

Guía para el Trabajo de Laboratorio Número 1

Arquitectura de Computadoras

11/04/2022

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` o `.s`.

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 RISC-V, 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 al ensamblador que todo lo que sigue a continuación debe ser considerado como datos y no como código, y la segunda es la inversa. Otra directiva de uso obligado es `.globl`, que sirve para definir símbolos globales que pueden ser utilizados desde otros archivos.

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 a continuación debe ocupar exactamente 8 bits en memoria. De manera similar, existen las directivas `.half` (16 bits) y `.word` (32 bits). También existe la directiva `.ascii` que considera lo que sigue a continuación como una cadena de texto ASCII terminada en cero. El ensamblador de RISC-V no soporta la directiva `.space` para reservar una determinada cantidad de bytes en memoria, por lo que las reservas de espacio deberán hacerse utilizando las directivas anteriores, las cuales originalmente se deben utilizar para declarar variables con un valor inicial.

Otra herramienta muy útil del lenguaje ensamblador es la posibilidad de utilizar etiquetas. Para el RISC-V 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 RISC-V 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`. Para terminar el programa correctamente se pide un servicio al sistema operativo.

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

```

        .data
numero_a: .word 2
numero_b: .word 3
resultado: .word 0

        .text
        .globl main
main:    lw t0,numero_a      # Se carga el primer operando desde memoria
        lw t1,numero_b      # Se carga el primer operando desde memoria
        add t0, t0, t1       # Se calcula el resultado como la suma de los operandos
        la t1,resultado     # Se carga la dirección para almacenar resultado
        sw t0,0(t1)         # Se almacena el resultado utilizando el puntero
fin:     li a0,10            # Selección del servicio terminar el programa sin error
        ecall               # Termina el programa y retorna al sistema operativo

```

Introducción al simulador Venus

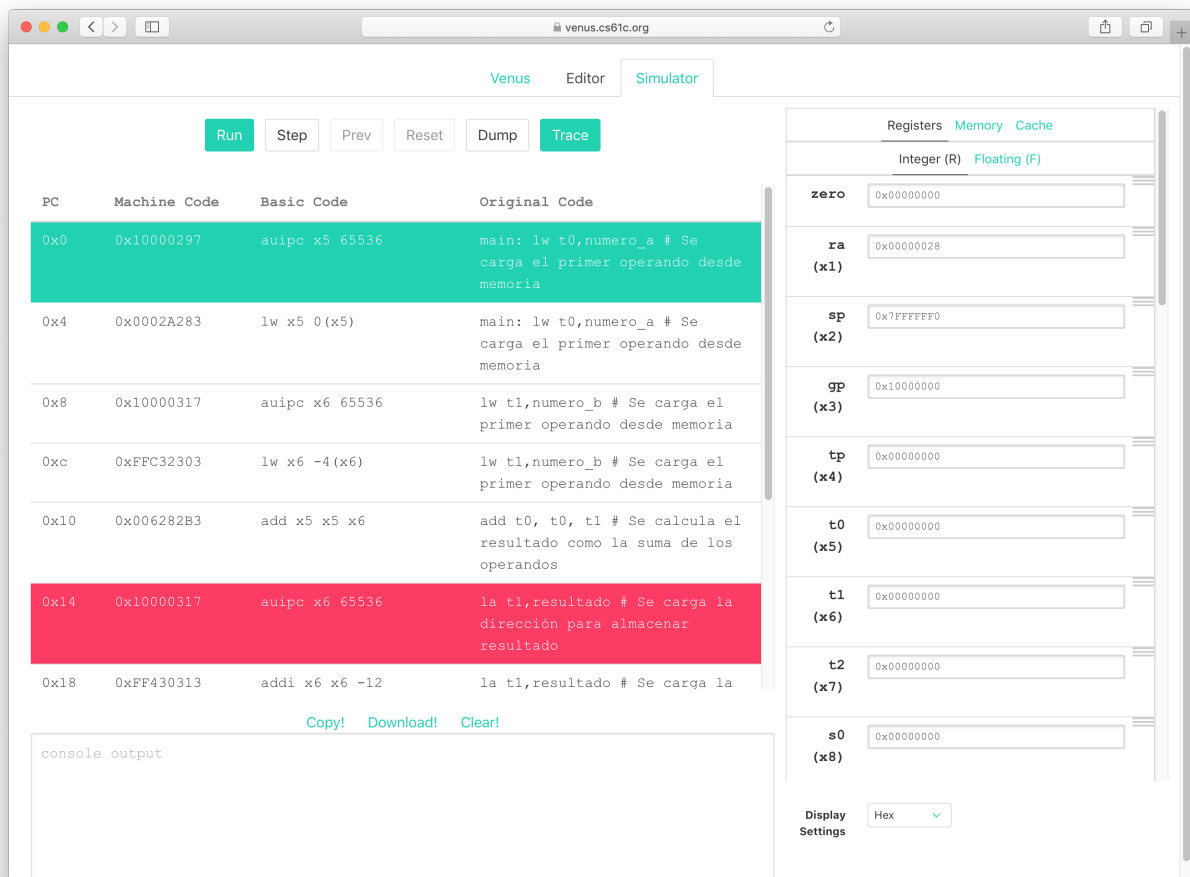
El simulador que utilizaremos es el Venus, el mismo puede ser utilizado directamente en un navegador WEB accediendo a la dirección **<https://venus.cs61c.org>**. Cuando ingresamos al simulador por primera vez accedemos a la ventana principal, y dentro de ella a la solapa del editor de código. En esta pantalla se puede escribir el programa que queremos simular. El editor dispone de resaltado y verificación de sintaxis.



```

1      .data
2  numero_a: .word 2
3  numero_b: .word 3
4  resultado: .word 0
5
6      .text
7      .globl main
8  main: lw t0,numero_a    # Se carga el primer operando desde memoria
9        lw t1,numero_b    # Se carga el primer operando desde memoria
10       add t0, t0, t1     # Se calcula el resultado como la suma de los operandos
11       la t1,resultado    # Se carga la dirección para almacenar resultado
12       sw t0,0(t1)        # Se almacena el resultado utilizando el puntero
13  fin:  li a0,10          # Selección del servicio terminar el programa sin error
14       ecall              # Termina el programa y retorna al sistema operativo
  
