Conceptos básicos del Lenguaje Ensamblador MIPS

Tipos de instrucciones

En lenguaje ensamblador hay tres tipos de instrucciones:

- Instrucciones máquina
- Directivos
- Pseudoinstrucciones o instrucciones sintéticas

Las **instrucciones máquina** son las que vienen en la tabla de instrucciones y se traducen casi literalmente a código máquina.

Los **directivos** son órdenes dadas al compilador y se caracterizan porque comienzan con un punto.

En un programa también se pueden poner **comentarios**, que son ignorados por el compilador y se ponen para facilitar la legibilidad del programa. En ensamblador de MIPS los comentarios comienzan con #, de forma que lo que esté entre ese símbolo y el final de la línea (que es el separador de instrucción) se ignora.

Las **pseudoinstrucciones o instrucciones sintéticas** son instrucciones similares a las instrucciones máquina pero que no están en la tabla de instrucciones. Es el compilador el que las traduce por una o más instrucciones máquina. Se utilizan para que el código sea más legible.

Ejemplo: `move` *`rd,rs`* se traduce como `add` *`rd, rs`*, `$zero`

Cualquier instrucción puede ir precedida de una etiqueta que es un identificador seguido de : Cada etiqueta representa la dirección de la instrucción o dato que le siga. Los identificadores son una secuencia de caracteres alfanuméricos que no comiencen por un número. Los códigos de las operaciones son palabras reservadas que no pueden ser utilizadas como identificadores.

Los números se interpretan por defecto en base 10. Si un número está precedido por “0x” se interpretara como hexadecimal.

Estructura de un programa en lenguaje ensamblador

Un programa tiene dos áreas bien diferenciadas: el **área de datos**, donde se reserva el espacio en memoria para las variables que lo necesiten, y el **área de código** donde están las instrucciones ejecutables.

El área de datos se identifica por comenzar mediante el directivo `.data`, a partir de ese punto se puede reservar espacio para las variables del programa mediante los directivos siguientes:

- `.ascii` “cadena”: Reserva espacio en memoria para la cadena de caracteres especificada, pero no escribe un terminador de carácter nulo al final. Si se quieren incluir en la cadena caracteres especiales seguiremos las mismas convenciones que en el lenguaje C (\n es el fin de línea, \t es el tabulador, \” son las comillas, etc.)
- `.asciiz` “cadena”: Hace lo mismo, con carácter terminador nulo al final.
- `.byte:` Reserva espacio y lo inicializa con los valores que vienen a continuación cada uno como un entero en un byte.
- `.half:` Reserva espacio y lo inicializa con los valores que vienen a continuación cada uno como un entero en 16 bits.

- `.word` Reserva espacio y lo inicializa con los valores que vienen a continuación cada uno como un entero en 32 bits.
- `.double:` Reserva espacio y lo inicializa con los valores que siguen cada uno como números en representación de punto flotante de precisión doble.
- `.float:` Lo mismo que el anterior, en precisión simple.
- `.space`: Reserva el número de bytes que siguen

El área de código se caracteriza por comenzar con el directivo `.text,` después ya se puede poner la primera instrucción del programa que debe de tener la etiqueta `__start`, que indica que es la dirección a la que se transfiere el control. Esta etiqueta hay que declararla previamente mediante el directivo `.globl` que indica que es una dirección que debe ser reconocida desde fuera del programa, es decir, el comienzo del área de código debe contener las siguientes instrucciones:

```
        .globl __start
        .text
__start:  (primera instrucción)
```

Para finalizar la ejecución de un programa hay que poner las instrucciones

```
        li   $v0 10
        syscall
```

que devuelven el control al sistema operativo.

