Lenguaje Ensamblador y la arquitectura MIPS

José Alejandro Logreira Ávila Código: 261722
David Ricardo Martínez Hernández Código: 261931
Edwin Fernando Pineda Vargas Código: 262100

***Palabras clave*— Assembly, Lenguaje, Mips32, Multiplicación QtSpim, Suma.**

***Resumen*— SPIM es uno de los simuladores de lenguaje ensamblador con que contamos actualmente. Mediante esta práctica, aprenderemos a utilizarlo, de tal forma que a futuro le podamos usar como herramienta en el desarrollo del proyecto de final de curso.**

I. Objetivo General

- Programar una aplicación en lenguaje ensamblador con el set de instrucciones de MIPS32 usando el simulador Spim.

II. Objetivos Específicos

- Entender el proceso de compilación de lenguaje de alto nivel y de lenguaje ensamblador.
- Programar una aplicación funcional usando el simulador Spim desarrollada en el lenguaje ensamblador.

III. Introducción

Actualmente los simuladores son de gran ayuda para la implementación de proyectos, ya que estos ofrecen una evaluación completa a bajo costo y sin exponer equipos ni personal para su desarrollo. En nuestro caso en particular, aprenderemos a utilizar el simulador SPIM, el cual nos permite simular programas en lenguaje ensamblador, de tal forma que podamos implementarlos a futuro en una FPGA u otro hardware que así lo soporte.

IV. Materiales

- Cable serial.
- Computador con Sistema Operativo Linux Ubuntu con Spim instalado.
- FPGA.
- Cable USB a mini USB.

V. Procedimiento

A. MIPS32 Y SPIM

1) Archivos assembly de QTSPIM: Los archivos que contienen el código assembly (denotados con la extensión .s), son archivos de texto plano, que contienen todas las instrucciones y directivas a usar en el ensamblado del programa final para la arquitectura de procesadores MIPS. Estos archivos se pueden crear fácilmente usando el editor de textos "Gedit", y guardando el archivo con la respectiva extensión *.S*. Por supuesto, el código assembly debe respetar cierta sintaxis y cierta organización del código, que es común para cualquier tipo de arquitectura de procesadores. Si existen errores en el código assembly, el simulador de MIPS lo hará saber al hallar errores en la compilación del código escrito en Gedit.
La interfaz de QTSpim es una interfaz gráfica, y además muy intuitiva, que permite reconocer fácilmente las funcionalidades del simulador. Para cargar un archivo *.S* para simulación, basta con ir a la pestaña de Archivo, y seleccionar la opción Cargar Archivo. En el buscador de archivos que aparece, solo son visibles los archivos con extensión *.S*.

2) Interfaz de QTSpim: La sección principal de la interfaz es la ventana que contiene el código assembly a simular, dividido por secciones, es decir, divide el código assembly cargado en una sección de "datos" que es el espacio de memoria donde se guardan las múltiples variables a usar en el programa. Se puede entender como el espacio de memoria RAM del sistema. Allí se encuentra la sección de RAM destinada para el usuario, la destinada para el Stack, y la destinada para las instrucciones de núcleo. Su organización es por columnas, donde la primera de ellas es la dirección de memoria de los datos mostrados a continuación, y las siguientes columnas son los datos en sí mismos, mostrados en base hexadecimal y en la última columna, mostrados en código ASCII (Caracteres).
Otra sección importante de la ventana del código assembly es la de "Texto", o datos permanentes. Estos son los datos que se han de escribir en la memoria de programa, es decir, las instrucciones del programa a ejecutar. Se puede entender como la memoria ROM o no volati. Allí se encuentran también distintas secciones que son: la sección de "texto del usuario", que es una traducción del programa escrito en Gedit a unas instrucciones base que usa el QTSpim para su simulación; la sección de "texto del núcleo", que es una nueva traducción del programa escrito en Gedit, pero con las instrucciones de núcleo del simulador.
A la izquierda de la sección principal, se encuentra una ventana que despliega los registros del procesador, tanto los 32 registros del banco de registros, como los registros de propósito especial, y los registros de la unidad de punto flotante.