```

Una vez completo el programa en el editor debemos cambiar la vista a la solapa del simulador y utilizar el botón *Assemble & Simulate from Editor* para compilar el programa del editor y cargarlo en el simulador. Si el programa no tiene errores entonces se puede empezar la simulación.



Run Step Prev Reset Dump Trace

PC	Machine Code	Basic Code	Original Code
0x0	0x10000297	auipc x5 65536	main: lw t0,numero_a # Se carga el primer operando desde memoria
0x4	0x0002A283	lw x5 0(x5)	main: lw t0,numero_a # Se carga el primer operando desde memoria
0x8	0x10000317	auipc x6 65536	lw t1,numero_b # Se carga el primer operando desde memoria
0xc	0xFFC32303	lw x6 -4(x6)	lw t1,numero_b # Se carga el primer operando desde memoria
0x10	0x006282B3	add x5 x5 x6	add t0, t0, t1 # Se calcula el resultado como la suma de los operandos
0x14	0x10000317	auipc x6 65536	la t1,resultado # Se carga la dirección para almacenar resultado
0x18	0xFF430313	addi x6 x6 -12	la t1,resultado # Se carga la

Copy! Download! Clear!

console output

Registers Memory Cache

Integer (R) Floating (F)

zero 0x00000000

ra (x1) 0x00000028

sp (x2) 0x7FFFFFF0

gp (x3) 0x10000000

tp (x4) 0x00000000

t0 (x5) 0x00000000

t1 (x6) 0x00000000

t2 (x7) 0x00000000

s0 (x8) 0x00000000

Display Settings Hex

En esta ventana se puede ejecutar el programa en forma completa con el botón *Run* o ejecutarlo paso a paso utilizando el botón *Step*. Otra opción muy interesante es el uso de breakpoints, que permiten que el simulador ejecute código continuamente hasta llegar a un punto determinado, para esto solo es necesario hacer click en la línea de código correspondiente, lo cual se señaliza pintando la misma de color rojo. En esta ventana tenemos los siguientes elementos:

- El panel izquierdo muestra las instrucciones del programa del usuario. Cada instrucción es mostrada en una línea como la siguiente:

0x10 0x10000317 auipc x6 65535 la t1, resultado # Se carga la

El primer campo de la línea corresponde al número de instrucción. El segundo número representa la instrucción codificada en lenguaje de máquina, también en hexadecimal. El tercer campo representa la instrucción en su forma mnemónica. El cuarto campo corresponde al código original del programa. A veces ocurre que se repite la cuarta columna para dos filas consecutivas, esto significa que pseudoinstrucción se convirtió en más de una instrucción.

- En el panel de la izquierda abajo de la lista de instrucciones se encuentra un panel con la salida de la consola. El simulador incorpora un pequeño sistema operativo al cual el programa le puede pedir una serie de servicios para escribir en esta consola.
- El panel izquierdo derecho permite la inspección y modificación de los registros o de la memoria en función de la solapa seleccionada en la parte superior. En la parte inferior se puede seleccionar el formato para visualizar la información utilizando la lista desplegable *Display Settings*

En la lista de los registros del procesador se puede ver el valor actual de los mismos y cambiarlos simplemente escribiendo un nuevo valor. Lo mismo sucede con el contenido de la memoria cuando cambiamos de solapa en el panel. Para buscar en una sección de memoria se puede utilizar el control *Jump to* en la zona inferior del panel. En la figura se pueden ver coloreadas las dos variables que se suman en el programa y el resultado de la suma.

