

Prácticas de ensamblador MIPS

Este documento recoge el material necesario para la realización de las prácticas de programación en ensamblador de la asignatura Tecnología de los Computadores. A continuación se presenta un índice con el contenido del documento.

1. Introducción y Objetivos
2. Simulador MARS
3. Programación en ensamblador MIPS
4. Ejercicios de laboratorio
5. Características del procesador MIPS

1. Introducción y Objetivos

Con esta práctica se pretende que el alumno se familiarice con el funcionamiento a nivel ISA (Instruction Set Architecture) de un procesador. Concretamente vamos a utilizar el procesador MIPS (de Microprocessor without Interlocked Pipeline Stages). Para ello al alumno se le presentarán una serie de códigos en ensamblador para que los entienda y después los vea en ejecución en un simulador. Posteriormente el alumno deberá desarrollar un programa en ensamblador para la resolución de un problema propuesto.

Los objetivos que se persiguen son:

- Comprender el funcionamiento básico del procesador MIPS
- Familiarizarse con el nivel ISA del procesador MIPS
- Introducción a la programación en ensamblador
- Utilización de un simulador de un procesador a nivel ISA

2. Simulador MARS (MIPS Assembler and Runtime Simulator)

Para realizar las prácticas con el ensamblador MIPS, puesto que no podemos acceder a una plataforma con dicho procesador, vamos a utilizar un software emulador del mismo. Dicho software se llama MARS y nos va a permitir:

- cargar y editar programas escritos en ensamblador MIPS
- ejecutar (con opción paso a paso) los programas cargados
- visualizar en todo momento el contenido de los registros del procesador así como de la memoria del sistema simulado

Descarga e instalación del programa

El programa es multiplataforma puesto que está escrito en lenguaje JAVA. Por tanto para su utilización bastará con descargarlo desde su web (<http://courses.missouristate.edu/KenVollmar/MARS/index.htm>) y disponer del runtime de JAVA. Una vez descargado, bastará con ejecutar el fichero .jar descargado.

Descripción del programa

(http://www.thehouseofblogs.com/articulo/mars_simulador_e_ide_para_lenguaje_ensamblador-47369.html)

MARS es un entorno de desarrollo integrado (IDE) ligero para programar en lenguaje ensamblador MIPS, destinado a la educación.

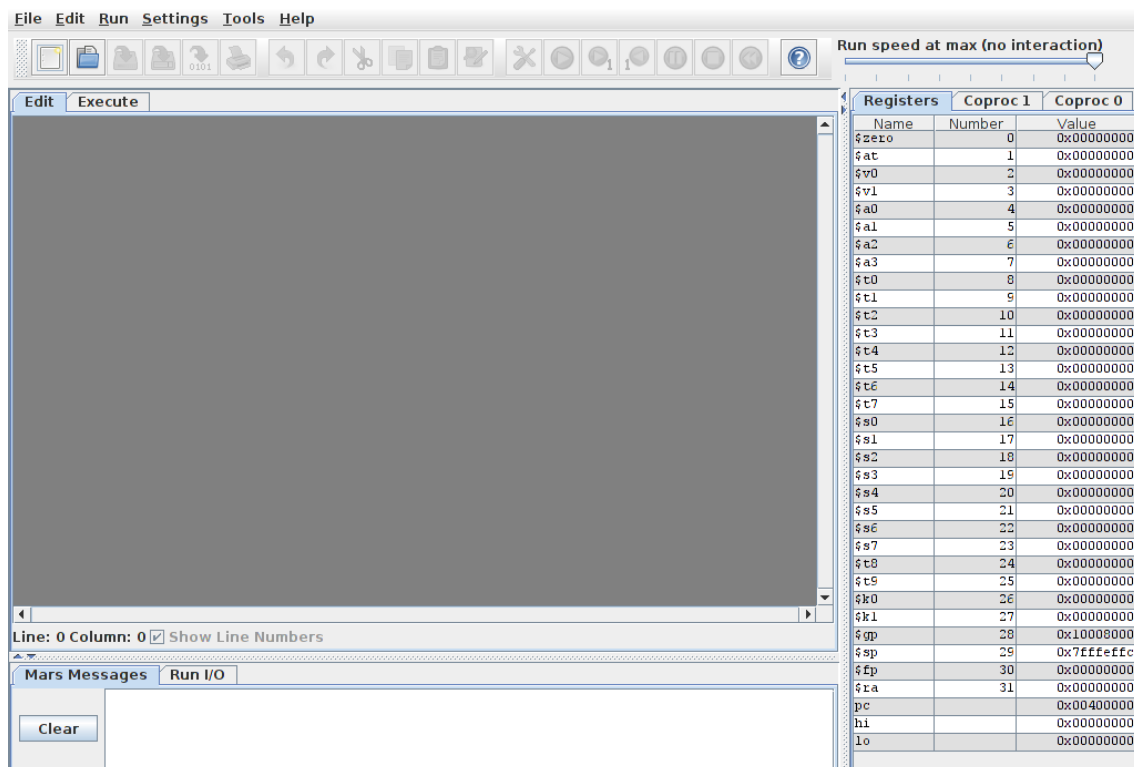
Ventajas Pedagógicas

Multiplataforma: MARS está escrito en lenguaje JAVA, y se distribuye empaquetado en formato .jar por lo que en teoría se puede correr en cualquier computadora que tenga instalada la máquina virtual de Java (JVM).

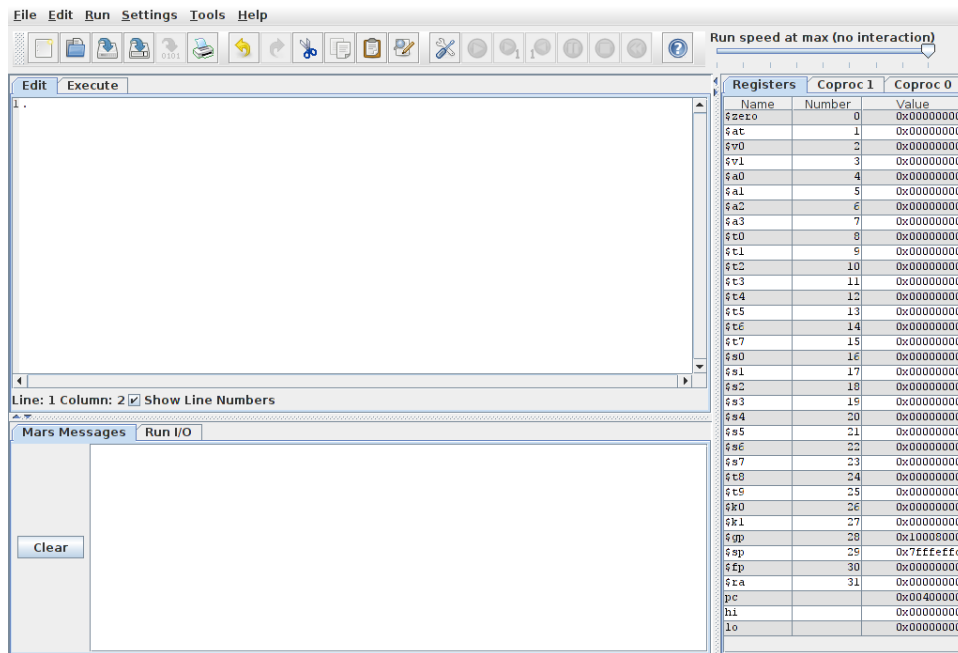
Licencia MIT: Lo que significa que puede usarse sin restricciones; incluyendo usar, copiar, modificar (por lo tanto adaptarlo a necesidades específicas), integrar con otro Software, publicar, sublicenciar o vender copias del Software, y además permitir a las personas a las que se les entregue el Software hacer lo mismo.

A pesar de ser una aplicación que del tipo "Simulación" presenta un interesante comportamiento que guía al aprendiz en el proceso de creación, ensamblaje y ejecución de un programa.

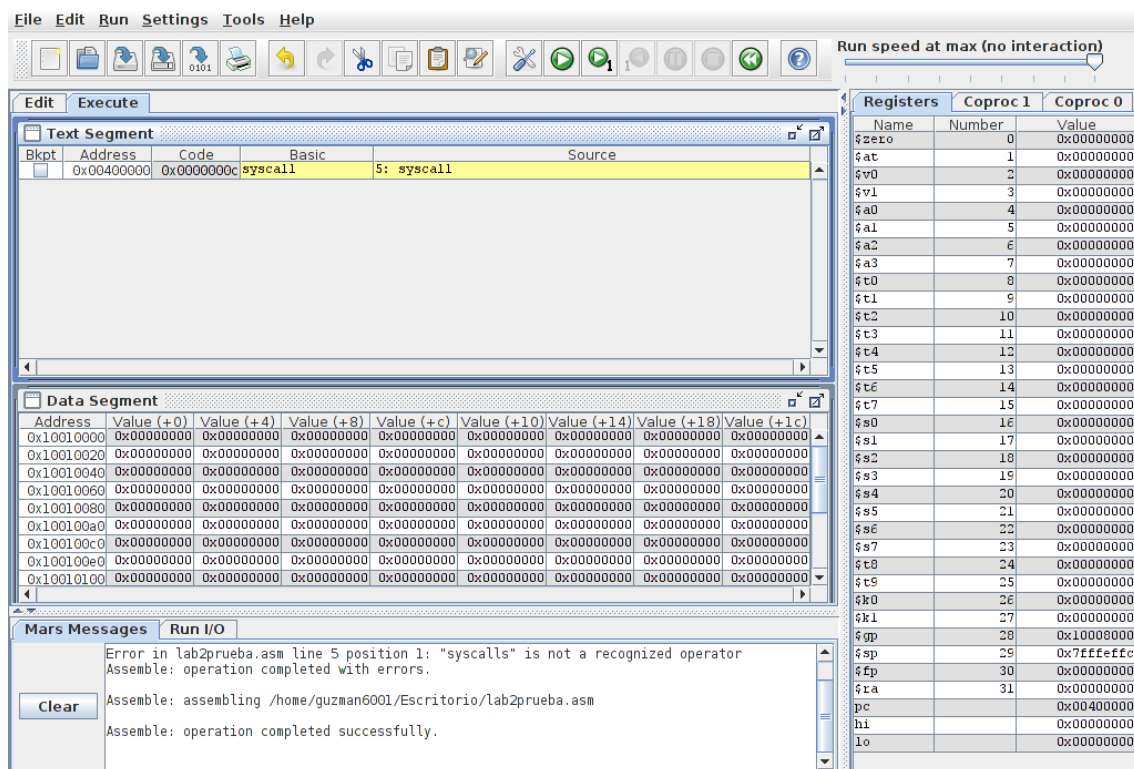
Al iniciar MARS sólo tiene 3 funciones habilitadas en la barra estándar: **new** (crear un archivo vacío), **open** (abrir un archivo ensamblador existente) y **help** (despliega la ayuda).



Al crear o abrir un archivo, se activan las funciones de edición de texto (deshacer, rehacer, cortar, copiar, pegar, buscar/reemplazar), de manejo de archivo (guardar, guardar como..., imprimir) y para la ejecución se activa sólo "assemble" (ensamblar) en el menú "run" (o en el grupo de íconos correspondientes a la ejecución).



Cuando un usuario pulsa "assamble", si el código tiene errores serán mostrados en la ventana "Mars Messages" describiendo el archivo, las líneas y los detalles correspondientes a los errores, si no hay errores se oculta la pestaña "Edit" y se activa la pestaña "Execute", que muestra por separado la vista del segmento de datos en memoria (pudiéndose editar sus valores) y el segmento de código, activándose las funciones de ejecución "Ejecutar", "Ejecutar una sola instrucción" y "Reiniciar", si está en ejecución es posible pausar la corrida o detenerla, incluso es posible retroceder una instrucción.



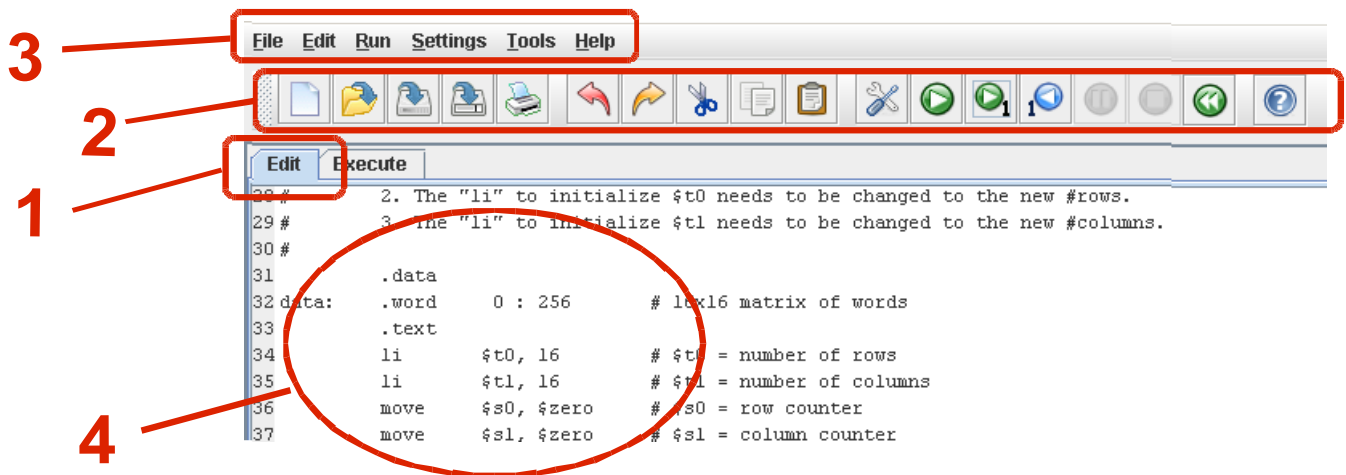
Funciones de depuración altamente intuitivas: Usualmente los IDEs de programación en cualquier lenguaje, agregan una opción de depuración en la que agregan todas las herramientas disponibles para este fin. En MARS están en la barra de herramientas estándar por lo que están disponibles en un sólo paso. Entre ellas están:

1. Ejecutar una instrucción.
2. Retroceder los cambios de una instrucción ejecutada.
3. Pausar la ejecución.
4. Detener la ejecución completamente.
5. Ajustar la velocidad de ejecución.

Todas esas opciones se complementan con otras funciones más comunes tales como

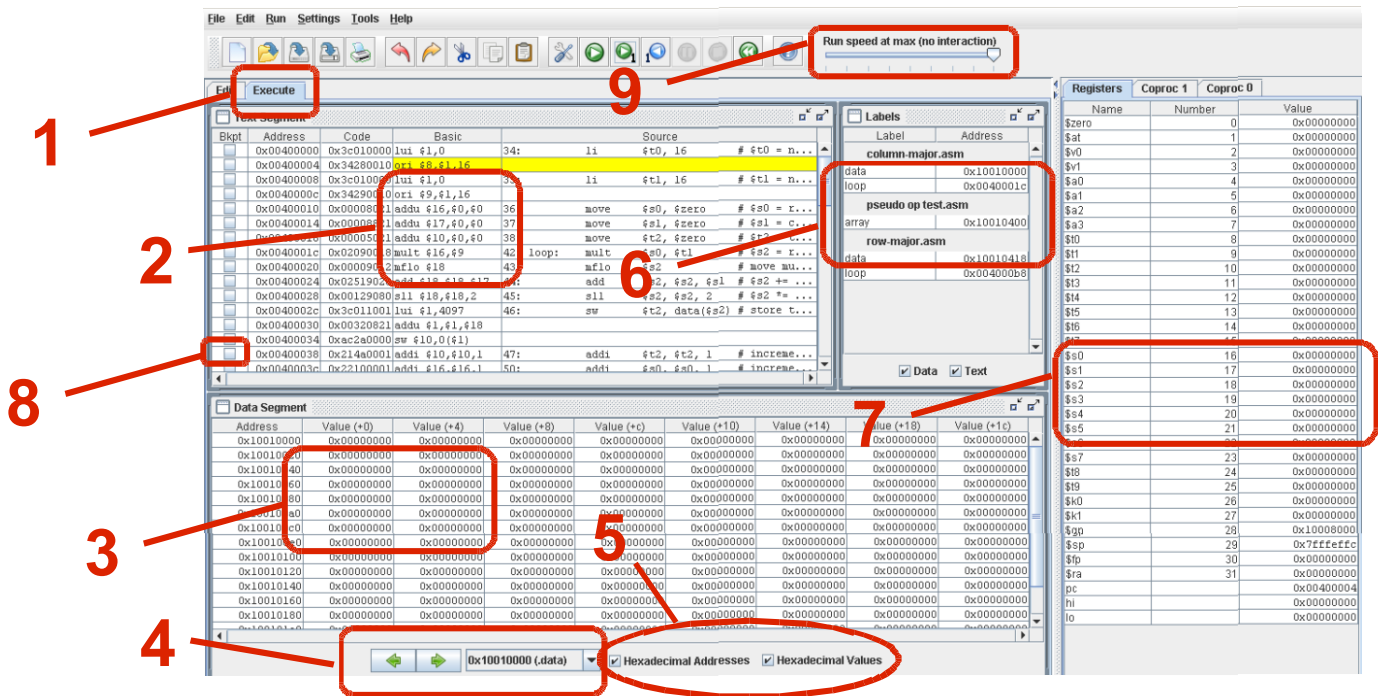
6. Ejecutar.
7. Ensamblar (limpia cualquier breakpoint en el programa).
8. Reiniciar Registros y Memoria.

Si volvemos a la ventana principal del programa:



1. Podemos ver que el tab seleccionado nos muestra que estamos en edición.
2. Las acciones típicas de edición y ejecución están disponibles por medio de iconos
3. o por medio de menús. Las que están oscurecidas no están disponibles o no se pueden aplicar.
4. Editor WYSIWYG para código en lenguaje ensamblador MIPS.


Si cambiamos el tab a ejecución obtenemos esta ventana:

