FUNDAMENTOS DE COMPUTADORES
Práctica 6
Introducción al simulador PCSpim

OBJETIVOS

El objetivo principal de esta sesión de laboratorio es usar el simulador PCSpim por primera vez. A través de la arquitectura MIPS y de su lenguaje ensamblador se pondrán en práctica los conocimientos básicos sobre representación de las instrucciones y de los números enteros en un computador. Así mismo, estudiaremos con detalle las operaciones de multiplicación y división de números enteros y, finalmente, diseñaremos un pequeño programa en lenguaje en ensamblador para poner en práctica todos los aspectos tratados en la sesión de laboratorio.

INTRODUCCIÓN A LA ARQUITECTURA MIPS

PCSpim es una aplicación que permite simular la ejecución de programas escritos en el lenguaje ensamblador de los procesadores con arquitectura MIPS I (actualmente un subconjunto de MIPS32) desarrollada por la empresa MIPS Technologies (Ilustración 1). Podemos encontrar procesadores MIPS en muchos sistemas informáticos: consolas de videojuegos PlayStation de 1988 (R3000) y Nintendo G64 de 1992 (R4000), impresoras láser HP LJ4000 de 1996 (R5000) y estaciones de trabajo como Silicon Graphics Indigo 2 de 1995 (R10000). En la actualidad, otros muchos dispositivos contienen procesadores con esta arquitectura, incluyendo cámaras fotográficas y *routers*.

Ilustración 1 MIPS Technologies (http://imgtec.com/mips/)

Como sabemos, el lenguaje ensamblador es un lenguaje de bajo nivel relativamente próximo al lenguaje máquina; este último es el lenguaje específico que ejecutan todos los procesadores. Frente al lenguaje máquina, el lenguaje ensamblador ofrece al programador una serie de herramientas que facilitan enormemente el desarrollo de programas. Entre ellas, se incluye la utilización de códigos nemotécnicos para recordar las instrucciones, etiquetas para referenciar posiciones de memoria y, finalmente, directivas, las cuales dan indicaciones al programa ensamblador.

Resulta imprescindible distinguir entre **lenguaje ensamblador** y **programa ensamblador**, ya que si no lo hacemos podemos incurrir en graves errores de concepto. El programa ensamblador es el equivalente al compilador cuando programamos con lenguajes de alto nivel como Java o C. Si un compilador traduce de lenguaje de alto nivel a lenguaje máquina, un programa ensamblador traduce el lenguaje ensamblador a lenguaje máquina. Esta traducción se hace teniendo en cuenta las directivas que el programador incluye en el código del programa. Las directivas sirven, por ejemplo, para decidir en qué zona concreta de memoria hay que colocar las variables o en qué dirección de memoria está situada la primera instrucción del programa.

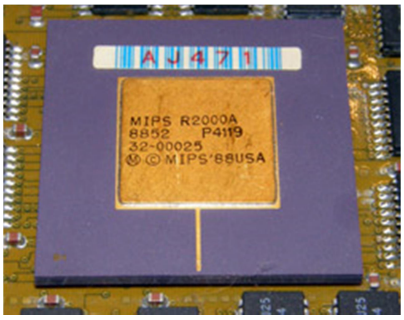

Ilustración 2 Encapsulado de un procesador MIPS R2000

El PCSpim incorpora, en una misma aplicación, tanto el programa ensamblador como el simulador de la arquitectura MIPS. En particular, se simula el modelo R2000 (Ilustración 2), un procesador de 32 bits diseñado en 1986 por la entonces compañía MIPS Computer Systems. Los primeros chips podían funcionar a una frecuencia de 15 MHz y usaban poco más de cien mil transistores. Para obtener mejor rendimiento en el procesamiento de números en coma flotante el R2000 puede complementarse con el R2010, un chip externo con una FPU (*Floating Point Unit*), el cual evita manejar números reales mediante instrucciones aritméticas y lógicas de enteros.

INTRODUCCIÓN AL SIMULADOR PCSPIM

Inicio y configuración

El programa se ejecuta desde el acceso directo que hay en "Inicio | Todos los programas | PCSpim". Cuando PCSpim se ejecuta por primera vez aparece la ventana de configuración mostrada en la Ilustración 3. Si ocurre esto, basta hacer clic con el ratón sobre la opción "Sí" para acceder a los parámetros de configuración del simulador.

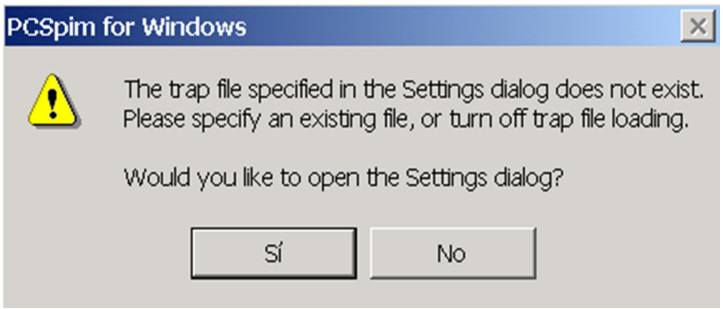

Ilustración 3 Ventana de diálogo que aparece la primera vez que se ejecuta PCSpim

Antes de comenzar a trabajar con el simulador es muy importante saber cómo está configurado. Para definir la configuración podemos hacerlo a través de la opción que mencionábamos anteriormente o bien acceder al menú "Simulator | Settings…" del simulador. La ventana de la Ilustración 4 muestra las opciones que recomendamos para trabajar en la sesiones de laboratorio donde trabajaremos con el lenguaje ensamblador.

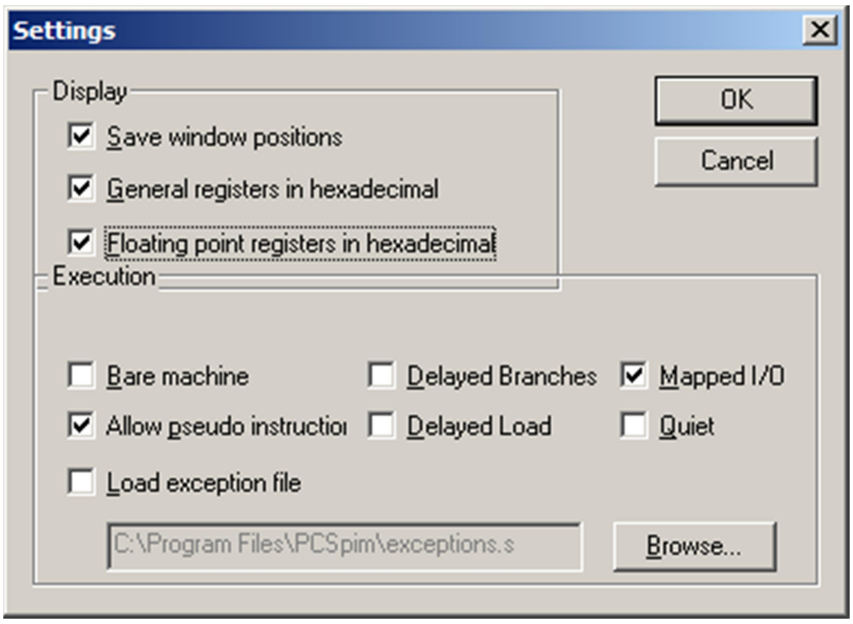

Ilustración 4 Configuración recomendada del simulador

Las opciones se pueden resumir en:

- La información contenida en los registros de números enteros (`$0...$31`) y de números reales (`$f0...$f31`) se muestran en hexadecimal. Como todos estos registros son de 32 bits, la información se mostrará en forma de 8 dígitos hexadecimales. Estas dos opciones se pueden cambiar si lo que queremos ver es el valor contenido en el registro en base decimal, algo bastante útil en muchas ocasiones.
- El programa ensamblador aceptará código con pseudoinstrucciones, es decir, instrucciones que en última instancia han de traducirse a una o varias instrucciones máquina.

La organización del entorno de trabajo del simulador se detalla en la Ilustración 5. En la pantalla se muestra el resultado obtenido tras cargar el programa contenido en el fichero `helloword.s` que viene incluido en el simulador. Después de la inicialización producida tras la carga de un programa, el simulador pone a cero la mayor parte de los contenidos de los registros y la memoria.

Dedique, en esta primera sesión, bastante tiempo a examinar esta pantalla y localizar los elementos más importantes. De esta habilidad dependerá el uso efectivo de la aplicación.

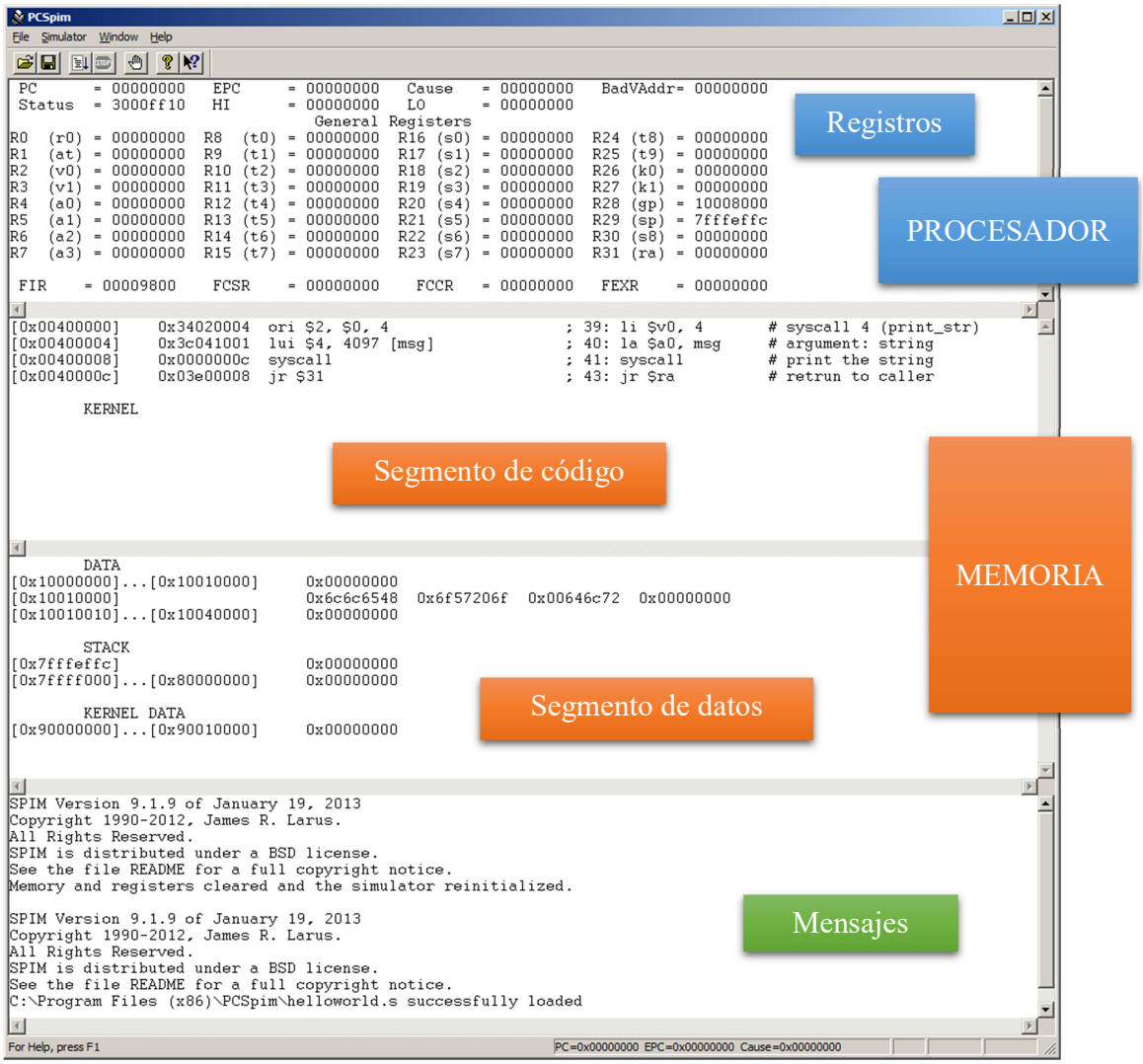

Ilustración 5 Entorno del simulador PCSpim dividido en cuatro secciones: registros del processador (números enteros y números reales), segmento de código (programa), segmento de datos (variables) y zona de mensajes

A continuación describimos cada una de las secciones en que se divide el entorno de trabajo.

VENTANA DE REGISTROS: muestra el valor almacenado en los registros del procesador. En esta ventana podemos ver el contador de programa (PC, *program counter)* y los registros `HI` y `LO` (registros especiales que almacenan el resultado de las multiplicaciones y divisiones de números enteros).

En la parte superior de la ventana vemos los registros (*General Registers*) para números enteros (`$0`…`$31`); nótese que, junto al número que actúa como identificador, también vemos un nombre alternativo que permite dar funciones específicas a cada uno de ellos. Por ejemplo, podemos usar el identificador `$8` o bien `$t0` (la letra *t* indica que el registro se usa de forma temporal).

Así mismo, si nos movemos con el ratón un poco más abajo en la ventana, veremos los registros para los números en coma flotante (*Double Floating Point Registers* o *Single Floating Point Registers*). Estos registros se etiquetan con los símbolos `$f0`…`$f31`. En este caso, los registros pueden interpretarse como 16 registros de doble precisión, esto es, de 64 bits (`$f0`, `$f2`, `$f4`, …, `$f30`) o bien 32 registros de simple precisión o de 32 bits (`$f0`, `$f1`, `$f2`, …, `$f31`). En cualquier caso, en los programas se recomienda hacer uso solamente de los registros con identificador par.

VENTANA DE SEGMENTO DE CÓDIGO: en esta ventana podemos ver el contenido de la memoria donde se ubica el programa una vez ensamblado. Dicho de otra manera, cada una de las posiciones de 32 bits de la memoria se corresponde con una instrucción máquina del programa. Los detalles mostrados por el simulador son:

- En la primera columna se muestra en hexadecimal y entre corchetes la dirección de memoria donde está almacenada cada una de las instrucciones del programa una vez ensamblado.
- En la segunda columna se muestra en hexadecimal la codificación de cada instrucción máquina.
- En la tercera columna se muestran los códigos nemotécnicos de estas instrucciones.
- Finalmente, en la cuarta columna, se muestran las instrucciones correspondientes tal como aparecen en el programa fuente. Nótese que, en ocasiones, las pseudoinstrucciones se traducen en más de una instrucción máquina.

Así, en el ejemplo mostrado en la Ilustración 5, la primera instrucción máquina del programa está almacenada en la dirección de memoria `0x00400000`, se codifica como `0x34020004`, su nemónico es `ori $2,$0,4` y es el resultado de traducir la instrucción del programa `li $v0,4` (acrónimo de *load immediate*, esta pseudoinstrucción se usa para inicializar registros con constantes).

VENTANA DE SEGMENTO DE DATOS: en esta ventana se muestra el contenido de la memoria donde se ubica la información definida en el segmento de datos. La memoria se despliega mostrando rangos de direcciones de cuatro palabras entre corchetes y el contenido de la memoria de este rango a su derecha.

En el ejemplo mostrado en la Ilustración 5 vemos que el segmento de datos comienza en la dirección `0x10000000` y todo el segmento está a cero hasta la dirección `0x10010000`, cuyo contenido es `0x6c6c6548` y corresponde con la cadena "`Hell`" codificada en ASCII. Nótese que la ordenación de los bytes se hace en modo *little endian*, por lo que la "`H`" está en la dirección `0x10010000`, la "`e`" en la `0x10010001`, la primera "`l`" en la `0x10010002` y así sucesivamente. En esta sesión los programas no hacen uso de la memoria de datos, por lo que veremos este espacio vacío.

VENTANA DE MENSAJES: aquí vemos varios detalles, como por ejemplo la versión del simulador. En el ejemplo mostrado se indica que el simulador ha inicializado (a cero) los registros y la memoria y, en la última línea, se nos informa de que el programa contenido en el archivo `helloword.s` ha sido correctamente ensamblado y cargado en la memoria.

Simulación de la ejecución de programas

Los pasos que hay que seguir para utilizar el simulador son:

- Escribir un programa en lenguaje ensamblador del MIPS R2000 utilizando un editor de texto plano. El programa se tiene que guardar en un fichero con extensión `.s` o `.asm`.
- Cargar el programa en el simulador (opción *File* | *Open*...).
- Seguir paso a paso la ejecución de las instrucciones (F10) e ir observando con detalle el valor de los registros.

1. Edición del programa

Para editar los programas basta con usar cualquier editor de texto plano. No se recomienda usar ningún programa de edición y composición de textos (por ejemplo, Wordpad, Word, etc.), ya que estas aplicaciones tienen sus propios formatos de codificación. Nosotros recomendamos usar un editor de texto profesional. Hay muchos programas de este tipo en el mercado, que además son gratuitos. En nuestro caso, hemos instalado el programa Notepad++ (https://notepad-plus-plus.org/), que tenéis disponible en los equipos del laboratorio.

Por último, no debemos olvidar que la etiqueta predefinida "`__start`" que se usa como marca de comienzo del código del programa empieza por dos caracteres de subrayado.

2. Carga del programa en el simulador

Para el ensamblado y la carga de un programa se utiliza el menú *File*, que tiene las opciones mostradas en la Ilustración 6. Nótese que estas dos operaciones, ensamblado y carga, se hacen en la misma operación. Si el programa tuviera errores sintácticos, el simulador no podrá efectuar el ensamblado y, por tanto, tampoco la carga del programa en la memoria.

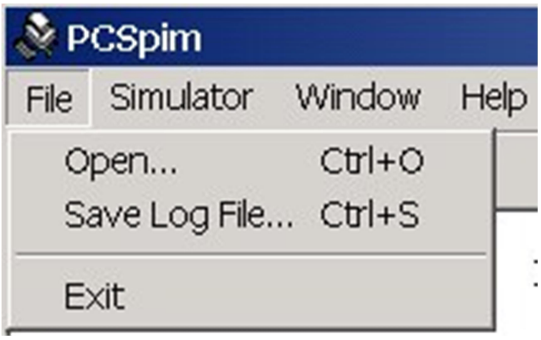

Ilustración 6 Menú para cargar el programa en el simulador

La opción *Open* es la que nos permite cargar ficheros en el simulador. La opción *Save Log File* tiene como finalidad almacenar en un fichero los mensajes que han aparecido en la ventana de mensajes. Su objetivo es analizar las salidas obtenidas al ejecutar el programa y detectar posibles causas de funcionamientos inadecuados. Finalmente, la opción *Exit* acaba la ejecución del simulador.

3. Simulación de la ejecución del programa

Mediante las diferentes opciones del menú *Simulator* es posible simular la ejecución del programa previamente cargado. La opción *Go* (tecla F5) lleva a cabo la ejecución completa del programa, es decir, ejecuta el programa en su totalidad (desde la primera instrucción hasta la última del programa).