Finalmente, en la parte superior del simulador se encuentran los botones de simulación y depuración del programa paso a paso, reiniciar el programa, pausarlo, terminarlo, etc.

3) Set de instrucciones para QTSpim: QTSpim es un simulador únicamente para la arquitectura MIPS32 en su primera versión, que es una de las varias arquitecturas MIPS. El set de instrucciones para MIPS32 contiene una rica cantidad de instrucciones (más de 150), incluyendo un extenso repertorio de instrucciones para operar números en punto flotante. Sin embargo, muchas de estas instrucciones son virtuales (o pseudoinstrucciones), y adicionalmente, la primera edición del ISA para MIPS32 no incluía una significativa cantidad de instrucciones que se han adicionado en posteriores ediciones. En el anexo A de este informe se incluye un resumen del ISA soportado por QTSpim.

B. El proceso de Compilación

1) ¿Cómo es el proceso de compilación?: Un compilador convierte un código de programación de un alto nivel de abstracción a un nivel bajo que es el código assembly. Existe un compilador para cada arquitectura de procesadores (ISA), precisamente porque cada procesador ha de manejar su propio set de instrucciones particular, que el compilador necesita conocer específicamente para cada arquitectura. Los compiladores hoy en día presentan un gran desarrollo y usan algoritmos de traducción de instrucciones de alto a bajo nivel muy eficientes, lo que permite obtener programas más pequeños y que se ejecutan más rápido.
Compilar un programa, en términos generales, significa generar un archivo ejecutable que desarrolle todas las funcionalidades que se programaron en el código de alto nivel. Para llegar a obtener solo un archivo ejecutable, se ha de pasar por varios procesos de traducción de código.

2) ¿Qué es un linker?: Un programa puede estar compuesto por múltiples archivos fuente (o módulos) que se encuentran separados entre sí. Cuando el compilador traduce cada archivo fuente (en lenguaje de alto nivel) a archivos en código ensamblador, se necesita un mecanismo para unir estos archivos fuente (ahora traducidos a assembly), y generar un único archivo fuente que sea la unión de todos los módulos de manera adecuada. Esta es la función del programa enlazador. En términos más específicos:

- Busca las librerías a usar en el programa.
- Determina las posiciones de memoria de cada módulo a enlazar, y el espacio que cada uno de estos va a ocupar. Reacomoda las instrucciones debidamente ajustando las referencias absolutas.
- Resolviendo las referencias relativas entre los distintos archivos objeto.

3) ¿Qué es un archivo objeto?: Los archivos objeto contienen el código ensamblador del programa a compilar, junto con toda la información necesaria para enlazar un archivo o módulo de programa con los demás. Estos archivos contienen 6 secciones:

1) El encabezado: describe el tamaño y posición de las demás secciones del archivo.
2) El segmento de texto: contiene el lenguaje de máquina para rutinas en el archivo fuente. Estas rutinas son aún imposibles de ejecutar porque sus referencias a procedimientos externos no han sido resueltas.
3) La sección de datos: contiene la representación binaria de los datos en el archivo fuente. El dato también ha de estar incompleto porque su referencias a otros archivos no han sido resueltas.
4) La información de relocalización: identifica las instrucciones y los datos que dependen de direcciones absolutas.
5) La tabla de símbolos: asocia direcciones con etiquetas externas en el archivo fuente, y lista las referencias sin resolver.
6) La información de depuración: contiene una descripción de la forma en que el programa fue compilado, tal que un depurador pueda encontrar qué dirección de instrucción corresponde a lineas de código en el archivo fuente, e imprime las estructuras de datos en una forma legible.

