Escuela Politécnica Superior

Práctica 1:

Estructura de Computadores

Introducción a RIPES

UNIVERSIDAD NEBRIJA

Índice/Tabla de contenidos

1. METODOLOGÍA Y FORMATO DE LA ACTIVIDAD

Esta actividad está basada en los contenidos estudiados en los temas 1, 2 y 3 de la asignatura.

Se proponen varios ejercicios que deberán ser completados por equipos de hasta tres alumnos en la sesión 1 de prácticas de laboratorio.

- La entrega se realizará a través del campus virtual mediante un único archivo ZIP, que albergue todos los ejercicios, antes del 2 de marzo de 2023.
- La evaluación se realizará a través del campus virtual, teniendo en cuenta:
    - La corrección del programa en base a las especificaciones descritas.
    - La calidad del código.
    - El cumplimiento de las normas de estilo.

2. DESARROLLO DE LA ACTIVIDAD PRÁCTICA

2.1. Objetivo

En esta práctica se aprenderán los aspectos generales de la **programación en ensamblador**, así como el uso de las instrucciones aritméticas básicas y de acceso a memoria.

Las presentes prácticas están diseñadas para utilizar el lenguaje ensamblador de RISC-V, sobre un entorno de simulación denominado RIPES.

2.2. El simulador

El simulador utilizado tiene tres ventanas: "*Editor*", "*Processor*" y "*Memory*".

La ventana "*Editor*" es donde se escribe el programa en ensamblador (pestaña "*Source code*"). También es posible cargar un fichero de texto con el programa. En caso de programar dentro del editor de RIPES, se recomienda guardar con frecuencia el código. En la pestaña "*Source code*" se puede ver el texto fuente con distintos códigos de colores. Por ejemplo, los comentarios y etiquetas se marcan en ámbar, los registros en rojo, las instrucciones en azul, las constantes en verde, etc. En caso de un error sintáctico, el compilador marcará la línea en rojo con el objetivo de que sea corregida por el programador. Por último, en la pestaña "*Disassembled code*" se puede ver el código desensamblado del programa.

A continuación, se muestra la pantalla principal de RIPES, tras cargar un archivo con código fuente en ensamblador:

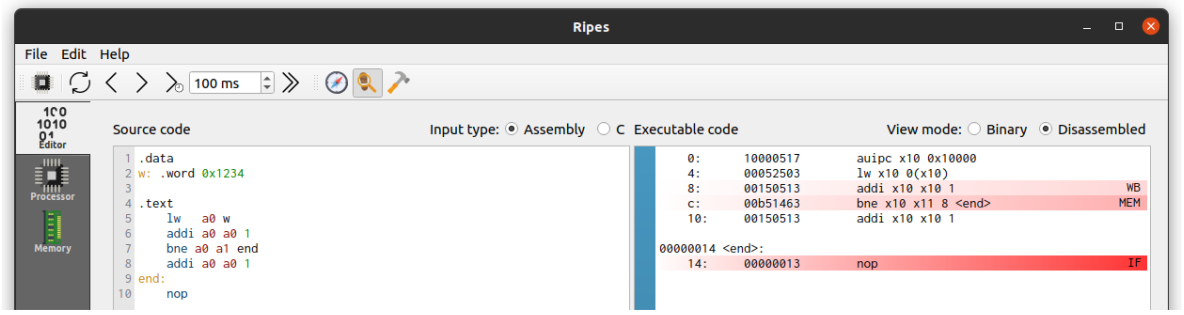

Ilustración 1: Vista por defecto del editor

En la barra de herramientas se pueden ver los siguientes iconos:

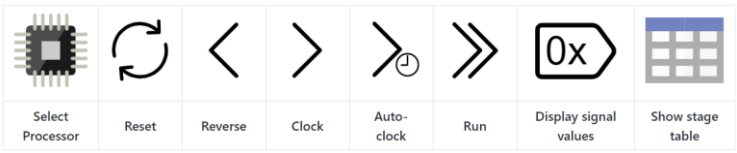

Ilustración 2: Iconos de la barra de herramientas

Dichos iconos permiten (de izquierda a derecha): cambiar el procesador a simular, relanzar la simulación, volver un paso atrás la simulación, avanzar un paso la simulación, lanzar la simulación un determinado tiempo, ejecutar completamente la simulación, mostrar los valores de las señales internas del procesador y mostrar una tabla con las etapas.