Si lo que queremos es ejecutar el programa instrucción a instrucción, entonces hay que usar la opción *Single step*, que en la práctica se lleva a cabo mediante la pulsación de la tecla F10. Esta manera de ejecutar los programas resulta de gran utilidad ya que podemos ver cómo van cambiando los contenidos de los registros y de la memoria según avanza la ejecución del programa.

MANIPULACIÓN BÁSICA DE NÚMEROS ENTEROS

Escriba el código siguiente en el editor Notepad++, grábelo en un fichero con el nombre `ejemplo1.s` y cárguelo en el PCSpim.

```
            .text 0x00400000      # Dirección inicio código
            .globl __start        # Etiqueta global
__start:
            addi $8,  $0,  1      # Inicializa $8 con 1
            addi $9,  $0, -2      # Inicializa $9 con -2
            add  $10, $8, $9      # $10 <- $8 + $9
            .end                  # Final del programa
```

Pregunta 1. ¿Qué hacen las dos instrucciones `addi`?

Pregunta 2. ¿Qué valor, expresado en decimal, se espera que contenga el registro `$10`?

Pregunta 3. Rellene la tabla siguiente con el contenido mostrado en la ventana de registros después de cargar el programa y **antes de ejecutarlo**. Es importante que anote todos los dígitos hexadecimales para tener una idea de las longitudes de las palabras de bits.

Registro	Valor en hexadecimal (con todos los bits)	Valor en decimal
$8		
$9		
$10		

Pregunta 4. Ahora ejecute el programa empleando la opción *Go* del menú *Simulator*. Una vez finalizada la ejecución, rellene la tabla siguiente con el nuevo contenido de los registros.

Registro	Valor en hexadecimal (con todos los bits)	Valor en decimal
$8		
$9		
$10		

Pregunta 5. ¿Cuántas instrucciones máquina tiene el código del programa?

Pregunta 6. ¿Cuántos bytes de memoria ocupa el código del programa?

Pregunta 7. ¿En qué dirección de memoria se almacena la instrucción `addi $8,$0,1`? ¿Cómo se codifica esta instrucción?

Pregunta 8. ¿En qué dirección de memoria se almacena la instrucción `add $10,$8,$9`? ¿Cómo se codifica esta instrucción?

Pregunta 9 ¿Para qué sirven las directivas usadas en el programa?

LA MULTIPLICACIÓN DE NÚMEROS ENTEROS

A continuación vamos a examinar un programa que hace uso de la instrucción de división. El código a estudiar es el siguiente:

```
            .text 0x00400000      # Dirección inicio código
            .globl __start        # Etiqueta global
__start:
            addi $8, $0, 10000    # Inicializa $8 con 10^4
            mult $8, $8           # HI-LO <- $8 * $8
            mflo $9               # $9 <- LO
            mult $8, $9           # HI-LO <- $8 * $9
            .end                  # Final del programa
```

En este programa se hace uso de instrucciones de suma y de multiplicación de números enteros. Así mismo, se utiliza una instrucción de movimiento de datos entre el banco de registros enteros (`$0…$31`) y un registro especial de la unidad aritmético-lógica (`LO`) denominada `mflo`, acrónimo de *Move from LO*.

En el caso de la operación aritmética de multiplicación es importante tener en cuenta que, cuando se multiplican dos números enteros de 32 bits, el resultado puede llegar a ocupar 64 bits.

Vamos a analizar el programa por partes. Escriba el código en el editor Notepad++, grábelo en un fichero con el nombre `ejemplo2.s` y cárguelo en el PCSpim.

Después de cargarlo y comprobar que no hay ningún error, vamos a ejecutar el programa paso a paso pero solamente hasta que se complete la segunda instrucción del programa (`mult $8, $8`). Para ello deberá pulsar **dos veces** la tecla F10, hasta que se destaque en color azul la instrucción `mflo $9`. La línea destacada en azul es siempre la siguiente instrucción que se ejecutaría con una nueva pulsación de F10.

Pregunta 10. Tras ejecutar las dos primeras instrucciones del programa (`add`, `mult`) anote el contenido de los registros especiales `HI` y `LO`. Es importante que anote todos los dígitos hexadecimales para tener una idea de las longitudes de las palabras de bits.

Registro	Valor en hexadecimal (los 32 bits)	Valor en decimal
`HI`		
`LO`		

Pregunta 11. ¿Cuál ha sido el resultado completo de la instrucción `mult $8,$8`? Expréselo en hexadecimal (los 64 bits) y en decimal.

Valor en hexadecimal (los 64 bits)	Valor en decimal

Pregunta 12. Ejecute ahora, pulsando una vez más la tecla F10, la instrucción `mflo $9`. ¿Qué hace esta instrucción?

Pregunta 13. Como se habrá comprobado, el registro `$9` contiene en este momento la parte baja del resultado calculado por el primer producto. ¿Cabe el resultado completo del producto en 32 bits? Justifique la respuesta.

Pregunta 14. Finalmente, complete la ejecución del programa pulsando una cuarta vez la tecla F10 para que se ejecuta la última instrucción. ¿En qué registros encontramos el resultado calculado? ¿Cuál es el valor obtenido? ¿Ha cabido en 32 bits?

Valor en hexadecimal (64 bits)	Valor en decimal

LA DIVISIÓN DE NÚMEROS ENTEROS

A continuación vamos a examinar un programa que hace uso de la instrucción de división entera. Tenga en cuenta que en este tipo de operaciones se generan dos resultados: por un lado, el cociente de la división, y por otro, el resto. El código a estudiar es el siguiente:

```
            .text 0x00400000      # Dirección inicio código
            .globl __start        # Etiqueta global
__start:
            addi $8, $0, 43       # Inicializa $8 con 43
            addi $9, $0, 12       # Inicializa $9 con 12
            div  $8, $9           # Divide $8 entre $9
            mflo $10              # LO -> $10
            mfhi $11              # HI -> $11
            .end                  # Final del programa
```

Escriba el código en el editor Notepad++, grábelo en un fichero con el nombre `ejemplo3.s` y cárguelo en el PCSpim. Ejecute el programa paso a paso y compruebe cómo van cambiando los valores almacenados en los registros que maneja el programa.

Pregunta 15. ¿En qué registro se almacena el cociente de la división? ¿Y el resto?

Pregunta 16. ¿Para qué se usan las instrucciones `mflo` y `mfhi`?

Pregunta 17. ¿Cambiaría el resultado si eliminamos del programa las dos instrucciones anteriores?

Pregunta 18. ¿En qué dirección de memoria se almacena la última instrucción del programa?

Pregunta 19. ¿Cuántos bytes ocupa el programa en memoria?

DISEÑO DE UN PROGRAMA EN LENGUAJE ENSAMBLADOR

En base a todos los conocimientos tratados a lo largo de esta sesión de laboratorio, vamos a diseñar un programa en lenguaje ensamblador equivalente al siguiente segmento de código escrito en un lenguaje de alto nivel. El operador `%` corresponde a la operación módulo (resto de la división entera).

```
int a, b, c, d;

a = 20;
b = -8;
c = (a*15)/(a-b);
d = (a*15)%(a-b);
```

Pregunta 20. Calcule, usando la calculadora, el valor asignado a las variables `c` y `d`.

Pregunta 21. Escriba el código ensamblador del programa y compruebe que el resultado de la ejecución coincide con el cálculo anterior.
Suponga que las variables del programa se almacenan en los siguientes registros del procesador: `a` en `$8`, `b` en `$9`, `c` en `$10` y `d` en `$11`; si necesita más registros, use `$12`..`$15`. Grabe el programa en un fichero con el nombre `ejemplo4.s`.

Pregunta 22. Indique las direcciones de la primera y de la última instrucción del programa.

Pregunta 23. ¿Cuántas palabras de 32 bits ocupa el programa? ¿A cuántos bytes corresponde esta cantidad?