Instrucciones MIPS32 utilizadas

(En negro, las que se comentan en las transparencias; en azul, otras que son útiles pero que en principio no se comentan en las transparencias)

1) Instrucciones de movimiento de datos

`lw rt,addr`	carga 32 bits en rt de la dirección addr; addr puede ser offset(rs)
`sw rt,addr`	almacena 32 bits de rt en la dirección addr; addr puede ser offset(rs)
`lb rt, addr`	carga 8 bits en rt de dirección addr; addr puede ser offset(rs). Extiende con signo.
`lh rt, addr`	carga 16 bits en rt de dirección addr; addr puede ser offset(rs). Extiende con signo.
`lbu rt,addr`	carga 8 bits en rt de dirección addr; addr puede ser offset(rs). Extiende con ceros.
`sb rt,addr`	almacena los 8 bits menos significativos de rt en addr; addr puede ser offset(rs).
`lhu rt,addr`	carga 16 bits en rt de dirección addr; addr puede ser offset(rs). Extiende con ceros.
`sh rt, addr`	almacena los 16 bits menos significativos de rt a partir de addr; addr puede ser offset(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 (pseudoinstrucción)
`la rd, addr`	carga la dirección addr en el registro rd (pseudoinstrucción)
`li rd,inm`	carga el inmediato inm (de cualquier tamaño) en rd (pseudoinstruc.)

2) Instrucciones de aritmética entera

`add rd,rs,rt`	suma entera rd = rs+rt
`sub rd,rs,rt`	resta entera rd = rs-rt
`addi rs,rt,inm`	suma entera de inmediato rs=rt+inm, inm de 16 bits.
`mul rd,rs,rt`	multiplicación entera rd =rs*rt
`mult rs,rt`	multiplicación entera HI:LO = rs*rt
`multu rs,rt`	multiplicación entera sin signo HI:LO = rs*rt
`div rs,rt`	división entera con signo rs/rt. HI=resto; LO=cociente
`divu rs,rt`	división entera sin signo rs/rt. HI=resto; LO=cociente

3) Instrucciones lógicas

`and rd,rs,rt`	rd = rs AND rt, bit a bit
`andi rt,rs,inm`	rt = rs AND inm (inmediato de 16 bits)
`or rd, rs, rt`	rd = rs OR rt, bit a bit
`ori rt, rs, inm`	rt = rs OR inm (inmediato de 16 bits)
`nor rd, rs, rt`	rd = rs NOR rt

4) Instrucciones de desplazamiento

`sll rd, rt, shamt`	rd = rt << shamt
`srl rd, rt, shamt`	rd = rt >> shamt

5) Instrucciones "set condicional"

`slt rd, rs, rt`	pone rd=1 si rs < rt en comparación con signo, y 0 en caso constrario
`slti rt, rs, inm`	pone rt=1 si rs < inm en comparación con sino, y 0 en caso contrario
`sltu rd, rs, rt`	pone rd=1 si rs<rt en comparación sin signo, y 0 en caso contrario
`sltui rt, rs, inm`	pone rt=1 si rs<inm en comparación sin signo, y 0 en caso contrario

6) Instrucciones de salto

`beq rs, rt, label`	salta a label si rs=rt
`bne rs, rt, label`	salta a label si rs es distinto de rt
`blt rs1, rs2, label`	salta a label si rs1 es menor que rs2 (pseudoinstrucción)
`j label`	salta a label incondicionalmente
`jal label`	salta incondicionamente a label y almacena la dirección de la siguiente instrucción en $ra
`jr rs`	salta incondicionamente a la dirección contenida en el registro rs
`b label`	hace un salto incondicional hacia label (pseudoinstrucción)

7) Instrucciones de punto flotante

a) De movimiento de datos

`mfc1.d rd, fs1`	copia de los registros FP fs1 y fs1+1 hacia los registros rd y rd+1. La parte alta la copia en rd+1 y la baja en rd. Sólo existe instrucción con .d. fs puede ser par o impar. (pseudoinstrucción)
`mtc1 rd, fs`	copia el contenido del registro rd hacia un registro FP fs (copia 32 bits).
`mfc1 rd, fs`	copia el contenido de un registro fs hacia el registro rd (copia 32 bits).
`mov.s fd, fs`	copia de un registro FP fs a otro fd
`mov.d fd, fs`	copia de un par de registros FP fs, fs+1 a otro fd, fd+1.

b) De acceso a memoria

`lwc1 fd, addr`	carga desde addr 32 bits que carga en registro FP fd
`ldc1 fd, addr`	carga desde addr 64 bits que carga en registros FP fd, fd+1
`swc1 fs, addr`	almacena en addr 32 bits que están en registro FP fs
`sdc1 fs, addr`	almacena en addr 64 bits que están en fs y fs+1

c) De 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`	convierte de FP 64 de fs, fs+1 a entero 32 en fd
`cvt.w.s`	convierte de FP 32 de fs a entero 32 en fd

d) Aritméticas

`add.s fd, fs, ft`	suma fd = fs + ft en FP32
`sub.s fd, fs, ft`	resta fd = fs – ft en FP32
`mul.s fd, fs, ft`	multiplica fd = fs*ft en FP32
`div.s fd, fs, ft`	divide fd = fs/ft en FP32
`add.d fd, fs, ft`	suma fd = fs + ft en FP64
sub.d fd, fs, ft	resta fd = fs – ft en FP64
`mul.d fd, fs, ft`	multiplica fd = fs*ft en FP64
`div.d fd, fs, ft`	divide fd = fs/ft en FP64
`neg.d fd, fs`	cambia de signo FP64: fd = - fs
`neg.s fd, fs`	cambia de signo FP32: fd = - fs
`abs.d fd, fs`	calcula valor absoluto de FP64: fd = abs (fs)
`abs.s fd, fs`	calcula valor absoluto de FP32: fd = abs (fs)
`trunc.w.d fd, fs`	trunca a entero el valor almacenado en un par de registros fs, fs+1 y el resultado lo deja en fd.
`trunc.w.s fd, fs`	trunca a entero el valor almacenado en el registro fs y deja el resultado en fd
`sqrt.d fd, fs`	raíz cuadrada, FP 64: fd = sqrt (fs)
`sqrt.s fd, fs`	raíz cuadrada, FP 32: fd = sqrt (fs)

e) Salto

`bc1t cc label`	salta a label si el valor de la bandera de condición de punto flotante cc es 1 (esto es, “verdadero” o “true”). Si no se coloca un valor cc se supone que es la bandera de condición 0.
`bc1f cc label`	salta a label si el valor de la bandera de condición de punto flotante cc es 0 (esto es, “falso” o “false”). Si no se coloca un valor cc, se supone que es la bandera de condición 0.

f) Instrucciones “set condicional”

`c.**.d cc fs, ft`	compara los registros FP de 64 bits (pares de 32) y pone a 1 la bandera de condición cc si se cumple la condición, y si no se cumple, la pone en cero. Si no se especifica un valor en cc se supone que es la bandera de condición 0. ** puede ser: “le” (menor o igual), “eq” (igual), “lt” (menor que).
`c.**.s cc fs, ft`	análogo a la anterior, para registros FP32.

Registros utilizados

`$s0-$s7:`	para almacenar valores de variables de código de algo nivel
`$t0-$t9:`	como almacenamiento temporal para cálculos intermedios
`$a0-$a3:`	para pasa argumentos a un procedimiento
`$v0-$v1:`	para devolver resultados de un procedimiento
`$0:`	`($zero)` todo ceros siempre. Sólo se lee sobre él.
`$sp:`	dirección más baja ocupada por la pila en cada momento (puntero de cabeza de pila)
`$ra:`	registro donde se almacena la dirección retorno después de una llamada de procedimiento
`$fp:`	(registro de “frame pointer”) registro en el que se almacena la dirección en la que comienza el “marco de procedimiento” o “almacén de activación”, que es la zona de la pila donde se almacenan inicialmente las variables locales del procedimiento 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 $sp antes de menter en la pila todas esas informaciones.
`$gp:`	(registro de “global pointer”) Contiene una dirección centrada en la zona donde se almacenan las variables estáticas para poderlas referenciar fácilmente.

Llamadas al sistema que proporciona SPIM

SPIM proporciona un pequeño conjunto de funciones de sistema operativo a través de la instrucción “syscall”. Para solicitar un servicio, un programa carga el código de la llamada de sistema en el registro $v0 y los argumentos en los registros $a0-$a3 (o en $f12 para los valores de punto flotante). Las llamadas de sistema que devuelven valores dejan sus resultados en el registro $v0 (o en $f0 para los resultados de punto flotante). A continuación se resumen las principales de estas llamadas de 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 en $v0
read-float	6		FP32 en $f0
read-double	7		FP64 en $f0
read-string	8	$a0=buffer, $a1=longitud	
exit	10		
print-char	11	$a0=carácter	
read-char	12		Carácter en $v0