El programa ensamblador se puede ejecutar desde cualquiera de las 3 ventanas (“*Editor*”, “*Processor*” y “*Memory*”). Si se lanza desde la ventana editor se mostrará la instrucción ejecutada en cada etapa del procesador. Si se lanza desde la ventana “*Processor*”, se puede ver el cauce de ejecución en un procesador de cinco etapas RISC-V, permitiendo comprobar cómo las instrucciones van fluyendo a través de cada una de dichas etapas (para esto es recomendable ejecutar el programa en modo paso a paso). En la parte derecha se ve el contenido de los registros (1) y así como las instrucciones que se ejecutan (2). En la parte de abajo (“Output”) se puede ver los mensajes y resultados que muestra el programa (4) y algunos parámetros estadísticos (3).

Finalmente, en la ventana “*Memory*” se puede ver todo el contenido direccionable del procesador (parte izquierda) y se puede acceder a un simulador de memorias caché que permite obtener varios parámetros y gráficas con los accesos a memoria durante la ejecución de un determinado programa ensamblador.

En las siguientes ilustraciones pueden observarse cada una de las pantallas:

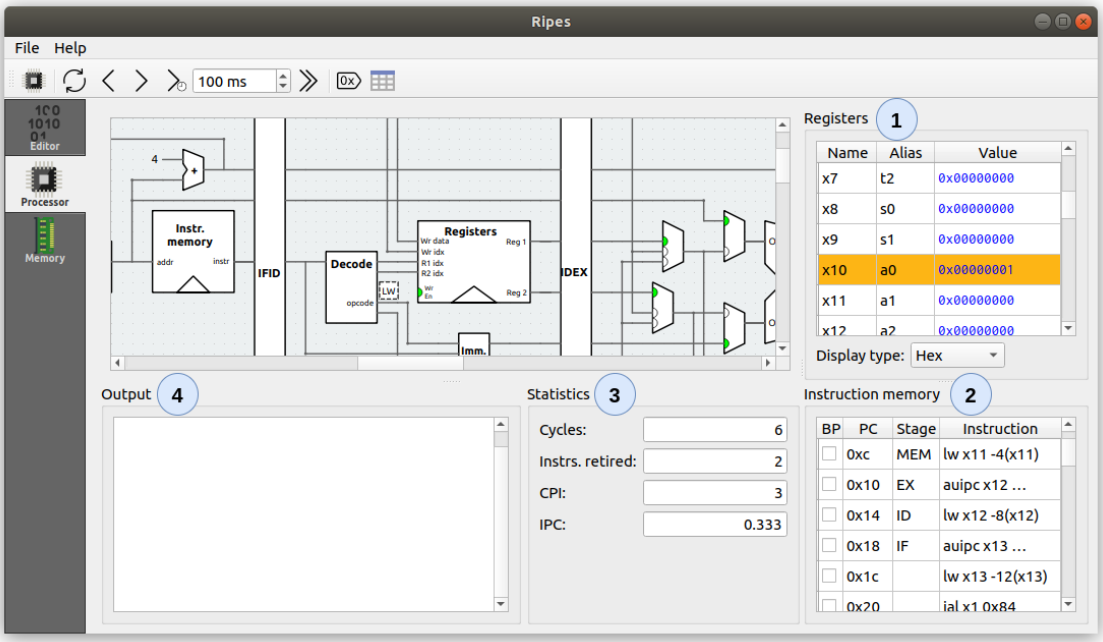

Ilustración 3: Vista por defecto del entorno de ejecución

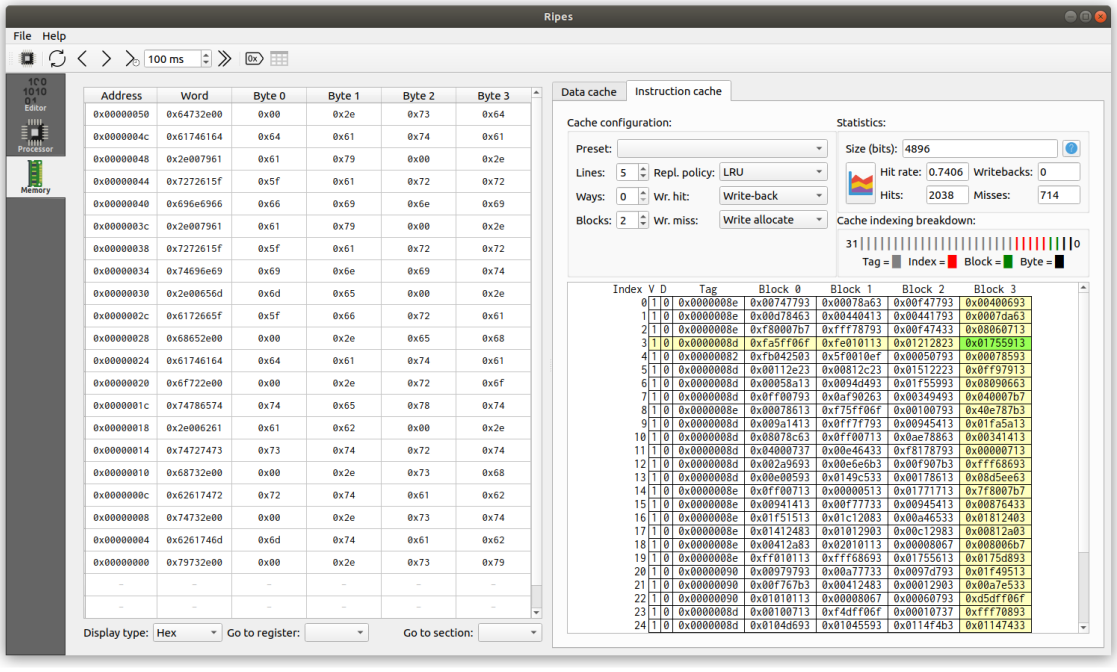

Ilustración 4: Vista por defecto de la memoria

2.3. Memoria

Una característica importante es que RIPES trabaja con un ancho de palabra de 32 bits, por tanto, en una palabra de memoria caben 4 bytes. Cada uno de los bytes puede ser direccionado de manera independiente. Como la mayoría de las arquitecturas RISC, RISC-V es una arquitectura "*little endian*", lo cual significa que la dirección de cada palabra corresponde a la dirección de su byte menos significativo.

