Guía Laboratorio Número 1

Arquitectura de Computadoras 25/04/2011

Repaso lenguaje ensamblador

Escribir códigos en lenguaje ensamblador es bastante sencillo, por lo que no es necesario el uso de un editor especial, pudiendo utilizarse cualquier editor de texto de su preferencia (como el Bloc de Notas en sistemas operativos Windows, por ejemplo). La diferencia con un archivo de texto común es que su extensión debe ser `.asm`.

Los códigos en lenguaje ensamblador suelen incluir ciertas "instrucciones especiales", denominadas ***directivas del ensamblador***, cuya función es indicar al programa ensamblador la manera en la que debe considerar a los datos o cómo ensamblar el código. En el ensamblador del MIPS, las directivas comienzan siempre con un punto.

Hay dos directivas cuyo uso es prácticamente obligatorio, que son las directivas `.data` y `.text`. La primera indica el ensamblador que todo lo que sigue a continuación debe ser considerado como datos y no como código, y la segunda es la inversa. Otras directivas muy usadas son aquellas que sirven para indicar cómo deben ser manipulados los datos. Por ejemplo, la directiva `.byte` indica que cada variable que sigue a continuación debe ocupar exactamente 8 bits en memoria. De manera similar, existen las directivas `.half` (16 bits) y `.word` (32 bits). Puede utilizarse la directiva .space para reservar una determinada cantidad de bytes en memoria. También existe la directiva `.asciiz` que considera lo que sigue a continuación como una cadena de texto ASCII terminada en cero. Otra directiva es la directiva `.globl`, que sirve para definir variables globales.

Otra herramienta muy útil del lenguaje ensamblador es la posibilidad de utilizar ***etiquetas***. Para el MIPS se considera etiqueta toda cadena de texto que comience en el primer carácter de una línea, que no comience con números y que finalice con el símbolo de dos puntos. Además, siempre es posible insertar ***comentarios*** dentro del código, que en este caso son siempre de una sola línea y se insertan a partir del símbolo de numeral (#).

Una mínima restricción que deben tener los códigos en lenguaje ensamblador del MIPS es que deben comenzar con una etiqueta `main`, y dicha etiqueta debe ser definida como una variable global. Esto sucede porque la última instrucción que ejecuta el simulador al cargar un programa es `jal main`.

Como ejemplo, mostramos a continuación el código en ensamblador del MIPS de un programa muy sencillo el cual suma dos números almacenados en memoria y guarda el resultado en memoria:

```
        .data
num_a:  .word 2
num_b:  .word 3
resul:  .word 0

        .text
        .globl main

main:   lw $t0,num_a
        lw $t1,num_b
        add $t2,$t0,$t1
        sw $t2,resul
fin:    jr $ra
```

Descarga e Instalación del simulador QtSpim

El simulador utilizado es el QtSpim, el mismo puede ser descargado desde el website de SPIM, (http://sourceforge.net/projects/spimsimulator/files/). Se recomienda utilizar la versión de QtSpim 9.0.1, que es con la cual se realizaron y se probaron todos los códigos del Laboratorio.

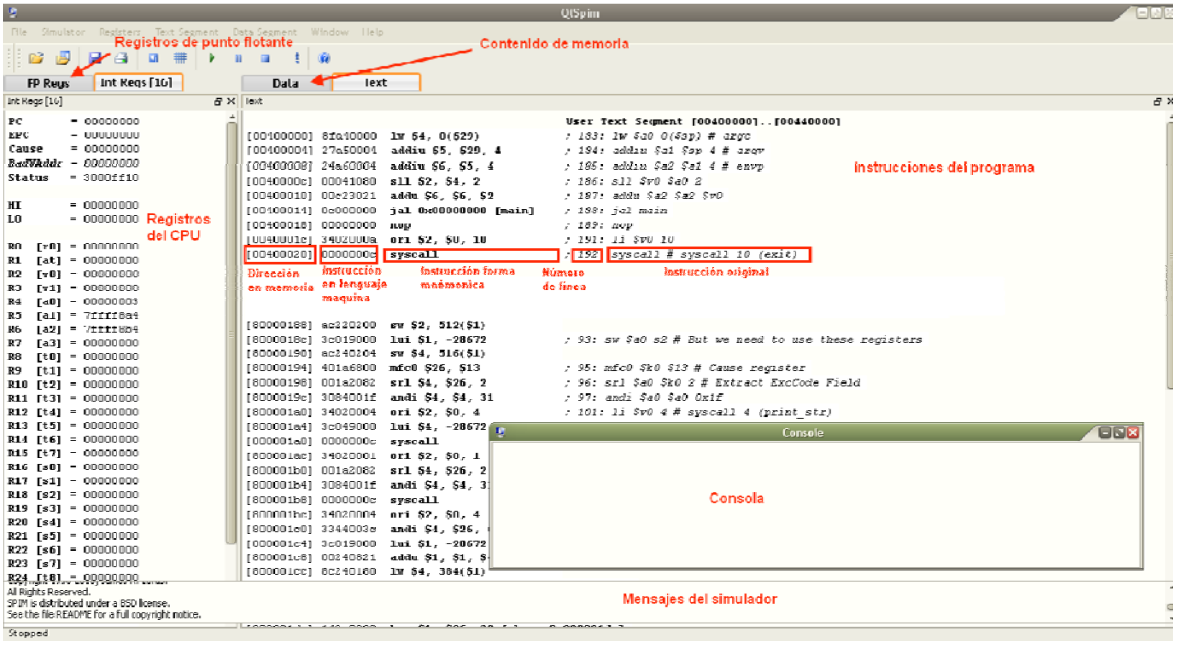
Lo primero que hay que hacer es instalar el simulador, y para ello hay que seguir los siguientes pasos:

1. Descargar el archivo correspondiente del sitio web (aprox. 20 MB en Windows, 1 MB en Linux, y 18 MB en Mac).
2. Descomprimir el archivo en cualquier directorio.
3. Ejecutar el archivo `setup.exe` (se les puede solicitar la instalación del Microsoft .NET Framework 4 Client Profile).

Introducción al simulador QtSpim

En esta sección los estudiantes podrán familiarizarse con el simulador QtSpim y poder cargar, ejecutar y hacer debugging de códigos en ensamblador.

Cuando se inicia el simulador, se abren dos ventanas: una principal y una consola. La ventana principal está dividida en tres paneles:



- El panel izquierdo muestra los valores de ***todos*** los registros del CPU. Separados en dos pestañas, los registros de punto flotante (**FP Regs**) y los registros enteros (**Int Regs**).

- El panel derecho muestra, en la pestaña **"Text"**, las instrucciones, tanto del programa del usuario como del sistema (que es cargado al iniciar el simulador). Cada instrucción es mostrada en una línea como la siguiente:

  ```
  [00400000] 8fa40000 lw $4, 0($29) ; 89: lw $a0, 0($sp)
  ```

  El primer número de la línea (entre corchetes) es la dirección de memoria de la instrucción, en hexadecimal. El segundo número representa la instrucción codificada en lenguaje de máquina, también en hexadecimal. El tercer campo representa la instrucción en su forma mnemónica. Lo que sigue a partir del punto y coma es la línea del código ensamblador que resulta en la instrucción. En el ejemplo, el número 89 es el número de línea dentro del archivo de código.
  A veces ocurre que no hay nada luego del punto y coma, esto significa que la instrucción fue producida por el SPIM como parte de la traducción de una pseudoinstrucción en más de una instrucción.

- En la pestaña **"Data"** del panel derecho, se muestran los datos cargados en memoria y en la pila.

- En el panel inferior es donde el simulador escribe mensajes, por ejemplo de error.

La otra ventana que se abre se denomina Consola y simula ser exactamente eso: una interfaz desde donde el programa pueda recibir y mostrar valores.

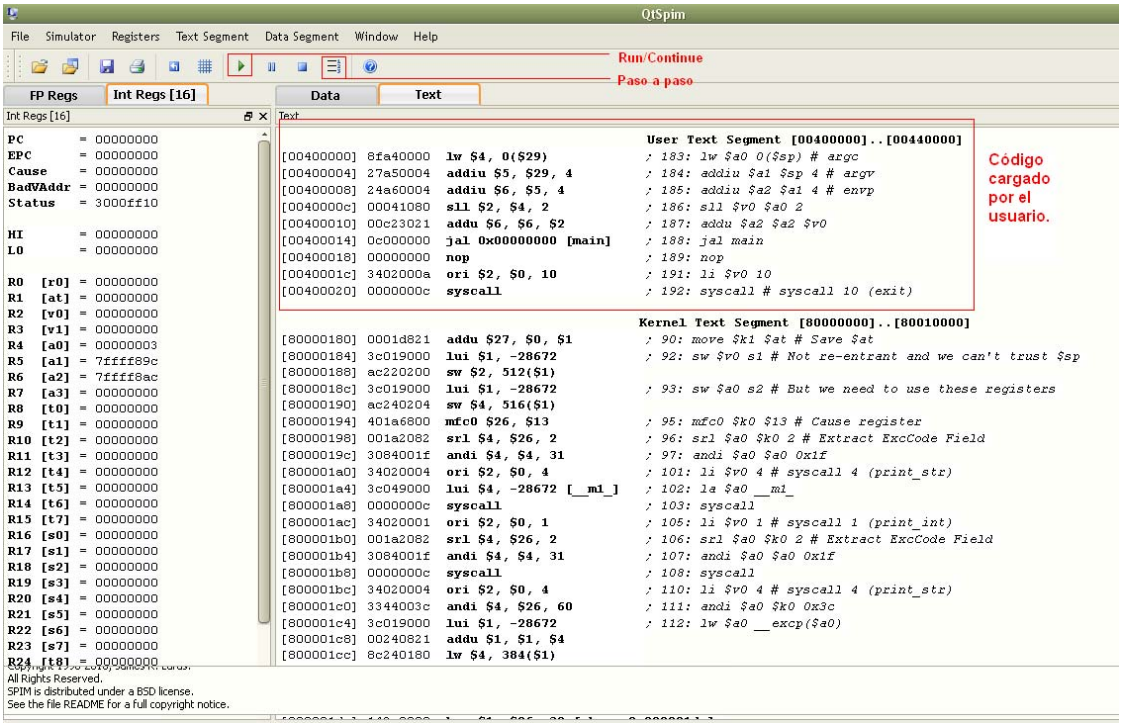
El siguiente paso es cargar un código en el simulador, para lo cual seleccionamos la opción *Abrir* ya sea desde la barra de herramientas o utilizando los iconos. El único detalle a tener en cuenta es que los archivos deben tener la extensión `.asm` o `.s`.