C. Lenguaje ensamblador, subrutinas e interrupciones

1) Lenguaje Ensamblador: Es un estándar que establece el formato y la sintaxis de escritura de las instrucciones de máquina, para cualquier arquitectura de procesadores. El lenguaje ensamblador no es un lenguaje propiamente dicho, en tanto que el set de instrucciones es único para cada tipo de procesador, y por tanto, las instrucciones usadas en el código ensamblador son solo reconocibles por el procesador respectivo que reconoce tales instrucciones.
Por tal motivo, el código ensamblador se puede entender mejor como un conjunto estándar de directrices y sintaxis de las instrucciones, que se debe respetar cuando se desee escribir un programa para cualquier tipo de procesador.
La sintaxis y el formato del código ensamblador definen una serie de reglas a seguir, tal que el programa ensamblador reconozca que tipo de comando se desea ejecutar. El formato define una organización por columnas, donde según la posición del comando escrito, es interpretado como una directriz, una etiqueta, una instrucción, un operando o un comentario. La sintaxis determina el orden de escritura de las instrucciones, esto es, el orden de los operandos y los mnemónicos de cada instrucción.

2) Ensamblador VS Alto nivel: Un lenguaje ensamblador es la primera abstracción que existe en el mundo del código de máquina, es decir, las instrucciones que se escriben en el código ensamblador son exactamente la representación en mnemónicos del código binario que representan, y que han de ser ejecutadas tal cual se escriben por el procesador que las implemente.
Un lenguaje de alto nivel es un lenguaje mucho más entendible para el ser humano (human readable) en cuanto al significado de las instrucciones que se desean realizar. Su sintaxis es mucho más compleja que el código ensamblador y sus directivas son mucho más diversas y complejas. No tiene el mismo enfoque de programación que el assembly, sus operaciones no están diseñadas para conocer si se trabaja sobre registros o sobre la memoria. Implementa funciones y

subrutinas mucho más complejas. No está orientado a conocer el set de instrucciones del procesador a usar, de hecho, no es necesario conocer el procesador. Requiere de compiladores para traducir el código a código de máquina. etc.

3) Subrutinas: Son procedimientos que pueden ser definidos en una sección aparte del código principal, y que son usados con el propósito de "ahorrar código". Existen tareas que pueden ser repetitivas en la ejecución de un programa, por lo que en el momento que necesiten ser ejecutadas, sólo basta hacer un llamado a tal rutina y la ejecución del programa rompe su esquema secuencial (linea por linea), y salta a la rutina llamada, ejecutando la tarea deseada, tantas veces como sea necesario, y regresar nuevamente a la ejecución del programa principal.

4) Interrupción: Es una señal de ruptura de la ejecución normal de un programa, originada por un evento interno o externo al procesador que necesita ser atendido inmediatamente, antes de continuar con la ejecución principal del código. Las interrupciones se asocian a la ejecución de un segmento de código que resuelve el evento que originó la interrupción y luego retorna a la ejecución secuencial normal. Comúnmente denominados "exception handlers".

5) "Hello World" en assembly: Código que imprime en consoloa "Hello World", usando la directiva system call:

Listing 1: Codigo inicial

```
.data     msg:    .asciiz "Hello World"
.text
main:    li $v0, 4
 # syscall 4 (print_str)
         la $a0, msg    # argument: string
         syscall        # print the string
```

D. Aplicación del lenguaje ensamblador en SPIM

1) Algoritmo de la multiplicación: El algoritmo que se diseñó para la multiplicación es:

Listing 2: Codigo inicial

```
##########################################
# ALGORITMO DE LA MULTIPLICACION POR SUMAS
# SUCESIVAS
# SIGNED INTEGER
# MAX VALUE: (2^31) - 1 =  2.147'483.647
# MIN VALUE: (-2^31)    = -2.147'483.648

.data

multiplicando: .asciiz "ingrese el multi
plicando\n";
multiplicador: .asciiz "ingrese el multi
picador\n";
producto: .asciiz "el producto es: \n";
endline: .asciiz "\n";

.text

init:

        add $a0,$zero,$zero
        add $v0,$zero,$zero
        add $v1,$zero,$zero
        add $t0,$zero,$zero
        add $s0,$zero,$zero
        add $s1,$zero,$zero
        add $s2,$zero,$zero

main:

##########################################
#### Carga inicial de operandos ##########
##########################################

la $a0, multiplicando  # carga en a0 la
# direccion del label "multiplicando"
li $v0, 4              # carga en v0
# el inmediato de 32 bits
syscall                # llamada al
# sistema No 4: print string: passes a
# pointer to a null-terminated string

li $v0,5               # carga en v0 el
# inmediato de 32 bits
syscall                # llamada al siste
# ma No 5: read int, read float, and read
# double read an entire line of input up
# to and including a newline. El valor
# leido lo almacena en v0

move $s0,$v0           # pasa a s0 el
# valor leido

la $a0, multiplicador  # carga en a0 la
# direccion del label "divisor"
li $v0, 4              # carga en v0 el
# inmediato de 32 bits
syscall
 # llamada al sistema
# No 4: print string: passes a pointer to
# a null-terminated string

li $v0,5
 # carga en v0 el in
# mediato de 32 bits
syscall                # llamada al siste
# ma No 5: read int, read float, and read
# double read an entire line of input up
# to and
                       # including a new
# line. El valor leido lo almacena en v0

move $s1,$v0
 # pasa a s1 el valor
# leido

##########################################
```

```
#### Algoritmo por sumas sucesivas #######
##########################################

#--- Verificacion de operandos en cero---#

        beq $s0,$zero,cero
        beq $s1,$zero,cero

#------- Sumas sucesivas --------------------#

        move $s2,$s0
 # mueve
# el multiplicando a s3, donde se acumula
# el resultado

loop:

add  $s1,$s1,-1
 # resta 1 al mul
# tiplicador
slti $t0,$s1,1
 # compara si el
# multiplicador ahora esta en cero
bne  $t0,$zero,resultado
 # si t0 = 1, es
# decir, si el multiplicador es igual a ce
# ro, que salte

add  $s2,$s2,$s0
 # acumula en el
# registro de respuesta las sumas sucesivas
j loop

#-------------------------------------------#

cero:

        move $s2,$zero
        j resultado

#-------- Impresion de resultado ----------#

resultado:

        la $a0, producto        # carga en
# a0 la direccion del label "multiplicando"
        li $v0, 4               # carga en
# v0 el inmediato de 32 bits
        syscall                 # llamada
# al sistema No 4: print string: passes a
# pointer to a null-terminated string

        move $a0,$s2            # mueve
# el resultado a a0
        li $v0, 1               # Llamada
# al sistema No 1: print int: passes an
#integer and prints it on the console.
        syscall

        la $a0, endline         # carga en
# a0 la direccion del label "endline"
        li $v0, 4               # carga en
# v0 el inmediato de 32 bits
        syscall                 # llamada
# al sistema No 4: print string: passes a
# pointer to a null-terminated string

        j main                  # repeti
# cion infinita del algoritmo
```

VI. Conclusiones

- El lenguaje ensamblador es de gran ayuda a la hora de conocer cuales son los pasos que el procesador ejecuta para completar cierta tares. El simulador QTSPIM provee de herramientas de depuración como la ejecución paso a paso, que permiten identificar claramente los errores de programas y algoritmos implementados. Sus múltiples ventanas de visualización de memoria y registros fueron la solución para seguir la pista del flujo de información desde la memoria hacia los registros, y de estos hacia las unidades de ejecución y reescritura.
- Una mejor comprensión del proceso de creación del archivo de memoria que se ha de guardar en la ROM de la FPGA, permite comprender con mayor claridad las diferentes secciones que se pueden definir con las directrices del código assembly, tales como .text, .data, .word, etc, y las características de la información y datos que van en cada sección.

Referencias

[1] Patterson, David & Hennessy John *"'Computer Organization And Design - The Hardware-Software Interface"'*. Kindle Edition, Fourth Edition, 2006.

[2] http://www.latticesemi.com/products/intellectualproperty/ipcores/mico32/index.cfm

[3] http://www.ohwr.org/documents/68

[4] http://www.linuxencaja.net/wiki/Arquitectura_LM32_JPRR_%28261744%29