Práctica 4
Aritmética de enteros (3) y funciones

Objetivos

- Profundizar en las instrucciones aritméticas del MIPS
- Entender el manejo de las instrucciones de desplazamiento.
- Entender las funciones de programa y saber escribir y llamar a funciones en MIPS.
- Saber hacer una multiplicación utilizando sumas y desplazamientos.
- Entender la multiplicación y la división entera.

Materiales

Simulador MARS y códigos fuente de partida

Desarrollo de la práctica

1. Instrucciones de desplazamiento

Las instrucciones de desplazamiento permiten mover los bits dentro de un registro. Estas instrucciones son importantes especialmente si se trabaja a bajo nivel, por ejemplo, al programar un driver. Una buena razón para utilizar instrucciones de desplazamiento es la mayor comprensión en la multiplicación y la división por una constante.
Podéis hacer desplazamiento de los bits a la derecha o a la izquierda, además las instrucciones del MIPS permiten especificar el número de bits a desplazar. Los desplazamientos lógicos rellenan los espacios con ceros. Los desplazamientos aritméticos rellenan los espacios dejados en los desplazamiento a derecha con el valor del bit de mayor orden (el bit situado a la izquierda del registro) que es el bit del signo en un entero, es decir, replican el bit de signo tantas veces como bits a desplazar.
En un desplazamiento circular el bit que sale por un extremo del registro es el que se introduce por el extremo contrario.
El lenguaje ensamblador MIPS proporciona además dos pseudoinstruccions para hacer las operaciones de rotación a la izquierda y a la derecha, estas son *ror* y *rol* (rotate left/right). Estos operadores se implementan con combinaciones de instrucciones de desplazamiento y la operación lógica *or*. El número de bits a desplazar en cualquiera de las instrucciones de desplazamiento está limitado al rango 0..31.

En la tabla siguiente se muestran las instrucciones de desplazamiento proporcionadas por el MIPS.

Instrucción	Ejemplo	Significado	Comentarios
sll *(shift left logical)*	sll Rd, Rt, k	Rd ← