El Lenguaje Ensamblador MIPS

Tipos de Instrucciones

- **Instrucciones Máquina:** son las que vienen en la tabla de instrucciones. Se traducen a código máquina.
- **Directivas:** palabras clave utilizadas por el compilador. Se caracterizan porque comienzan con un punto.
- **Pseudoinstrucciones:** son instrucciones máquina que no forman parte de la tabla básica de instrucciones. El compilador las traduce por una o más instrucciones máquina. Permiten que el código sea más legible.

  Ejemplo: `move` rd`,` rs $\rightarrow$ `add` rd`,` rs`, $zero`

Etiquetas y Datos Numéricos

Las **etiquetas** consisten en un identificador alfanumérico seguido de dos puntos (:) Representan direcciones de memoria (de las instrucciones (o datos) en la que están situadas). No pueden comenzar por un número ni pueden ser mnemónicos de las propias instrucciones. Los **números** se interpretan en decimal. Para que se interpreten en hexadecimal, deben ir precedidos de `0x`.

Comentarios

Comienzan con **#** y abarcan hasta el final de línea (que actúa, a su vez, como separador de instrucción). Son ignorados por el compilador y se utilizan para facilitar la legibilidad y el flujo de ejecución del programa.

Estructura de un Programa

- **Área de Datos**: comienza con la directiva `.data` y está destinada a la declaración de variables y reserva de espacio en memoria para aquellas que lo necesiten, utilizando para ello las directivas:
  - `.byte`: reserva espacio y lo inicializa con los valores indicados a continuación, cada uno como un entero en un byte.
  - `.half`: reserva espacio y lo inicializa con los valores indicados a continuación, cada uno como un entero en 2 bytes.
  - `.word`: reserva espacio y lo inicializa con los valores indicados a continuación, cada uno como un entero en 4 bytes.
  - `.space`: reserva el número de bytes indicado y los inicializa a 0.
  - `.ascii "`$cadena$`"`: reserva espacio en memoria para la cadena de caracteres especificada, pero no escribe el caracter terminador de cadena al final. Si se quieren incluir caracteres especiales, hay que seguir las mismas convenciones que en el lenguaje C (`\n` es el fin de línea, `\t` es el tabulador, `\"` son las comillas, etc.)
  - `.asciiz "`$cadena$`"`: ídem al anterior, pero incluye el caracter terminador al final.
  - `.float`: reserva espacio y lo inicializa con los valores indicados a continuación, cada uno como un número en representación de punto flotante de simple precisión (32 bits)
  - `.double`: ídem al anterior, pero de doble precisión (64 bits)
- **Área de Código**: comienza con la directiva `.text` y contiene las instrucciones ejecutables. La primera instrucción del programa debe tener la etiqueta `main` (dirección a la que se transfiere el control al iniciar el programa). Para finalizar la ejecución y devolver el control al Sistema Operativo, hay que indicar las instrucciones:

```
li $v0 10
syscall
```

Instrucciones MIPS32

Negro: instrucciones que se comentan en clase de teoría.
Azul: instrucciones útiles (pero no se comentan en clase)
Verde: pseudoinstrucciones.

1. **Movimiento de Datos**:

`lw` $rt,\ d(rs)$	Carga 32 bits en rt desde la dirección $d + rs$
`lw` $rt,\ (rs)$	Carga 32 bits en rt desde la dirección especificada en rs
`lh` $rt,\ d(rs)$	Carga 16 bits en rt desde la dirección $d + rs$ extendiendo signo
`lh` $rt,\ (rs)$	Carga 16 bits en rt desde la dirección especificada en rs extendiendo signo
`lhu` $rt,\ d(rs)$	Carga 16 bits en rt desde la dirección $d + rs$ sin extender signo rellenando con 0's
`lhu` $rt,\ (rs)$	Carga 16 bits en rt desde la dirección dada en rs sin extender signo rellenando con 0's
`lb` $rt,\ d(rs)$	Carga 8 bits en rt desde la dirección $d + rs$ extendiendo signo
`lb` $rt,\ (rs)$	Carga 8 bits en rt desde la dirección especificada en rs extendiendo signo
`lbu` $rt,\ d(rs)$	Carga 8 bits en rt desde la dirección $d + rs$ sin extender signo rellenando con 0's
`lbu` $rt,\ (rs)$	Carga 8 bits en rt desde la dirección dada en rs sin extender signo rellenando con 0's
`sw` $rt,\ d(rs)$	Almacena rt en la dirección $d + rs$
`sw` $rt,\ (rs)$	Almacena rt en la dirección especificada en rs
`sh` $rt,\ d(rs)$	Almacena los 16 bits más bajos de rt en la dirección $d + rs$
`sh` $rt,\ (rs)$	Almacena los 16 bits más bajos de rt en la dirección especificada en rs
`sb` $rt,\ d(rs)$	Almacena los 8 bits más bajos de rt en la dirección $d + rs$
`sb` $rt,\ (rs)$	Almacena los 8 bits más bajos de rt en la dirección especificada en rs
`mfhi` rd	Copia el contenido del registro `HI` en rd
`mflo` rd	Copia el contenido del registro `LO` en rd
`move` $rd,\ rs$	Copia el contenido de rs en rd (`add` $rd,\ rs,$ `$zero`)
`lui` $rd,\ Inm$	Copia el valor Inm de 16 bits en la mitad más alta de rd
`la` $rd,\ Etiqueta$	Carga la dirección representada por $Etiqueta$ en el registro rd
`li` rd, Inm	Carga el valor Inm (de cualquier tamaño) en rd

2. **Aritmética Entera**:

`add` $rd,\ rs,\ rt$	Suma con detección de desbordamiento: $rd \leftarrow rs + rt$
`addi` $rt,\ rs,\ Inm$	Suma inmediata: $rt \leftarrow rs + Inm$ (con signo extendido)
`addu` $rd,\ rs,\ rt$	Suma sin detección de desbordamiento: $rd \leftarrow rs + rt$
`addiu` $rt,\ rs,\ Inm$	Suma inmediata sin detección de desbord.: $rt \leftarrow rs + Inm$ (signo extendido)
`sub` $rd,\ rs,\ rt$	Resta con detección de desbordamiento: $rd \leftarrow rs - rt$
`subu` $rd,\ rs,\ rt$	Resta sin detección de desbordamiento: $rd \leftarrow rs - rt$
`mult` $rs,\ rt$	Multiplicación con signo: `HI:LO` $\leftarrow rs \times rt$
`multu` $rs,\ rt$	Multiplicación sin signo: `HI:LO` $\leftarrow rs \times rt$
`mul` $rd,\ rs,\ rt$	Multiplicación: $rd \leftarrow$ 32 bits más bajos de $rs \times rt$ (extendiendo signo)
`mulu` $rd,\ rs,\ rt$	Multiplicación: $rd \leftarrow$ 32 bits más bajos de $rs \times rt$ (sin extender signo)
`div` $rs,\ rt$	División con signo: rs/rt: `HI` $\leftarrow$ resto; `LO` $\leftarrow$ cociente
`divu` $rs,\ rt$	División sin signo: rs/rt: `HI` $\leftarrow$ resto; `LO` $\leftarrow$ cociente

3. **Lógicas**:

`and` $rd,\ rs,\ rt$	$rd \leftarrow rs \wedge rt$, bit a bit
`andi` $rt,\ rs,\ Inm$	$rt \leftarrow rs \wedge Inm$ (inmediato de 16 bits sin extensión de signo)
`or` $rd,\ rs,\ rt$	$rd \leftarrow rs \vee rt$, bit a bit
`ori` $rt,\ rs,\ Inm$	$rt \leftarrow rs \vee Inm$ (inmediato de 16 bits sin extensión de signo)
`xor` $rd,\ rs,\ rt$	$rd \leftarrow rs \oplus rt$
`xori` $rt,\ rs,\ Inm$	$rt \leftarrow rs \oplus Inm$ (inmediato de 16 bits sin extensión de signo)
`nor` $rd,\ rs,\ rt$	$rd \leftarrow !(rs \vee rt)$

4. **Desplazamiento**:

`sll` $rd,\ rt,\ shamt$	$rd \leftarrow rt << shamt$
`srl` $rd,\ rt,\ shamt$	$rd \leftarrow rt >> shamt$

5. **Set Condicional**:

`slt` $rd,\ rs,\ rt$	Si $(rs < rt)\ rd \leftarrow 1$ si no $rt \leftarrow 0$ (comparación con signo)
`slti` $rt,\ rs,\ Inm$	Si $(rs < Inm)\ rd \leftarrow 1$ si no $rt \leftarrow 0$ (comparación con signo extendido)
`sltu` $rd,\ rs,\ rt$	Si $(rs < rt)\ rd \leftarrow 1$ si no $rt \leftarrow 0$ (comparación sin signo)
`sltui` $rt,\ rs,\ Inm$	Si $(rs < Inm)\ rd \leftarrow 1$ si no $rt \leftarrow 0$ (comparación sin signo)

6. **Salto**:

`beq` $rs,\ rt,\ Etiqueta$	Si $(rs == rt)$ continúa la ejecución en la instrucción situada en $Etiqueta$ si no, continúa en la instrucción siguiente
`bne` $rs,\ rt,\ Etiqueta$	Si $(rs\ != rt)$ continúa la ejecución en la instrucción situada en $Etiqueta$ si no continúa en la instrucción siguiente
`blt` $rs,\ rt,\ Etiqueta$	Si $(rs < rt)$ continúa la ejecución en la instrucción situada en $Etiqueta$ si no, continúa en la instrucción siguiente
`b` $Etiqueta$	Salta incondicionalmente a $Etiqueta$ (direccionamiento relativo)
`j` $Etiqueta$	Salta incondicionalmente a $Etiqueta$ (direccionamiento pseudoabsoluto)
`jal` $Etiqueta$	`$ra` $\leftarrow$ dirección de la siguiente instrucción y salta incondicionalmente a $Etiqueta$
`jr` rs	Salta incondicionalmente a la dirección contenida en el registro rs

7. **Punto Flotante**:

a) Aritméticas:

`add.s` $fd,\ fs,\ ft$	$fd \leftarrow fs + ft$ (simple precisión)
`sub.s` $fd,\ fs,\ ft$	$fd \leftarrow fs - ft$ (simple precisión)
`mul.s` $fd,\ fs,\ ft$	$fd \leftarrow fs \times ft$ (simple precisiónn)
`div.s` $fd,\ fs,\ ft$	$fd \leftarrow fs/ft$ (simple precisión)
`neg.s` fd, fs	$fd \leftarrow -fs$ (simple precisión)
`abs.s` $fd,\ fs$	$fd \leftarrow \|fs\|$ (simple precisión)
`sqrt.s` $fd,\ fs$	$fd \leftarrow \sqrt{fs}$ (simple precisión)
`trunc.w.s` $fd,\ fs$	$fd \leftarrow$ Truncamiento a entero del valor almacenado en fs
`add.d` $fd,\ fs,\ ft$	$fd \leftarrow fs + ft$ (doble precisión)
`sub.d` $fd,\ fs,\ ft$	$fd \leftarrow fs - ft$ (doble precisión)
`mul.d` $fd,\ fs,\ ft$	$fd \leftarrow fs \times ft$ (doble precisión)
`div.d` $fd,\ fs,\ ft$	$fd \leftarrow fs/ft$ (doble precisión)
`neg.d` $fd,\ fs$	$fd \leftarrow -fs$ (doble precisión)
`abs.d` $fd,\ fs$	$fd \leftarrow \|fs\|$ (doble precisión)
`sqrt.d` $fd,\ fs$	$fd \leftarrow \sqrt{fs}$ (doble precisión)
`trunc.w.d` $fd,\ fs$	$fd \leftarrow$ Truncamiento a entero del valor almacenado en el par de registros $fs : fs + 1$

b) Movimiento de Datos:

`mov.s` $fd,\ fs$	$fd \leftarrow fs$
`mov.d` $fd,\ fs$	$fd : fd + 1 \leftarrow fs : fs + 1$
`mfc1` $rd,\ fs$	$rd \leftarrow fs$
`mfc1.d` $rd,\ fs$	$rd : rd + 1 \leftarrow fs : fs + 1$
`mtc1` $rd,\ fs$	$fs \leftarrow rd$

c) Acceso a Memoria:

`lwc1` $fd,\ d(rs)$	Carga 32 bits en el registro FP fd desde la dirección $d + rs$
`lwc1` $fd,\ (rs)$	Carga 32 bits en el registro FP fd desde la dirección especificada en rs
`swc1` $fd,\ d(rs)$	Almacena en la dirección $d + rs$ los 32 bits del registro FP fd
`swc1` $fd,\ (rs)$	Almacena en la dirección especificada en rs los 32 bits del registro FP fd
`ldc1` $fd,\ d(rs)$	Carga 64 bits en los registros FP $fd : fd + 1$ desde la dirección $d + rs$
`ldc1` $fd, (rs)$	Carga 64 bits en los registros FP $fd : fd + 1$ desde la dirección dada en rs
`sdc1` $fd,\ d(rs)$	Almacena en la dirección $d + rs$ los 64 bits de los registros FP $fd : fd + 1$
`sdc1` $fd,\ (rs)$	Almacena en dirección dada en rs los 64 bits de los registros FP $fd : fd+1$

d) Conversión de Datos:

`cvt.d.s` $fd,\ fs$	Convierte el valor FP 32 de fs a FP 64 y lo deja en $fd : fd+1$
`cvt.d.w` $fd,\ fs$	Convierte el valor entero de 32 bits de fs a FP 64 en $fd : fd+1$
`cvt.s.d` $fd,\ fs$	Convierte FP 64 de $fs : fs+1$ a FP 32 en fd
`cvt.s.w` $fd,\ fs$	Convierte entero 32 de fs a FP 32 en fd
`cvt.w.d` $fd,\ fs$	Convierte de FP 64 de $fs : fs+1$ a entero 32 en fd
`cvt.w.s` $fd,\ fs$	Convierte de FP 32 de fs a entero 32 en fd

e) Set Condicional:

`c.**.s` $cc\ fs,\ ft$	Compara los registros FP de 32 bits y pone a 1 la bandera de condición cc si se cumple la condición, y si no se cumple, la pone en 0. ** puede ser: "le"(menor o igual), "eq"(igual), "lt"(menor que) Si no se especifica cc se supone que es la bandera de condición 0.
`c.**.d` $cc\ fs,\ ft$	Análoga a la anterior en doble precisión.

f) Bifurcación:

`bc1t` $cc\ Etiqueta$	Salta a $Etiqueta$ si la bandera de condición de punto flotante cc es 1. Si no se especifica cc se supone que es la bandera de condición 0.
`bc1f`$cc\ Etiqueta$	salta a $Etiqueta$ si la bandera de condición de punto flotante cc es 0. Si no se especifica cc se supone que es la bandera de condición 0.

Registros

`$s0-$s7`: Para almacenar valores de variables del código escrito en lenguaje de alto nivel.
`$t0-$t9`: Para almacenamiento temporal de cálculos intermedios.
`$a0-$a3`: Para pasar parámetros a una función.
`$v0-$v1`: Para devolver resultados de una función.
`$0` o `$zero`: Almacena el valor 0. Sólo lectura.
`$sp`: *Stack Pointer*: puntero de cima de la pila.
`$ra`: Almacena la dirección de retorno a la que se vuelve tras invocar a una función.
`$fp`: *Frame Pointer*: almacena la dirección en la que comienza la "trama de función"o "registro de activación", que es la zona de la pila donde se almacenan inicialmente las variables locales de la función que no van a registros, arrays, direcciones de retorno almacenadas, etc. Los accesos a esas posiciones se hacen relativos al valor del registro `$fp`, en el cual se almacena el valor de `$sp` antes de poner en la pila toda la información indicada.
`$gp`: *Global Pointer*: contiene la dirección de la zona de memoria donde se almacenan las variables estáticas, para poder referenciarlas más fácilmente.

Llamadas al Sistema

Permiten realizar operaciones de E/S. Para solicitar una operación, se carga su código en el registro `$v0`, los parámetros en los registros `$a0-$a3` (`$f12` para valores de punto flotante) y, finalmente, se invoca al sistema usando la instrucción `syscall`. Las operaciones que devuelven un resultado lo depositan en el registro `$v0` (`$f0` para resultados de punto flotante). La siguiente tabla resume las principales llamadas al sistema:

Servicio	Código de Llamada	Argumentos	Resultado
print-int	1	`$a0`=entero	
print-float	2	`$f12`=valor en FP32	
print-double	3	`$f12`=valor en FP64	
print-string	4	`$a0`=buffer	
read-int	5		Entero leído (en `$v0`)
read-float	6		FP32 leído (en `$f0`)
read-double	7		FP64 leído (en `$f0`)
read-string	8	`$a0`=buffer, `$a1`=longitud	Almacena la cadena leída a partir de la dirección contenida en `$a0`
exit	10		
print-char	11	`$a0`=caracter	
read-char	12		Caracter leído (en `$v0`)