The screenshot shows the Venus CS61C simulator interface. The top bar includes tabs for 'Venus', 'Editor', and 'Simulator'. Below the tabs are buttons for 'Run', 'Step', 'Prev', 'Reset', 'Dump', and 'Trace'. The main panel displays assembly instructions with their addresses, hexadecimal values, mnemonics, and comments. The instruction at address 0x14 is highlighted in red. To the right, there are three tabs: 'Registers', 'Memory', and 'Cache'. The 'Registers' tab is active, showing a table of registers with their addresses and values. The 'Memory' tab is also visible, showing a table of memory addresses and values. At the bottom, there is a 'Jump to' section with a dropdown menu and buttons for 'Up' and 'Down'. Below that is a 'Display Settings' section with a dropdown menu set to 'Hex'.

Address	+0	+1	+2	+3
0x10000018	00	00	00	00
0x10000014	00	00	00	00
0x10000010	00	00	00	00
0x1000000C	00	00	00	00
0x10000008	05	00	00	00
0x10000004	03	00	00	00
0x10000000	02	00	00	00
0x0FFFFFFC	00	00	00	00
0x0FFFFFF8	00	00	00	00
0x0FFFFFF4	00	00	00	00
0x0FFFFFF0	00	00	00	00
0x0FFFFFFEC	00	00	00	00
0x0FFFFFFE8	00	00	00	00

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: li a0,4      # Selección del servicio: mostrar una cadena por pantalla
      la a1,cad0   # Apuntamos al inicio de la cadena a imprimir
      ecall        # Pedimos el servicio, la cadena se muestra en la pantalla

      lw s0,num_a  # Cargamos el primer operando
      li a0,1      # Selección del servicio: mostrar un entero por pantalla
      mv a1,s0     # Cargamos el valor que queremos mostrar
      ecall        # Pedimos el servicio, el entero se muestra en la pantalla

      li a0,4      # Selección del servicio: mostrar una cadena por pantalla
      la a1,cad1   # Apuntamos al inicio de la cadena a imprimir
      ecall        # Pedimos el servicio, la cadena se muestra en la pantalla

      lw s1,num_b  # Cargamos el segundo operando
      li a0,1      # Selección del servicio: mostrar un entero por pantalla
      mv a1,s1     # Cargamos el valor que queremos mostrar
      ecall        # Pedimos el servicio, el entero se muestra en la pantalla

      li a0,4      # Selección del servicio: mostrar una cadena por pantalla
      la a1,cad2   # Apuntamos al inicio de la cadena a imprimir
      ecall        # Pedimos el servicio, la cadena se muestra en la pantalla

      add s2,s0,s1 # Sumamos ambos operandos
      la t0,result # Cargamos la dirección para almacenar resultado
      sw s2,0(t0)  # Almacenamos el resultado
      li a0,1      # Selección del servicio: mostrar un entero por pantalla
      mv a1,s2     # Cargamos el valor que queremos mostrar
      ecall        # Pedimos el servicio, el entero se muestra en la pantalla

fin:  li a0,10     # Selección del servicio terminar el programa sin error
      ecall        # Termina el programa y retorna al sistema operativo
```

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

Manejo de la pila

Las máquinas RISC puras como el RISC-V no poseen instrucciones especiales para el manejo del stack, sino que utilizan 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. El siguiente ejemplo corresponde 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 operativo

Las llamadas al sistema operativo permiten pedirles servicios al mismo tales como interacción con el usuario mediante la consola o la lectura y escritura de archivos. Para realizar una llamada al sistema lo primero que debemos hacer es cargar el código del servicio que deseamos ejecutar en el registro **a0**. De acuerdo al servicio deseado, si el mismo requiere argumentos deberán cargarse los mismos en los registros correspondientes antes de ejecutar la instrucción **ecall**. Una vez ejecutada esta instrucción, podrá leerse la información recibida de los registros especificados (para los servicios que retornan un valor). La lista completa de los servicios disponibles se puede consultar en la Wiki del simulador en la dirección <https://github.com/ThaumicMekanism/venus/wiki>.

Servicio	Propósito	Código	Argumentos	Resultados
print_int	Imprime un entero por consola.	1	a1 = entero	
print_string	Imprime una cadena por consola.	4	a1 = cadena	
exit	Salir del programa	10		
print_character	Imprime un caracter por consola	11	a1 = caracter	