El hecho de que el ancho sea de 32 bits hace que las operaciones de lectura ("*load*") y escritura ("*store*") en memoria se referencien con respecto a la palabra, usándose las instrucciones "**lw**" ("*load word*") y "**sw**" ("*store word*"). De esta manera, no se usarán las lecturas y cargas de 64 bits ("*double word*"), es decir "**ld**" y "**sd**".

Cuando se trabaje con caracteres ASCII, hay que tener en cuenta que cada uno de estos caracteres ocupa un byte. Por lo tanto, para leer y escribir caracteres de memoria se usarán las instrucciones "**lb**" ("*load byte*") y "**sb**" ("*store byte*").

Los datos se almacenan en memoria en tres zonas, atendiendo al tipo de cada uno:

- Zona de instrucciones ("*Text*"). Es el área donde el sistema almacena las instrucciones del programa a ejecutar. Empieza en la dirección 0x00000000 y va creciendo hacia direcciones más altas.

- Zona de datos estáticos ("*Static data*"). Es el área donde se almacenan los datos estáticos del programa (el equivalente a las constantes y variables). Empieza en la dirección 0x10000000 y va creciendo hacia direcciones más altas

- Zona de pila ("*Stack*"). Es el área donde se guardan datos dinámicos, indexados mediante el puntero de pila "**sp**". Empieza en la dirección 0x7ffffff0 y va decreciendo hacia direcciones más bajas.

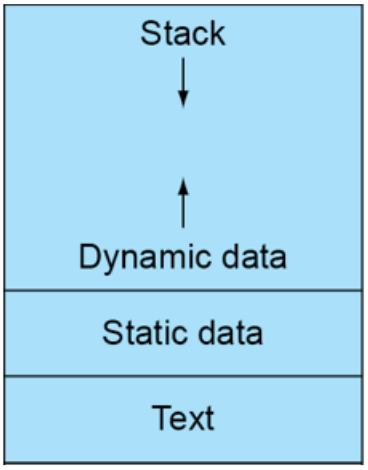

Ilustración 5: Estructura de la memoria

Los programas empiezan con la declaración de la zona de datos estáticos, mediante el uso de la expresión "**.data**".