Al cargar un código, el simulador refresca los paneles para mostrar las instrucciones y datos que correspondan. Además al cargar el código el simulador realiza un control de sintaxis sobre el mismo, es

decir, que si el programa que se desea cargar tiene algún error de sintaxis en este punto el simulador indicará la presencia y ubicación del mismo.

Como se mencionó en la sección anterior, al cargar un programa se agregan ciertas líneas de código necesarias para inicializar el simulador las cuales terminan en una instrucción `jal main` a partir de la cual comienza a ejecutarse el código cargado por el usuario.

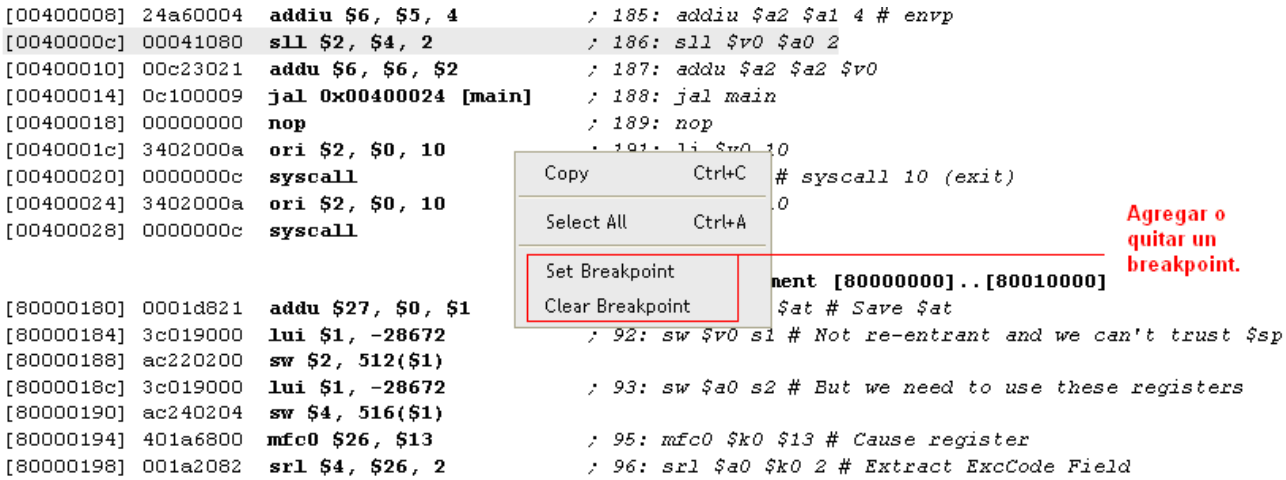
Para ejecutar el código, se pulsa el botón *Run/Continue* (resaltado en la siguiente captura).



Cuando un programa está corriendo, la pantalla ***no es actualizada*** para refrescar los cambios que están ocurriendo.

Una opción muy usada es la ejecución paso a paso (también puede usarse la tecla F10), resaltada en la imagen anterior. Con esta opción, el simulador ejecuta solamente una instrucción y se detiene, permitiendo visualizar los cambios producidos en los registros.

Otra opción muy interesante es el uso de breakpoints, que permiten que el simulador ejecute código continuamente hasta llegar a un punto determinado, donde se detiene. Para establecer un breakpoint, hay que posicionar el cursor en la instrucción donde queremos hacer la pausa, abrir el menú contextual con el click derecho del mouse y seleccionar *Set Breakpoint*. Con la opción *Clear Breakpoint*, se puede eliminar el breakpoint. Pueden establecerse todos los breakpoints que consideremos necesarios.