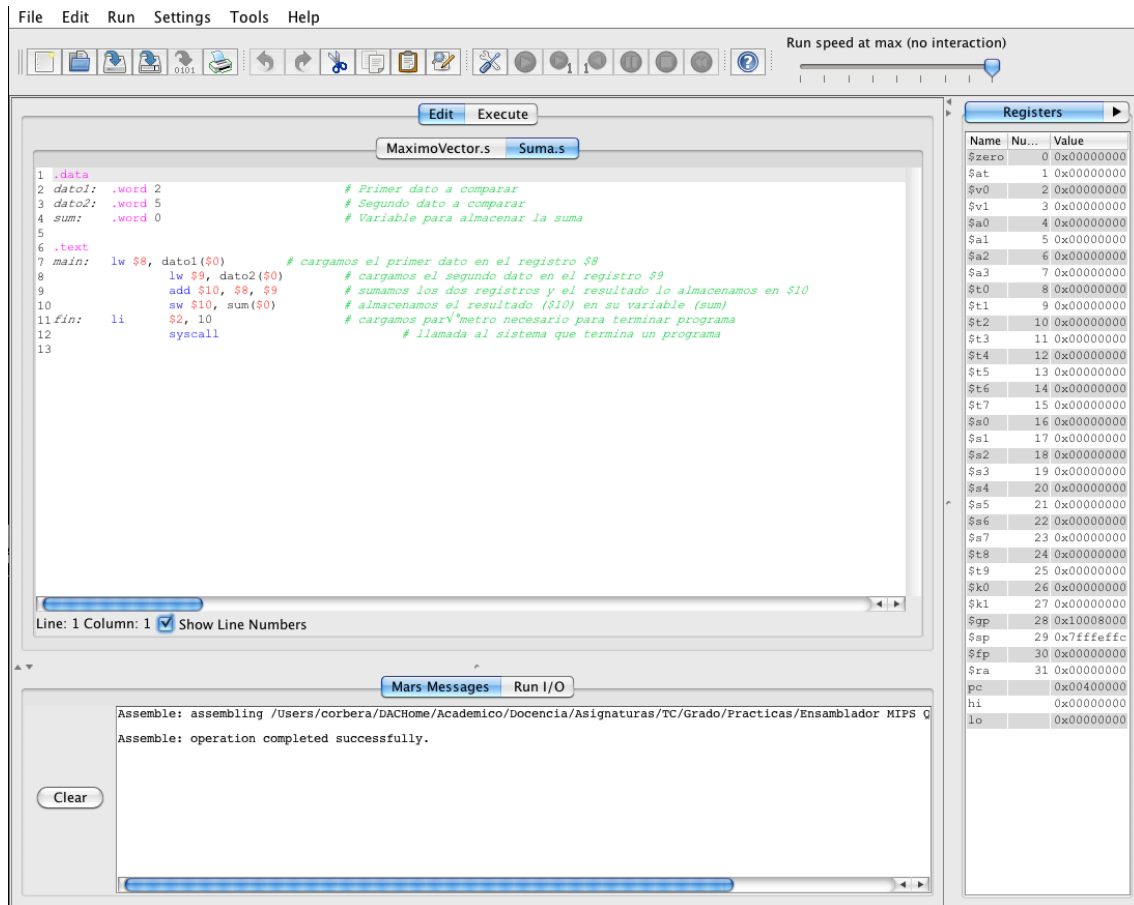



1. La pestaña seleccionada nos indica que estamos en modo ejecución
2. El código ensamblador se visualiza junto con su dirección, código máquina y la correspondiente línea del fichero fuente (el código fuente y código ensamblador pueden ser diferentes cuando se utilizan pseudo-instrucciones).
3. Los valores almacenados en memoria se pueden editar directamente.
4. La ventana con los contenidos de la memoria se puede controlar de varias formas: con las flechas siguiente/anterior y con un menú de sitios destacados (tope de la pila, ...).
5. Se puede seleccionar las bases decimal y hexadecimal para representar los valores y las direcciones (memoria y registros).
6. También se puede acceder a las direcciones de las etiquetas declaradas en el código (main, direcciones de salto, etc...)
7. Los valores almacenados en los registros también son editables.
8. Se puede poner un punto de ruptura en cualquier instrucción máquina activando su correspondiente checkbox.
9. La velocidad de ejecución del programa se puede variar para ver como ocurre la acción de cada instrucción en vez de ver sólo el resultado final.

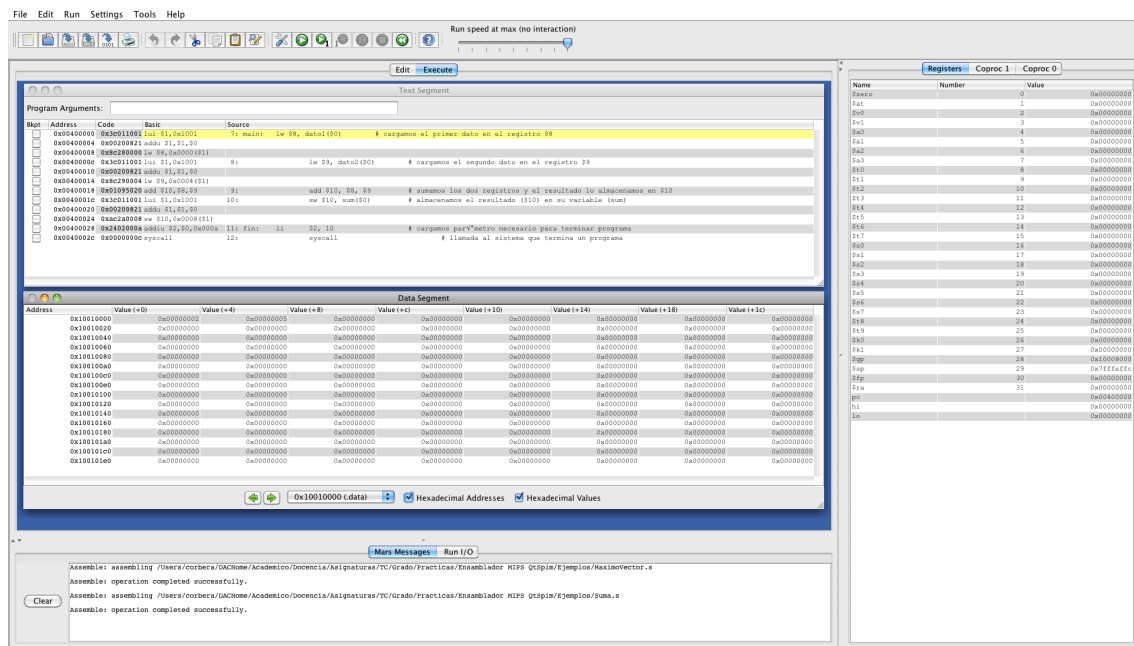
Uso básico del simulador MARS

Los ficheros fuente con los que trabaja el programa, son ficheros de texto con extensión .s o .asm que contienen código en ensamblador MIPS. Para editar dichos ficheros utilizaremos el tab Edit de MARS.





También podremos cargar un fichero previamente creado con las opciones del menú File (open) o con el botón  de la barra de herramientas. Una vez cargado, el aspecto será similar al de la siguiente figura.



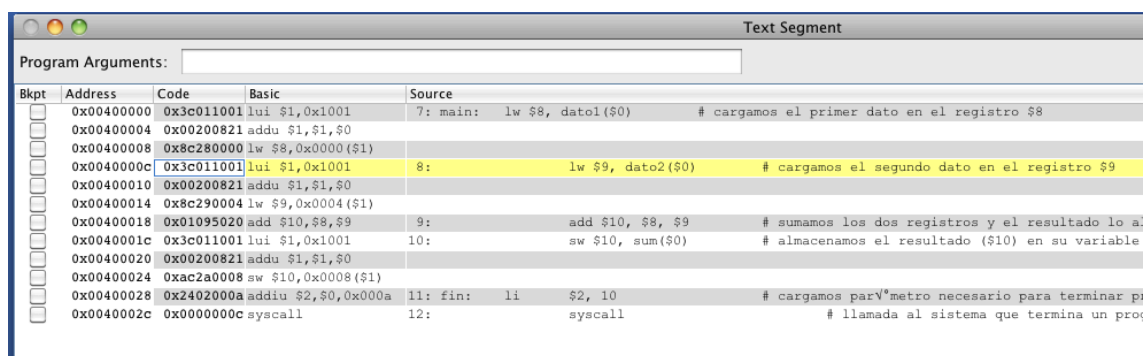
En esa ventana podremos editar el código fuente. Para poder empezar a trabajar con el simulador, deberemos compilar el código ensamblador. Para ello utilizaremos el botón  que automáticamente nos pasará a la ventana de ejecución como la de la siguiente pantalla.



Para ejecutar el programa existen varias opciones:

- El icono  ejecuta el programa hasta el final.
- El icono  resetea el programa y el simulador a sus valores iniciales. El contenido de la memoria será el especificado por el programa y el de los registros será generalmente cero.
- El icono  ejecutará una instrucción cada vez. Esto nos ayudará a ver la evolución de los registros y de la memoria paso a paso. La instrucción a ejecutar aparecerá resaltada en el código. Su complementario es , que deshace los cambios producidos por la última instrucción (undo).

La siguiente figura muestra con un poco más de detalle la ventana que muestra el segmento de texto (código):



- La primera columna (**Bkpt**) son los checkbox para establecer los puntos de ruptura (breakpoints).
- La siguiente columna (**Address**) especifica la dirección de memoria dónde ha sido almacenada la instrucción.
- La tercera columna (**Code**) muestra el código máquina asociado a cada instrucción en ensamblador.
- A continuación tenemos la columna (**Basic**) que muestra la representación en ensamblador de la instrucción máquina de la columna code.
- Por último tenemos la columna (**Source**) que muestra el código fuente original incluidos comentarios. Podemos observar como algunas instrucciones del código original han sido transformadas en varias instrucciones máquina.

Así por ejemplo, la línea resaltada que sería la próxima instrucción a ejecutar (paso a paso) nos indica que:

- la instrucción ha sido cargada en la posición de memoria 0x004000C
- el código máquina de la instrucción (32 bits) es 0x3C011001
- la instrucción ensamblador asociada es: lui \$1, 0x1001
- la instrucción ensamblador original de la que se deriva la anterior sería: lw \$9, dato2(\$0)

La ventana para inspeccionar la memoria (**Data Segment**) muestra el segmento de datos del programa, la pila y el espacio del kernel (sistema operativo), etc . El que nos interesa por ahora es el segmento de datos del usuario (.data). En la siguiente imagen vemos un ejemplo de segmento de datos:

Data Segment									
Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)	
0x10010000	0x00000002	0x00000005	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	
0x10010040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	
0x10010060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	
0x10010080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	
0x100100a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	
0x100100c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	
0x100100e0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	
0x10010100	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	
0x10010120	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	
0x10010140	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	
0x10010160	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	
0x10010180	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	
0x100101a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	
0x100101c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	
0x100101e0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	

Vemos varias columnas:

- La primera columna nos muestra la dirección base de memoria que representa la fila
- El resto de columnas muestran el contenido de memoria de la posición base más el desplazamiento especificado en la columna (**Value(+desp)**)

Así, en el ejemplo, la primera dirección visualizada es la 0x10010000, cuyo contenido sería 0x00000002 (primera columna Value(+0)). La siguiente columna muestra un valor de 0x00000005 correspondiente a la dirección 0x10010004 (la dirección base de la fila 0x10010000 más el desplazamiento correspondiente a la columna (+4))

3. Programación en ensamblador MIPS

Para programar en ensamblador MIPS además de conocer las instrucciones disponibles (tanto su significado como su nomenclatura), se tienen que conocer una serie de reglas sintácticas y léxicas que debe respetar todo fichero con un programa en ensamblador para que pueda ser entendido por el compilador.