Las etiquetas son símbolos que se asocian a la dirección de una determinada palabra de memoria, y se representan por cadenas alfanuméricas acabadas en "**:**", por ejemplo "**data:**".

Para reservar una palabra de memoria y guardar un valor entero, se usa "**word**", seguido por el valor. Para reservar una cadena de caracteres, se usa "**string**", seguido por su valor entre comillas.

```
1 .data
2 data:    .word 5
3 str1:    .string "This is an example"
```

Ilustración 6: Ejemplo de reserva de datos

En este ejemplo, se ha reservado una palabra de memoria y se ha almacenado en ella el número 5. Esta palabra se referenciará con el nombre "*data*" (en otras palabras, "*data*" contendrá un puntero con la dirección de esa palabra).

De la misma manera, se ha reservado espacio en la memoria para almacenar la cadena "*This is an example*" (donde cada carácter ASCII ocupa un byte). El identificador, o etiqueta, "*str1*" será un puntero con la dirección al primer carácter de la cadena.

A continuación de la sección de datos, se encuentra la sección de instrucciones, identificada con "**.text**". A partir de este punto, se escriben todas las instrucciones del programa. En esta sección se pueden escribir tanto instrucciones como etiquetas.

```
1 .data
2 data:    .word 5
3 str1:    .string "This is an example"
4
5 .text
6 begin:
7 lw a0,data      # Load data value (5)
8 la a1,str1      # Load string address
9
10 end:
11 li a7,10
12 ecall
```

Ilustración 7: Ejemplo de una zona de datos e instrucciones

En este programa de ejemplo, se han usado dos instrucciones.

La primera instrucción, "**lw a0, data**" ("*load word*"), carga el contenido de la palabra referenciada por "*data*" en el registro a0. Es decir, carga el valor 5. En cambio, la segunda instrucción, "**la a1, str1**" ("*load address*"), carga la dirección de la cadena "*str1*" en el registro a1. En este caso, está dirección es 0x10000004, como se puede ver en el contenido del registro *a1*, en la sección "*Memory*" del simulador.

Las etiquetas "**begin:**" y "**end:**" no son obligatorias, pero son recomendables por cuestiones de claridad. Las dos instrucciones que aparecen después de la etiqueta "*end:*" se utiliza para finalizar el programa y detener el simulador. Es similar a usar el "*return 0;*" en lenguaje C.

2.4. Normas de estilo

Los programas en ensamblador suelen tener poca legibilidad. Por ello, es preciso mejorar su claridad con un buen estilo de escritura y organización del código. El uso de la sangría cuando sea preciso, dejar líneas en blanco, usar espacios, y en general utilizar cualquier recurso que permita que el código sea fácil de seguir e interpretar, ayuda a mejorarla claridad y documentación del programa.

Especial atención merecen los **comentarios**. Es preciso añadirlos a cada parte del código para indicar qué es lo que se está realizando en cada momento, así como identificar las secciones y aclarar las instrucciones que sean pertinentes. Los comentarios en el entorno se añaden **con el símbolo "#"**. Todos los caracteres desde ese símbolo hasta el final de línea son interpretados como un comentario, y por tanto ignorados por el compilador. El entorno no reconoce caracteres especiales como la letra "ñ" o los acentos. Por este motivo, y para aumentar la portabilidad del código, las etiquetas y comentarios de los programas han de ser escritos en inglés.

Otro tema importante es usar los registros con sus nombres significativos. Aunque el sistema reconoce los registros de forma secuencial, de x0 a x31, es preferible utilizar sus alias. Por ejemplo, usar "**sp**" ("*stack pointer*") en lugar de x2, a0 en lugar de x10 o s5 en lugar de x21. En general, los registros "a" se usan para pasar argumentos a funciones, los registros "t" son para valores temporales y los registros "s" son registros guardados en las llamadas a procedimientos.

2.5. Pseudoinstrucciones

Aparte de las instrucciones estándar del repertorio del RISC-V, el compilador entiende otras, llamadas "*pseudoinstrucciones*". Estas instrucciones se usan para simplificar la escritura del código, bien sea agrupando un conjunto de instrucciones o "redefiniendo" el nombre de una de ellas. Como es obvio, es el compilador quien se encarga de traducirlas a instrucciones reales del ISA.