Si por algún motivo queremos realizar cambios en nuestro código, luego de modificar el archivo fuente y guardarlo usamos la opción *Reinitialize Simulator* (que se encuentra a la izquierda del botón Run) del simulador, la cual reinicializa todos los registros y vuelve a cargar el código. También es recomendable utilizar *Reinitialize Simulator* cuando terminada la ejecución del programa queremos ejecutar el mismo nuevamente.

Ejemplo

Para ilustrar el uso de la consola, mostraremos el mismo ejemplo anterior, pero usando además la instrucción `syscall` (llamada al sistema):

```
# Este programa suma dos números ubicados en memoria y guarda
#  el resultado en memoria, pero además lo muestra por pantalla.

      .data
num_a:      .word 2
num_b:      .word 3
result:     .word 0
cad0:       .asciiz "La suma de "
cad1:       .asciiz " y "
cad2:       .asciiz " es "
      .text
      .globl main
main:
      lw $t0,num_a      # Cargamos el valor de num_a en t0
      lw $t1,num_b      # Cargamos el valor de num_b en t1
      add $t2,$t0,$t1   # Guardamos la suma de t0 y t1 en t2
      sw $t2,result     # Almacenamos t2 en la dirección result

      li $v0,4          # Estas tres líneas de
      la $a0,cad0       # código imprimen la cadena
      syscall           # cad0
      li $v0,1
      move $a0,$t0
      syscall
      li $v0,4
      la $a0,cad1
      syscall
      li $v0,1          # Estas tres líneas de
      move $a0,$t1      # código imprimen el entero
      syscall           # almacenado en t1
      li $v0,4
      la $a0,cad2
      syscall
      li $v0,1
      move $a0,$t2
      syscall
Fin:  jr $ra            # Retorna al programa principal
```

Sugerimos guardar este código como `ejemplo.asm` y ejecutarlo paso a paso para entender su funcionamiento.

Manejo del Stack

A diferencia de las máquinas CISC (como el CPU08), las máquinas RISC (como el MIPS R2000) no poseen instrucciones especiales para el manejo del stack, sino que utilizamos uno de los registros de propósito general del procesador que apunta al último dato del stack, el $sp (stack pointer).

Es importante tener en cuenta que la pila crece desde direcciones altas de memoria a direcciones bajas, de este modo cuando queremos almacenar un nuevo valor en el stack (operación push) es necesario reservar primero la memoria que será utilizada decrementando el valor de $sp y luego almacenar allí la información.

En el siguiente ejemplo se almacena los 16 bits menos significativos del registro `$t0` en el stack:

```
addi $sp,$sp,-2              # Reservamos los 16 bits necesarios
sh $t0, 0($sp)               # Guardamos los dos bytes menos
                             # significativos de $t0 en el stack
```

Por último si queremos tomar un valor del stack (operación pop) es necesario leer el valor referenciado por el `$sp` y luego incrementarlo en la cantidad de bytes que posee el dato. De esta forma el puntero apunta ahora al siguiente dato, es decir, sacamos el dato leído del stack.

Las siguientes líneas corresponden a la lectura de un dato de 1 byte desde la pila al registro $a1:

```
lb $a1, 0($sp)               # Carga el último byte del stack
addi $sp,$sp,1               # Actualizamos el valor del puntero
```

Una de las funciones del stack es permitir el paso de parámetros entre funciones. De este modo las funciones no se ven afectadas por las posibles interrupciones que pudiesen afectar el valor de los registros que deberían utilizar para el pasaje de los parámetros. Además cuando se trabaja con funciones recursivas es fundamental el uso del stack para poder almacenar los distintos valores de salida de la función hasta que los mismos sean utilizados.

Llamadas al Sistema

Para realizar una llamada al sistema lo primero que debemos hacer es cargar el código del servicio que deseamos ejecutar en el registro `$v0`. De acuerdo al servicio ejecutado, si el mismo requiere argumentos deberán cargarse los mismos en los registros correspondientes antes de ejecutar la instrucción `syscall`. Una vez ejecutada esta instrucción, podrá leerse la información recibida de los registros especificados (para los servicios que retornan un valor).

Servicio	Propósito	Código	Argumentos	Resultados
print_int	Imprime un entero por consola.	1	$a0 = entero	
print_float	Imprime un número real por consola.	2	$f12 = flotante	
print_string	Imprime una cadena por consola.	4	$a0 = cadena	
read_int	Lee de la consola un entero.	5		entero en $v0
read_float	Lee de la consola un número real.	6		real en $f0
read_string	Lee una cadena de la consola.	8	$a0 = buffer $a1 = longitud	cadena a partir de buffer
exit	Salir del programa	10		