Para programar en ensamblador de una manera cómoda y sencilla se tienen que utilizar etiquetas y directivas para el compilador.

Las **etiquetas** identifica una instrucción, su valor es la posición en memoria de dicha instrucción.

Las **directivas** sirven para dar estructura al programa (`.data`, `.text`) y para definir tipos de datos (`.byte`, `.word`, `.ascii`, ...).

La estructura de un programa en ensamblador consta de una serie de variables que empiezan con la directiva `.data` y de una serie de instrucciones consecutivas que empiezan por la directiva `.text`.

En el apartado `.data`, la declaración de variables cada línea está formada por:

- una etiqueta (seguida de “:”)
- una directiva de definición de datos
- los datos
- comentarios (comenzando por “#”)

Por ejemplo:

```
.data
micadena: .ascii "cadena ejemplo" # caracteres
palabra:  .word 50                # esto ocupa 4 bytes
vectorW:  .word 2,6,9,1           # 4 posiciones de 4 bytes consecutivas
vectorB:  .byte 3,5,7             # esto ocupa 3 bytes
bloque:   .space 30               # 30 bytes no inicializados
```

En el apartado `.text`, se escriben las instrucciones que forman el programa. Cada línea está formada por:

- una etiqueta (opcional)
- un mnemónico (identifica la instrucción)
- operandos
- comentarios (si existe comienza por “#”)

Por ejemplo:

```
main: lw $8, dato1($0) # cargamos dato1 en registro $8
```

A continuación tenemos el código completo de un programa que suma dos variables (`dato1` y `dato2`) y el resultado lo almacena en otra variable (`sum`):

```

.data
dato1: .word 2    # Primer dato a comparar
dato2: .word 5    # Segundo dato a comparar
sum:   .word 0    # Variable para almacenar la suma

.text
main: lw $8, dato1($0) # cargamos el primer dato en el registro $8
      lw $9, dato2($0) # cargamos el segundo dato en el registro $9
      add $10, $8, $9  # sumamos los registros y el resultado va a $10
      sw $10, sum($0)  # almacenamos el resultado ($10) en sum
fin:  li $2, 10        # cargamos parámetro necesario para syscall
      syscall          # llamada al sistema que termina un programa

```

4. Ejercicios de laboratorio

Ejercicio 1.

Carga el programa *Suma.s* en el simulador *MARS*. Inspecciona el segmento del texto y de los datos. Contesta a las siguientes cuestiones:

1. ¿En qué posición de memoria se ha cargado la primera instrucción de tu programa (la etiquetada con *main*)?

main	
------	--

2. ¿Que posición de memoria se le ha asociado a las variables?

dato1	
dato2	
sum	

3. ¿Por qué la instrucción `lw $9, dato2($0)` se ha descompuesto en tres?

4. Codifica en binario la instrucción `add $10, $8, $9` (puedes consultar los formatos de instrucción al final de este documento).

Binario	Hexadecimal

Comprueba que tu codificación coincide con la que aparece en el simulador.

Ejecuta el programa completo y comprueba que en la variable `sum` se ha cargado el correspondiente valor (suma de `dato1` y `dato2`).

Vuelve a ejecutar el programa pero ahora paso a paso y ve comprobando la correcta ejecución de todas las instrucciones que lo componen (mirando como cambia el contenido de los registros y/o memoria implicados en cada una). Cambia el valor de las variables y ejecuta el código para ver los nuevos resultados.

Ejercicio 2.

Carga el programa *Maximo.s* en el simulador. Este código va a escribir en la variable `max` el valor máximo contenido en las variables `dato1` y `dato2`. Para entender el código deberás repasar el significado y funcionamiento de las instrucciones `slt` y `beq`.

Ejecuta el programa paso a paso y observa como se comporta el código para conseguir su objetivo. Cambia el valor de los datos para forzar que se comporte de manera distinta.

Ejercicio 3.

Carga el programa *SumaVector.s* que como su nombre indica realiza la suma de todos los valores almacenados en un vector. La variable `tam` indica el tamaño del vector, `datos` contiene los valores del vector y `res` contendrá el resultado de la suma.

A continuación aparece el código fuente, comenta al lado de cada instrucción que sentido tiene en el código (para conseguir el objetivo de sumar todos los valores de un vector):

```
.data
tam: .word 8
datos: .word 2, 4, 6, 8, -2 -4, -6 -7
res: .word 0
.text
main: lw $8, tam($0)
-----
      la $9, datos
-----
      sub $11, $11, $11
-----
loop: lw $10, 0($9)
-----
      add $11, $11, $10
-----
      addi $9, $9, 4
-----
      addi $8, $8, -1
-----
      beq $8, $0, salir
-----
      j loop
-----
salir: sw $11, res($0)
-----
      li $2, 10
      syscall
```

Ejecuta el código y comprueba su correcto funcionamiento. Cambia los valores del vector (tamaño incluido) y comprueba que sigue funcionando correctamente. Fíjate especialmente en aquellas instrucciones necesarias para el control del bucle.

En cuanto a la implementación del bucle, ¿Tú lo habrías hecho de la misma forma? ¿Se te ocurre otra forma de hacerlo?

Ejercicio 4.

Copia el código anterior con el nombre `MaximoVector.s` y modifícalo para que en vez de sumar los elementos del vector *datos*, lo que almacene en *res* sea el valor máximo encontrado en dicho vector.

Ejecuta el código que has creado y comprueba su correcto funcionamiento. Haz varias pruebas cambiando el tamaño y los valores almacenados en el vector.

5. Características del procesador MIPS

Características del procesador MIPS

Se trata de un procesador de 32 bits:

- Registros de 32 bits
- ALU con operandos de 32 bits
- Ancho de los buses de 32 bits

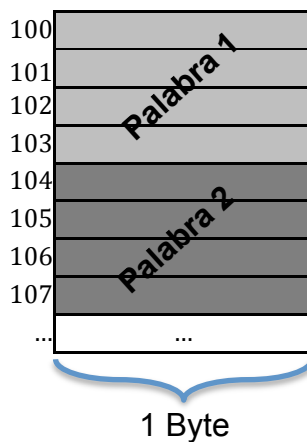
Tiene un conjunto de 32 registros de propósito general visibles por el programador.

Instrucciones de tamaño fijo de 32 bits (1 palabra).

Organización de la memoria:

- Se organiza en palabras de 4 bytes (32 bits)
- Direccionamiento a nivel de byte:
 - Dos palabras de 32 bits consecutivas en memoria están separadas por 4 posiciones. Así por ejemplo si la primera de dos instrucciones MIPS consecutivas (que ocupan cada una 1 palabra) estuviera en la posición de memoria $100_{(16)}$ la siguiente instrucción estaría en la posición $104_{(16)}$.
 - Todas las instrucciones están en direcciones múltiplo de 4 (los dos bits menos significativos serán 00)
 - Las instrucciones deben estar en posiciones alineadas: PC (32 bits) se incrementará de 4 en 4
- Número de bytes en la memoria: 2^{32}
- Número de palabras en la memoria: $2^{32}/4 = 2^{30}$

MEMORIA



CUADRO RESUMEN DEL LENGUAJE ENSAMBLADOR BÁSICO DEL MIPS-R2000

CARGA		ARITMÉTICAS		COMPARACIONES	
lw rt, dirección	Carga palabra Carga los 32 bits almacenados en la palabra de memoria especificada por dirección en el registro rt. lw \$s0, 12(\$a0) # \$s0 ← Mem[12 + \$a0]	add rd, rs, rt	Suma Suma el contenido de los registros rs y rt, considerando el signo. El resultado se almacena en el registro rd add \$t0, \$a0, \$a1 # \$t0 ← \$a0 + \$a1	slt rd, rs, rt	Activa si menor Pone el registro rd a 1 si rs es menor que rt y a 0 en caso contrario slt \$t0, \$a0, \$a1 # if (\$a0 < \$a1) \$t0 ← 1 # else \$t0 ← 0
lb rt, dirección	Carga byte y extiende signo Carga los 8 bits almacenados en el byte de memoria especificado por dirección en el LSB del registro rt y extiende el signo lb \$s0, 12(\$a0) # \$s0(7..0) ← Mem[12 + \$a0] _(1byte) # \$s0(31..8) ← \$s0(7)	addu rd, rs, rt	Suma sin signo Suma el contenido de los registros rs y rt, sin considerar el signo. El resultado se almacena en el registro rd addu \$t0, \$a0, \$a1 # \$t0 ← \$a0 + \$a1	slti rt, rs, inm	Activa si menor con inmediato Pone el registro rt a 1 si rs es menor que el dato inmediato inm y a 0 en caso contrario slti \$t0, \$a0, -15 # if (\$a0 < -15) \$t0 ← 1 # else \$t0 ← 0
lbu rt, dirección	Carga byte y no extiende signo Carga los 8 bits almacenados en el byte de memoria especificado por dirección en el LSB del registro rt sin extender el signo lbu \$s0, 12(\$a0) # \$s0 ← 0x000000(Mem[12 + \$a0]) _(1byte)	sub rd, rs, rt	Resta Resta el contenido de los registros rs y rt considerando el signo. El resultado se almacena en el registro rd sub \$t0, \$a0, \$a1 # \$t0 ← \$a0 - \$a1	seq rdest, rsrc1, rsrc2	Activa si igual Pone el registro rdest a 1 si rsrc1 es igual que rsrc2 y a 0 en caso contrario seq \$t0, \$a0, \$a2 # if (\$a0 == \$a2) \$t0 ← 1 # else \$t0 ← 0
lh rt, dirección	Carga media palabra y ext. signo Carga media palabra (16 bits) almacenada en la media palabra de memoria especificada por la dirección en la parte baja del registro rt y extiende el signo lh \$s0, 12(\$a0) # \$s0(15..0) ← Mem[12 + \$a0] _(2bytes) # \$s0(31..16) ← \$s0(15)	subu rd, rs, rt	Resta sin signo Resta el contenido de los registros rs y rt, sin considerar el signo. El resultado se almacena en el registro r. subu \$t0, \$a0, \$a1 # \$t0 ← \$a0 - \$a1	sge rdest, rsrc1, rsrc2	Activa si mayor o igual Pone el registro rdest a 1 si rsrc1 es mayor o igual que rsrc2 y a 0 en caso contrario sge \$t0, \$a0, \$a2 # if (\$a0 >= \$a2) \$t0 ← 1 # else \$t0 ← 0
lhu rt, dirección	Carga media palabra y no ext. signo Carga media palabra (16 bits) almacenada en la media palabra de memoria especificada por la dirección en la parte baja del registro rt y no extiende el signo lhu \$s0, 12(\$a0) # \$s0 ← 0x0000Mem[12 + \$a0] _(2bytes)	addiu rt, rs, valor	Suma inmediata sin signo Suma el contenido del registro rs con el valor inmediato, considerando el signo. El resultado se almacena en el registro rt. addiu \$t0, \$a0, -24 # \$t0 ← \$a0 + (-24)	sgt rdest, rsrc1, rsrc2	Activa si mayor Pone el registro rdest a 1 si rsrc1 es mayor que rsrc2 y a 0 en caso contrario sgt \$t0, \$a0, \$a2 # if (\$a0 > \$a2) \$t0 ← 1 # else \$t0 ← 0
la reg, dirección	Carga dirección Carga la dirección calculada en reg la \$s0, VAR # \$s0 ← dir. asociada a etiqueta VAR	mult rs, rt	Multiplicación Multiplica el contenido de los registros rs y rt. Los 32 MSB del resultado se almacenan en el registro HI y los 32 LSB en el registro LO mult \$s0, \$s1 # \$HI ← (\$s0 * \$s1) (31...16) # \$LO ← (\$s0 * \$s1) (15...0)	sle rdest, rsrc1, rsrc2	Activa si menor o igual Pone el registro rdest a 1 si rsrc1 es menor o igual que rsrc2 y a 0 en caso contrario sle \$t0, \$a0, \$a2 # if (\$a0 <= \$a2) \$t0 ← 1 # else \$t0 ← 0
lui rt, dato	Carga inmediata superior Carga el dato inmediato en los 16 MSB del registro rt lui \$s0, 12 # \$s0(31..16) ← 12 # \$s0(15..0) ← 0x0000	div rs, rt	División Divide el registro rs por el rt. El cociente se almacena en LO y el resto en HI. div \$s0, \$s1 # \$LO ← \$s0 / \$s1 # \$HI ← \$s0 % \$s1	sne rdest, rsrc1, rsrc2	Activa si no igual Pone el registro rdest a 1 si rsrc1 es diferente de rsrc2 y a 0 en caso contrario sne \$t0, \$a0, \$a2 # if (\$a0 != \$a2) \$t0 ← 1 # else \$t0 ← 0
li reg, dato	Carga inmediato Carga el dato inmediato en el registro reg. li \$s0, 12 # \$s0 ← 12				

CUADRO RESUMEN DEL LENGUAJE ENSAMBLADOR BÁSICO DEL MIPS-R2000

ALMACENAMIENTO			
sw rt, dirección	Almacena palabra	I	
Almacena el contenido del registro rt en la palabra de memoria indicada por dirección			
sw \$s0, 12(\$a0) # Mem[12 + \$a0] ← \$s0			
sb rt, dirección	Almacena byte	I	
Almacena el LSB del registro en el byte de memoria indicado por dirección			
sb \$s0, 12(\$a0) # Mem[12 + \$a0] ← \$s0(7..0)			
sh rt, dirección	Almacena media palabra	I	
Almacena en los 16 bits de menos peso del registro en la media palabra de memoria indicada por dirección.			
sh \$s0, 12(\$a0) # Mem[12 + \$a0] ← \$s0(15..0)			

LÓGICAS			
and rd, rs, rt	AND entre registros	R	
Operación AND bit a bit entre los registros rs y rt. El resultado se almacena en rd			
and \$t0, \$a0, \$a1 # \$t0 ← \$a0 & \$a1			
andi rt, rs, imm	AND con inmediato	I	
Operación AND bit a bit entre el dato inmediato, extendiendo ceros, y el registro rs. El resultado se almacena en rt.			
andi \$t0, \$a0, 0xA1FF # \$t0 ← \$a0 & (0x0000A1FF)			
or rd, rs, rt	OR entre registros	R	
Operación OR bit a bit entre los registros rs y rt. El resultado se almacena en rd			
or \$t0, \$a0, \$a1 # \$t0 ← \$a0 \$a1			
ori rt, rs, imm	OR con inmediato	I	
Operación OR bit a bit entre el dato inmediato, extendiendo ceros, y el registro rs. El resultado se almacena en rt.			
ori \$t0, \$a0, 0xA1FF # \$t0 ← \$a0 (0x0000A1FF)			
MOVIMIENTO ENTRE REGISTROS			
mfihi rd	mueve desde HI	R	
Transfiere el contenido del registro HI al registro rd.			
mfini \$t0	# \$t0 ← HI		
mfiio rd	mueve desde LO	R	
Transfiere el contenido del registro LO al registro rd.			
mfiio \$t1	# \$t1 ← LO		

FORMATO DE LAS INSTRUCCIONES					
tipo R					
inst(6 bits)	rs(5bits)	rt (5bits)	rd (5bits)	shamt(5bits)	co (6bits)
tipo I					
inst(6bits)	rs(5bits)	rt(5bits)	imm(16 bits)		
tipo J					
inst(6bits)	objetivo (26 bits)				

DESPLAZAMIENTO			
sll rd, rt, shamt	Desplazamiento logico a la izquierda	R	
Desplaza el registro rt a la izquierda tantos bits como indica shamt			
sll \$t0, \$t1, 16 # \$t0 ← \$t1 << 16			
srl rd, rt, shamt	Desplazamiento logico a la derecha	R	
Desplaza el registro rt a la derecha tantos bits como indica shamt.			
srl \$s0, \$t1, 4 # \$s0 ← \$t1 >> 4			
sra rd, rt, shamt	Desplaz. aritmético a la derecha	R	
Desplaza el registro rt a la derecha tantos bits como indica shamt. Los bits MSB toman el mismo valor que el bit de signo de rt. El resultado se almacena en rd			
sra \$s0, \$t1, 4 # \$s0 ← \$t1 >> 4			
	# \$s0(31..28) ← \$t1(31)		

SALTOS INCONDICIONALES			
j dirección	Salto incondicional	J	
Salta a la instrucción apuntada por la etiqueta dirección			
j finbucle	# \$pc ← dirección etiqueta finbucle		
j al dirección	Salta y enlazar	J	
Salta a la instrucción apuntada por la etiqueta dirección y almacena la dirección de la instrucción siguiente en \$ra			
j al rutina	# \$pc ← dirección etiqueta rutina		
	# \$ra ← dirección siguiente instrucción		
jr rs	Salta a registro	R	
Salta a la instrucción apuntada por el contenido del registro rs.			
jr \$ra	# \$pc ← \$ra		

SALTOS CONDICIONALES			
beq rs, rt, etiqueta	Salto si igual	I	
Salta a etiqueta si rs es igual a rt			
beq \$t0, \$t1, DIR # if (\$t0=\$t1) \$pc ← DIR			
bgez rs, etiqueta	Salto si mayor o igual que cero	I	
Salta a etiqueta si rs es mayor o igual que 0			
bgez \$t0, SLT # if (\$t0>=0) \$pc ← SLT			
bgtz rs, etiqueta	Salto si mayor que cero	I	
Salta a etiqueta si rs es mayor que 0			
bgtz \$t0, SLT # if (\$t0>0) \$pc ← SLT			
blez rs, etiqueta	Salto si menor o igual que cero	I	
Salta a etiqueta si rs es menor o igual que 0			
blez \$t1, ETQ # if (\$t1<=0) \$pc ← ETQ			
bltz rs, etiqueta	Salto si menor que cero	I	
Salta a etiqueta si rs es menor que 0			
bltz \$t1, ETQ # if (\$t1<0) \$pc ← ETQ			
bne rs, rt, etiqueta	Salto si distinto	I	
Salta a etiqueta si rs es diferente de rt			
bne \$t0, \$t1, DIR # if (\$t0<=\$t1) \$pc ← DIR			
bge reg1, reg2, etiq	Salto mayor o igual	PS	
Salta a etiq si reg1 es mayor o igual que reg2			
bge \$t0, \$t1, DIR # if (\$t0>=\$t1) \$pc ← DIR			
bgt reg1, reg2, etiq	Salto mayor	PS	
Salta a etiq si reg1 es mayor que reg2			
bgt \$t0, \$t1, DIR # if (\$t0>\$t1) \$pc ← DIR			
ble reg1, reg2, etiq	Salto menor o igual	PS	
Salta a etiq si reg1 es menor o igual que reg2			
ble \$t0, \$t1, DIR # if (\$t0<=\$t1) \$pc ← DIR			
bit reg1, reg2, etiq	Salto menor	PS	
Salta a etiq si reg1 es menor que reg2			
bit \$t0, \$t1, DIR # if (\$t0<\$t1) \$pc ← DIR			

MIPS32® Instruction Set

Quick Reference

- Rd** — DESTINATION REGISTER
- Rs, Rt** — SOURCE OPERAND REGISTERS
- RA** — RETURN ADDRESS REGISTER (R31)
- PC** — PROGRAM COUNTER
- ACC** — 64-BIT ACCUMULATOR
- Lo, Hi** — ACCUMULATOR LOW (ACC_{LO}) AND HIGH (ACC_{HI}) PARTS
- ±** — SIGNED OPERAND OR SIGN EXTENSION
- ∅** — UNSIGNED OPERAND OR ZERO EXTENSION
- ::** — CONCATENATION OF BIT FIELDS
- R2** — MIPS32 RELEASE 2 INSTRUCTION
- DOTTED** — ASSEMBLER PSEUDO-INSTRUCTION

PLEASE REFER TO “MIPS32 Architecture For Programmers Volume II: The MIPS32 Instruction Set” FOR COMPLETE INSTRUCTION SET INFORMATION.

Arithmetic Operations	
ADD	Rd, Rs, Rt Rd = Rs + Rt (OVERFLOW TRAP)
ADDI	Rd, Rs, CONST16 Rd = Rs + CONST16 [±] (OVERFLOW TRAP)
ADDIU	Rd, Rs, CONST16 Rd = Rs + CONST16 [±]
ADDU	Rd, Rs, Rt Rd = Rs + Rt
CLO	Rd, Rs Rd = COUNTLEADINGONES(Rs)
CLZ	Rd, Rs Rd = COUNTLEADINGZEROS(Rs)
LA	Rd, LABEL Rd = ADDRESS(LABEL)
LI	Rd, IMM32 Rd = IMM32
LIUI	Rd, CONST16 Rd = CONST16 << 16
MOVE	Rd, Rs Rd = Rs
NEGU	Rd, Rs Rd = −Rs
SEB _{R2}	Rd, Rs Rd = Rs ₇₀ [±]
SEH _{R2}	Rd, Rs Rd = Rs ₁₅₀ [±]
SUB	Rd, Rs, Rt Rd = Rs − Rt (OVERFLOW TRAP)
SUBU	Rd, Rs, Rt Rd = Rs − Rt

Shift And Rotate Operations	
ROTR _{R2}	Rd, Rs, BITS5 Rd = R _{BITS5:10} :: R _{31:BITS5}
ROTRV _{R2}	Rd, Rs, Rt Rd = R _{BITS10:10} :: R _{31:R_{RT10}}
SLL	Rd, Rs, SHIFT5 Rd = Rs << SHIFT5
SLLV	Rd, Rs, Rt Rd = Rs << R _{T10}
SRA	Rd, Rs, SHIFT5 Rd = Rs [±] >> SHIFT5
SRAV	Rd, Rs, Rt Rd = Rs [±] >> R _{T10}
SRL	Rd, Rs, SHIFT5 Rd = Rs [∅] >> SHIFT5
SRLV	Rd, Rs, Rt Rd = Rs [∅] >> R _{T10}

Logical And Bit-Field Operations	
AND	Rd, Rs, Rt Rd = Rs & Rt
ANDI	Rd, Rs, CONST16 Rd = Rs & CONST16 [∅]
EXT _{R2}	Rd, Rs, P, S Rs = R _{S_P:S+1:P} [∅]
INS _{R2}	Rd, Rs, P, S R _{D_P:S+1:P} = R _{S_P:1:0}
NO _R	No-OP
NOR	Rd, Rs, Rt Rd = ~(Rs Rt)
NOT	Rd, Rs Rd = ~Rs
OR	Rd, Rs, Rt Rd = Rs Rt
ORI	Rd, Rs, CONST16 Rd = Rs CONST16 [∅]
WSBH _{R2}	Rd, Rs Rd = R _{S_{23:16}} :: R _{S_{13:4}} :: R _{S_{7:0}} :: R _{S_{15:8}}
XOR	Rd, Rs, Rt Rd = Rs ⊕ Rt
XORI	Rd, Rs, CONST16 Rd = Rs ⊕ CONST16 [∅]

Condition Testing And Conditional Move Operations	
MOVN	Rd, Rs, Rt IF Rt ≠ 0, Rd = Rs
MOVZ	Rd, Rs, Rt IF Rt = 0, Rd = Rs
SLT	Rd, Rs, Rt Rd = (Rs < Rt) ? 1 : 0
SLT _{TI}	Rd, Rs, CONST16 Rd = (Rs [±] < CONST16 [±]) ? 1 : 0
SLT _{TU}	Rd, Rs, CONST16 Rd = (Rs [∅] < CONST16 [∅]) ? 1 : 0
SLT _{TU}	Rd, Rs, Rt Rd = (Rs [∅] < R _{T7}) ? 1 : 0

Multiply And Divide Operations	
DIV	Rs, Rt L ₀ = Rs [±] / R _{T[±]} ; H ₁ = Rs [±] MOD R _{T[±]}
DIVU	Rs, Rt L ₀ = Rs [∅] / R _{T[∅]} ; H ₁ = Rs [∅] MOD R _{T[∅]}
MADD	Rs, Rt ACC += Rs [±] × R _{T[±]}
MADDU	Rs, Rt ACC += Rs [∅] × R _{T[∅]}
MSUB	Rs, Rt ACC −= Rs [±] × R _{T[±]}
MSUBU	Rs, Rt ACC −= Rs [∅] × R _{T[∅]}
MUL	Rd, Rs, Rt Rd = Rs [±] × R _{T[±]}
MULT	Rs, Rt ACC = Rs [±] × R _{T[±]}
MULTU	Rs, Rt ACC = Rs [∅] × R _{T[∅]}

Accumulator Access Operations	
MFFH	Rd Rd = H ₁
MFFO	Rd Rd = L ₀
MTHI	Rs H ₁ = Rs
MTLO	Rs L ₀ = Rs

Jumps And Branches (Note: One Delay Slot)	
B	OFF18 PC += OFF18 [±]
BAL	OFF18 RA = PC + 8, PC += OFF18 [±]
BEQ	Rs, Rt, OFF18 IF Rs = Rt, PC += OFF18 [±]
BEOZ	Rs, OFF18 IF Rs = 0, PC += OFF18 [±]
BGEZ	Rs, OFF18 IF Rs ≥ 0, PC += OFF18 [±]
BGEZAL	Rs, OFF18 RA = PC + 8, IF Rs ≥ 0, PC += OFF18 [±]
BGTZ	Rs, OFF18 IF Rs > 0, PC += OFF18 [±]
BLEZ	Rs, OFF18 IF Rs ≤ 0, PC += OFF18 [±]
BLTZ	Rs, OFF18 IF Rs < 0, PC += OFF18 [±]
BLTZAL	Rs, OFF18 RA = PC + 8, IF Rs < 0, PC += OFF18 [±]
BNE	Rs, Rt, OFF18 IF Rs ≠ Rt, PC += OFF18 [±]
BNEZ	Rs, OFF18 IF Rs ≠ 0, PC += OFF18 [±]
J	ADDR28 PC = PC _{31:28} :: ADDR28 [∅]
JAL	ADDR28 RA = PC + 8, PC = PC _{31:28} :: ADDR28 [∅]
JALR	Rd, Rs Rd = PC + 8, PC = Rs
JR	Rs PC = Rs

Load And Store Operations	
LB	Rd, OFF16(Rs) Rd = MEM8(Rs + OFF16 [±]) [±]
LBU	Rd, OFF16(Rs) Rd = MEM8(Rs + OFF16 [±]) [∅]
LH	Rd, OFF16(Rs) Rd = MEM16(Rs + OFF16 [±]) [±]
LHU	Rd, OFF16(Rs) Rd = MEM16(Rs + OFF16 [±]) [∅]
LW	Rd, OFF16(Rs) Rd = MEM32(Rs + OFF16 [±]) [±]
LWL	Rd, OFF16(Rs) Rd = LoadWordLEFT(Rs + OFF16 [±])
LWR	Rd, OFF16(Rs) Rd = LoadWordRIGHT(Rs + OFF16 [±])
SB	Rs, OFF16(Rt) MEM8(Rt + OFF16 [±]) = Rs _{7:0}
SH	Rs, OFF16(Rt) MEM16(Rt + OFF16 [±]) = Rs _{15:0}
SW	Rs, OFF16(Rt) MEM32(Rt + OFF16 [±]) = Rs
SWL	Rs, OFF16(Rt) StoreWordLEFT(Rt + OFF16 [±] , Rs)
SWR	Rs, OFF16(Rt) StoreWordRIGHT(Rt + OFF16 [±] , Rs)
L _U W	Rd, OFF16(Rs) Rd = UNALIGNED_MEM32(Rs + OFF16 [±])
L _U SW	Rs, OFF16(Rt) UNALIGNED_MEM32(Rt + OFF16 [±]) = Rs

Atomic Read-Multiply-Write Operations	
LL	Rd, OFF16(Rs) Rd = MEM32(Rs + OFF16 [±]); LINK
SC	Rd, OFF16(Rs) IF ATOMIC, MEM32(Rs + OFF16 [±]) = Rd; Rd = ATOMIC ? 1 : 0

REGISTERS		
0	zero	Always equal to zero
1	at	Assembler temporary; used by the assembler
2-3	v0-v1	Return value from a function call
4-7	a0-a3	First four parameters for a function call
8-15	t0-t7	Temporary variables; need not be preserved
16-23	s0-s7	Function variables; must be preserved
24-25	t8-t9	Two more temporary variables
26-27	k0-k1	Kernel use registers; may change unexpectedly
28	gp	Global pointer
29	sp	Stack pointer
30	fp/s8	Stack frame pointer or subroutine variable
31	ra	Return address of the last subroutine call

DEFAULT C CALLING CONVENTION (O32)	
Stack Management	
<ul style="list-style-type: none"> The stack grows down. Subtract from \$sp to allocate local storage space. Restore \$sp by adding the same amount at function exit. The stack must be 8-byte aligned. Modify \$sp only in multiples of eight. 	
Function Parameters	
<ul style="list-style-type: none"> Every parameter smaller than 32 bits is promoted to 32 bits. First four parameters are passed in registers \$a0–\$a3. 64-bit parameters are passed in register pairs: <ul style="list-style-type: none"> Little-endian mode: \$a1:\$a0 or \$a3:\$a2. Big-endian mode: \$a0:\$a1 or \$a2:\$a3. Every subsequent parameter is passed through the stack. First 16 bytes on the stack are not used. Assuming \$sp was not modified at function entry: <ul style="list-style-type: none"> The 1st stack parameter is located at 16(\$sp). The 2nd stack parameter is located at 20(\$sp), etc. 64-bit parameters are 8-byte aligned. 	
Return Values	
<ul style="list-style-type: none"> 32-bit and smaller values are returned in register \$v0. 64-bit values are returned in registers \$v0 and \$v1: Little-endian mode: \$v1:\$v0. Big-endian mode: \$v0:\$v1. 	

MIPS32 Virtual Address Space				
kseg3	0xE000.0000	0xFFFF.FFFF	Mapped	Cached
kseg	0xC000.0000	0xDFFF.FFFF	Mapped	Cached
kseg1	0xA000.0000	0xBFFF.FFFF	Unmapped	Uncached
kseg0	0x8000.0000	0x9FFF.FFFF	Unmapped	Cached
useg	0x0000.0000	0x7FFF.FFFF	Mapped	Cached

READING THE CYCLE COUNT REGISTER FROM C
<pre>unsigned mips_cycle_counter_read() { unsigned cc; asm volatile("mfcc0 %0, \$9" : "=r" (cc)); return (cc << 1); }</pre>

ASSEMBLY-LANGUAGE FUNCTION EXAMPLE
<pre># int asm_max(int a, int b) # { # int r = (a < b) ? b : a; # return r; # } .text .set nomacro .set noreorder .global asm_max .ent asm_max asm_max: move \$v0, \$a0 # r = a slt \$t0, \$a0, \$a1 # a < b ? jr \$ra # return movn \$v0, \$a1, \$t0 # if yes, r = b .end asm_max</pre>

C / ASSEMBLY-LANGUAGE FUNCTION INTERFACE
<pre>#include <stdio.h> int asm_max(int a, int b); int main() { int x = asm_max(10, 100); int y = asm_max(200, 20); printf("%d %d\n", x, y); }</pre>

INVOKING MULT AND MADD INSTRUCTIONS FROM C
<pre>int dp(int a[], int b[], int n) { int i; long long acc = (long long) a[0] * b[0]; for (i = 1; i < n; i++) acc += (long long) a[i] * b[i]; return (acc >> 31); }</pre>

ATOMIC READ-MODIFY-WRITE EXAMPLE
<pre>atomic_inc: ll \$t0, 0(\$a0) # load linked addiu \$t1, \$t0, 1 # increment sc \$t1, 0(\$a0) # store cond'l beqz \$t1, atomic_inc # loop if failed nop</pre>

ACCESSING UNALIGNED DATA	
NOTE: ULW AND USW AUTOMATICALLY GENERATE APPROPRIATE CODE	
LITTLE-ENDIAN MODE	BIG-ENDIAN MODE
LWR Rd, 0xFF16(Rs)	LWL Rd, 0xFF16(Rs)
LWL Rd, 0xFF16+3(Rs)	LWR Rd, 0xFF16+3(Rs)
SWR Rd, 0xFF16(Rs)	SWL Rd, 0xFF16(Rs)
SWL Rd, 0xFF16+3(Rs)	SWR Rd, 0xFF16+3(Rs)

ACCESSING UNALIGNED DATA FROM C
<pre>typedef struct { int u; } __attribute__((packed)) unaligned; int unaligned_load(void *ptr) { unaligned *uptr = (unaligned *)ptr; return uptr->u; }</pre>

MIPS SDE-GCC COMPILER DEFINES	
__mips	MIPS ISA (= 32 for MIPS32)
__mips_isa_rev	MIPS ISA Revision (= 2 for MIPS32 R2)
__mips_dsp	DSP ASE extensions enabled
_MIPSEB	Big-endian target CPU
_MIPSEL	Little-endian target CPU
_MIPS_ARCH_CPU	Target CPU specified by -march=CPU
_MIPS_TUNE_CPU	Pipeline tuning selected by -mtune=CPU

NOTES
<ul style="list-style-type: none"> Many assembler pseudo-instructions and some rarely used machine instructions are omitted. The C calling convention is simplified. Additional rules apply when passing complex data structures as function parameters. The examples illustrate syntax used by GCC compilers. Most MIPS processors increment the cycle counter every other cycle. Please check your processor documentation.

MIPS Instruction Reference

(<http://www.mrc.uidaho.edu/mrc/people/jff/digital/MIPSir.html>)

Esta es una descripción del conjunto de instrucciones MIPS, su significado, sintaxis, semántica y codificación. La sintaxis dada para cada instrucción hace referencia a la sintaxis del lenguaje ensamblador soportado por el ensamblador MIPS. Guiones en la descripción de la codificación indican que esos bits no serán considerados en la decodificación (no importa su valor).

Los registros de propósito general (GPRs) se indican con el signo del dolar (\$). Las palabras SWORD y UWORD hacen referencia a tipos de datos de 32-bits con signo y sin signo respectivamente.

La función `advance_pc (int)` se usa para indicar el nuevo valor de PC tras la ejecución de la instrucción. La función se define como:

```
void advance_pc (SWORD offset)
{
    PC    = nPC;
    nPC   += offset;
}
```

A continuación presentamos la descripción del conjunto de instrucciones:

ADD – Add (with overflow)

Description:	Adds two registers and stores the result in a register
Operation:	\$d = \$s + \$t; advance_pc (4);
Syntax:	add \$d, \$s, \$t
Encoding:	0000 00ss ssst tttt dddd d000 0010 0000

ADDI -- Add immediate (with overflow)

Description:	Adds a register and a sign-extended immediate value and stores the result in a register
Operation:	\$t = \$s + imm; advance_pc (4);
Syntax:	addi \$t, \$s, imm
Encoding:	0010 00ss ssst tttt iiii iiii iiii iiii

ADDIU -- Add immediate unsigned (no overflow)

Description:	Adds a register and a sign-extended immediate value and stores the result in a register
Operation:	\$t = \$s + imm; advance_pc (4);
Syntax:	addiu \$t, \$s, imm

Encoding:	0010 01ss ssst tttt iiii iiii iiii iiii
-----------	---

ADDU -- Add unsigned (no overflow)

Description:	Adds two registers and stores the result in a register
Operation:	\$d = \$s + \$t; advance_pc (4);
Syntax:	addu \$d, \$s, \$t
Encoding:	0000 00ss ssst tttt dddd d000 0010 0001

AND -- Bitwise and

Description:	Bitwise ands two registers and stores the result in a register
Operation:	\$d = \$s & \$t; advance_pc (4);
Syntax:	and \$d, \$s, \$t
Encoding:	0000 00ss ssst tttt dddd d000 0010 0100

ANDI -- Bitwise and immediate

Description:	Bitwise ands a register and an immediate value and stores the result in a register
Operation:	\$t = \$s & imm; advance_pc (4);
Syntax:	andi \$t, \$s, imm
Encoding:	0011 00ss ssst tttt iiii iiii iiii iiii

BEQ -- Branch on equal

Description:	Branches if the two registers are equal
Operation:	if \$s == \$t advance_pc (offset << 2)); else advance_pc (4);
Syntax:	beq \$s, \$t, offset
Encoding:	0001 00ss ssst tttt iiii iiii iiii iiii

BGEZ -- Branch on greater than or equal to zero

Description:	Branches if the register is greater than or equal to zero
Operation:	if \$s >= 0 advance_pc (offset << 2)); else advance_pc (4);
Syntax:	bgez \$s, offset
Encoding:	0000 01ss sss0 0001 iiii iiii iiii iiii

BGEZAL -- Branch on greater than or equal to zero and link

Description:	Branches if the register is greater than or equal to zero and saves the return address in \$31
Operation:	if \$s >= 0 \$31 = PC + 8 (or nPC + 4); advance_pc (offset << 2)); else advance_pc (4);
Syntax:	bgezal \$s, offset
Encoding:	0000 01ss sss1 0001 iiii iiii iiii iiii

BGTZ -- Branch on greater than zero

Description:	Branches if the register is greater than zero
Operation:	if \$s > 0 advance_pc (offset << 2)); else advance_pc (4);
Syntax:	bgtz \$s, offset
Encoding:	0001 11ss sss0 0000 iiii iiii iiii iiii

BLEZ -- Branch on less than or equal to zero

Description:	Branches if the register is less than or equal to zero
Operation:	if \$s <= 0 advance_pc (offset << 2)); else advance_pc (4);
Syntax:	blez \$s, offset
Encoding:	0001 10ss sss0 0000 iiii iiii iiii iiii

BLTZ -- Branch on less than zero

Description:	Branches if the register is less than zero
Operation:	if \$s < 0 advance_pc (offset << 2)); else advance_pc (4);
Syntax:	bltz \$s, offset
Encoding:	0000 01ss sss0 0000 iiii iiii iiii iiii

BLTZAL -- Branch on less than zero and link

Description:	Branches if the register is less than zero and saves the return address in \$31
Operation:	if \$s < 0 \$31 = PC + 8 (or nPC + 4); advance_pc (offset << 2)); else advance_pc (4);
Syntax:	bltzal \$s, offset
Encoding:	0000 01ss sss1 0000 iiii iiii iiii iiii

BNE -- Branch on not equal

Description:	Branches if the two registers are not equal
--------------	---

Operation:	if \$s != \$t advance_pc (offset << 2)); else advance_pc (4);
Syntax:	bne \$s, \$t, offset
Encoding:	0001 01ss ssst tttt iiii iiii iiii iiii

DIV -- Divide

Description:	Divides \$s by \$t and stores the quotient in \$LO and the remainder in \$HI
Operation:	\$LO = \$s / \$t; \$HI = \$s % \$t; advance_pc (4);
Syntax:	div \$s, \$t
Encoding:	0000 00ss ssst tttt 0000 0000 0001 1010

DIVU -- Divide unsigned

Description:	Divides \$s by \$t and stores the quotient in \$LO and the remainder in \$HI
Operation:	\$LO = \$s / \$t; \$HI = \$s % \$t; advance_pc (4);
Syntax:	divu \$s, \$t
Encoding:	0000 00ss ssst tttt 0000 0000 0001 1011

J -- Jump

Description:	Jumps to the calculated address
Operation:	PC = nPC; nPC = (PC & 0xf0000000) (target << 2);
Syntax:	j target
Encoding:	0000 10ii iiii iiii iiii iiii iiii iiii

JAL -- Jump and link

Description:	Jumps to the calculated address and stores the return address in \$31
Operation:	\$31 = PC + 8 (or nPC + 4); PC = nPC; nPC = (PC & 0xf0000000) (target << 2);
Syntax:	jal target
Encoding:	0000 11ii iiii iiii iiii iiii iiii iiii

JR -- Jump register

Description:	Jump to the address contained in register \$s
Operation:	PC = nPC; nPC = \$s;
Syntax:	jr \$s

Encoding:	0000 00ss sss0 0000 0000 0000 0000 1000
-----------	---

LB -- Load byte

Description:	A byte is loaded into a register from the specified address.
Operation:	\$t = MEM[\$s + offset]; advance_pc (4);
Syntax:	lb \$t, offset(\$s)
Encoding:	1000 00ss ssst tttt iiii iiii iiii iiii

LUI -- Load upper immediate

Description:	The immediate value is shifted left 16 bits and stored in the register. The lower 16 bits are zeroes.
Operation:	\$t = (imm << 16); advance_pc (4);
Syntax:	lui \$t, imm
Encoding:	0011 11-- ---t tttt iiii iiii iiii iiii

LW -- Load word

Description:	A word is loaded into a register from the specified address.
Operation:	\$t = MEM[\$s + offset]; advance_pc (4);
Syntax:	lw \$t, offset(\$s)
Encoding:	1000 11ss ssst tttt iiii iiii iiii iiii

MFHI -- Move from HI

Description:	The contents of register HI are moved to the specified register.
Operation:	\$d = \$HI; advance_pc (4);
Syntax:	mfhi \$d
Encoding:	0000 0000 0000 0000 dddd d000 0001 0000

MFLO -- Move from LO

Description:	The contents of register LO are moved to the specified register.
Operation:	\$d = \$LO; advance_pc (4);
Syntax:	mflo \$d
Encoding:	0000 0000 0000 0000 dddd d000 0001 0010

MULT -- Multiply

Description:	Multiplies \$s by \$t and stores the result in \$LO.
Operation:	\$LO = \$s * \$t; advance_pc (4);
Syntax:	mult \$s, \$t
Encoding:	0000 00ss ssst tttt 0000 0000 0001 1000

MULTU -- Multiply unsigned

Description:	Multiplies \$s by \$t and stores the result in \$LO.
Operation:	\$LO = \$s * \$t; advance_pc (4);
Syntax:	multu \$s, \$t
Encoding:	0000 00ss ssst tttt 0000 0000 0001 1001

NOOP -- no operation

Description:	Performs no operation.
Operation:	advance_pc (4);
Syntax:	noop
Encoding:	0000 0000 0000 0000 0000 0000 0000 0000

Note: The encoding for a NOOP represents the instruction SLL \$0, \$0, 0 which has no side effects. In fact, nearly every instruction that has \$0 as its destination register will have no side effect and can thus be considered a NOOP instruction.

OR -- Bitwise or

Description:	Bitwise logical ors two registers and stores the result in a register
Operation:	\$d = \$s \$t; advance_pc (4);
Syntax:	or \$d, \$s, \$t
Encoding:	0000 00ss ssst tttt dddd d000 0010 0101

ORI -- Bitwise or immediate

Description:	Bitwise ors a register and an immediate value and stores the result in a register
Operation:	\$t = \$s imm; advance_pc (4);
Syntax:	ori \$t, \$s, imm
Encoding:	0011 01ss ssst tttt iiii iiii iiii iiii

SB -- Store byte

Description:	The least significant byte of \$t is stored at the specified address.
Operation:	MEM[\$s + offset] = (0xff & \$t); advance_pc (4);
Syntax:	sb \$t, offset(\$s)
Encoding:	1010 00ss ssst tttt iiii iiii iiii iiii

SLL -- Shift left logical

Description:	Shifts a register value left by the shift amount listed in the instruction and places the result in a third register. Zeroes are shifted in.
Operation:	\$d = \$t << h; advance_pc (4);
Syntax:	sll \$d, \$t, h
Encoding:	0000 00ss ssst tttt dddd dhhh hh00 0000

SLLV -- Shift left logical variable

Description:	Shifts a register value left by the value in a second register and places the result in a third register. Zeroes are shifted in.
Operation:	\$d = \$t << \$s; advance_pc (4);
Syntax:	sllv \$d, \$t, \$s
Encoding:	0000 00ss ssst tttt dddd d--- --00 0100

SLT -- Set on less than (signed)

Description:	If \$s is less than \$t, \$d is set to one. It gets zero otherwise.
Operation:	if \$s < \$t \$d = 1; advance_pc (4); else \$d = 0; advance_pc (4);
Syntax:	slt \$d, \$s, \$t
Encoding:	0000 00ss ssst tttt dddd d000 0010 1010

SLTI -- Set on less than immediate (signed)

Description:	If \$s is less than immediate, \$t is set to one. It gets zero otherwise.
Operation:	if \$s < imm \$t = 1; advance_pc (4); else \$t = 0; advance_pc (4);
Syntax:	slti \$t, \$s, imm
Encoding:	0010 10ss ssst tttt iiii iiii iiii iiii

SLTIU -- Set on less than immediate unsigned

Description:	If \$s is less than the unsigned immediate, \$t is set to one. It gets zero otherwise.
Operation:	if \$s < imm \$t = 1; advance_pc (4); else \$t = 0; advance_pc (4);
Syntax:	sltiu \$t, \$s, imm
Encoding:	0010 11ss ssst tttt iiii iiii iiii iiii

SLTU -- Set on less than unsigned

Description:	If \$s is less than \$t, \$d is set to one. It gets zero otherwise.
Operation:	if \$s < \$t \$d = 1; advance_pc (4); else \$d = 0; advance_pc (4);
Syntax:	sltu \$d, \$s, \$t
Encoding:	0000 00ss ssst tttt dddd d000 0010 1011

SRA -- Shift right arithmetic

Description:	Shifts a register value right by the shift amount (shamt) and places the value in the destination register. The sign bit is shifted in.
Operation:	\$d = \$t >> h; advance_pc (4);
Syntax:	sra \$d, \$t, h
Encoding:	0000 00-- ---t tttt dddd dhhh hh00 0011

SRL -- Shift right logical

Description:	Shifts a register value right by the shift amount (shamt) and places the value in the destination register. Zeroes are shifted in.
Operation:	\$d = \$t >> h; advance_pc (4);
Syntax:	srl \$d, \$t, h
Encoding:	0000 00-- ---t tttt dddd dhhh hh00 0010

SRLV -- Shift right logical variable

Description:	Shifts a register value right by the amount specified in \$s and places the value in the destination register. Zeroes are shifted in.
Operation:	\$d = \$t >> \$s; advance_pc (4);
Syntax:	srlv \$d, \$t, \$s
Encoding:	0000 00ss ssst tttt dddd d000 0000 0110

SUB -- *Subtract*

Description:	Subtracts two registers and stores the result in a register
Operation:	\$d = \$s - \$t; advance_pc (4);
Syntax:	sub \$d, \$s, \$t
Encoding:	0000 00ss ssst tttt dddd d000 0010 0010

SUBU -- *Subtract unsigned*

Description:	Subtracts two registers and stores the result in a register
Operation:	\$d = \$s - \$t; advance_pc (4);
Syntax:	subu \$d, \$s, \$t
Encoding:	0000 00ss ssst tttt dddd d000 0010 0011

SW -- *Store word*

Description:	The contents of \$t is stored at the specified address.
Operation:	MEM[\$s + offset] = \$t; advance_pc (4);
Syntax:	sw \$t, offset(\$s)
Encoding:	1010 11ss ssst tttt iiii iiii iiii iiii

SYSCALL -- *System call*

Description:	Generates a software interrupt.
Operation:	advance_pc (4);
Syntax:	syscall
Encoding:	0000 00-- ---- ---- ---- ---- --00 1100

The syscall instruction is described in more detail on the [System Calls](#) page.

XOR -- *Bitwise exclusive or*

Description:	Exclusive ors two registers and stores the result in a register
Operation:	\$d = \$s ^ \$t; advance_pc (4);
Syntax:	xor \$d, \$s, \$t
Encoding:	0000 00ss ssst tttt dddd d--- --10 0110

XORI -- *Bitwise exclusive or immediate*

Description:	Bitwise exclusive ors a register and an immediate value and stores the result in a register
Operation:	\$t = \$s ^ imm; advance_pc (4);
Syntax:	xori \$t, \$s, imm
Encoding:	0011 10ss ssst tttt iiii iiii iiii iiii