Por ejemplo, en RISC-V no existe la instrucción de movimiento entre registros, pero sí la pseudoinstrucción "**mv**". De esta manera, "*mv a1, t1*" copia el contenido del registro t1 en el registro a1. Esto se traduce como "*add a1, t1, zero*", que suma t1 con cero y guarda el resultado en a1. Como se ve, el resultado es el mismo, pero la primera forma es más intuitiva que la segunda.

Aparte de "mv", otras pseudoinstrucciones son por ejemplo "**la**", "**li**", "**j**".

2.6. Impresión por pantalla

Se usa la instrucción "**ecall**" para diferentes usos: imprimir por pantalla una cadena de caracteres, un número o incluso leer de un archivo.

Para **imprimir un valor entero** hay que seguir tres pasos: cargar en el registro a0 el valor a imprimir, en a7 el valor 1 y llamar a la función "ecall".
Para **imprimir una cadena de caracteres** hay que seguir tres pasos: cargar en el registro a0 la dirección (puntero) de la cadena a imprimir, en a7 el valor 4 (que significa imprimir en pantalla) y llamar a la función "ecall".
Por último, para **imprimir el símbolo ASCII del byte** hay que seguir estos tres pasos: cargar en el registro a0 el valor a imprimir, en a7 el valor 11 y llamar a la función "ecall".

2.7. Ejemplo de instrucciones

INSTRUCCION	RESULTADO
`li s4, 5`	Carga en s4 el valor 5
`la a2, var`	Carga en a2 la dirección de la variable "var" reservada en memoria
`lw a0, var`	Carga en a0 el contenido de la variable "var" (en memoria).
`lw a0, 4(a1)`	Carga en a0 la palabra de memoria cuya dirección es a1+4
`lb a0, 4(a1)`	Carga en a0 el byte de memoria cuya dirección es a1+4
`sw a3, 8(a2)`	Almacena el valor de a3 en la palabra de memora cuya dirección es a2+8
`sb a3, 8(a2)`	Almacena el valor de a3 en el byte de memora cuya dirección es a2+8
`add a0, a1, a2`	Suma a1 más a2, y guarda el resultado en a0
`addi a0, a1, 6`	Suma a1 más 6, y guarda el resultado en a0
`sub a0, a1, a2`	Resta a1 menos a2, y guarda el resultado en a0
`addi a0, a1, -1`	Resta a1 menos 1, y guarda el resultado en a0
`mul a0, a1, a2`	Multiplica a1 por a2, y guarda el resultado en a0
`div a0, a1, a2`	Divide a1 entre a2, y guarda el cociente de la división en a0
`rem a0,a1,a2`	Divide a1 entre a2, y guarda el resto de la división en a0

3. ENTREGABLES

Se entregará un programa en ensamblador para cada uno de los ejercicios siguientes:

3.1. Impresión por pantalla

Ejercicio (1.A) : Mostrar por pantalla el texto "**39steps**"
Ejercicio (1.B) : Mostrar por pantalla (en líneas distintas) el texto "**39**
steps"

Para escribir en distinta línea, considerar que el código ASCII para nueva línea es el 10.

3.2. Operaciones aritméticas simples

Ejercicio (2) : Realizar un programa que realice las siguientes operaciones:

- Realizar la operación matemática **`F = (A+B) – (C+D)`**.
- Habrá que definir las variables A, B, C, D y F en memoria.
- Dar los valores iniciales A=5, B=3, C=2, D=2.
- Escribir el resultado de la operación en la variable F.
- Mostrar también el valor de F por pantalla.

3.3. Resto de la división

Ejercicio (3.A) : Definir en memoria dos variables enteras positivas, A y B (valores iniciales a elegir), y calcular el resto de dividir A entre B. Almacenar el resultado en la variable R y mostrar el resultado por pantalla. Para realizar este apartado, utilizar la instrucción "rem".

Ejercicio (3.B) : Repetir el punto anterior, pero esta vez sin utilizar la instrucción "rem". Para ello, utilizar la propiedad de la división de que el dividendo es el divisor multiplicado por el cociente más el resto.