archivos que se codificaron en el lenguaje propio del microprocesador con extension .S (mayuscula), deben precompilarse con gcc junto con los .c. Los archivos con extension .S (mayuscula) son interpretados por gcc como archivos a precompilar, a diferencia de los archivos con extension .s (minuscula) que son interpretados como codigo assembly ya pasado por el preprocesador. Un ejemplo de compilacion es el siguiente:

```
gcc -Wall -c *.c *.S
```

Luego debemos linkear los archivos para generar el archivo ejecutable:

```
gcc -o validate *.c *.S
```

Mas alla de estas dos consideraciones, si todo se ejecuto correctamente entonces el proceso sigue siendo el mismo, terminando con un ejecutable que se comporta de manera similar pero con diferencias internas dado su diferente codificacion.

7 Conclusiones

A partir de este trabajo adquirimos experiencia en la compilacion en Assembly y su incorporacion a codigo de un nivel superior.

Tuvimos ademas que lidiar con los warnings generados por el compilador en Assembly y realizar depuracion al mas bajo nivel posible antes de pasar a codigo en binario, viendo como se comportan las instrucciones en un lenguaje como C cuando se las traduce a codigo assembly y luego a codigo maquina.

Tambien vimos como manejar la informacion con la que trabaja el procesador, viendo las instrucciones basicas que este puede realizar y los bloques de datos con los que este opera.

References

[1] J. L. Hennessy and D. A. Patterson, Computer Architecture. A Quantitative Approach, 3ra Edicion, Morgan Kaufmann Publishers, 2000.

[2] Grupo de la materia http://groups.yahoo.com/group/orga6620/

[3] Wikipedia en.wikipedia.org

[4] Code::Blocks IDE: www.codeblocks.org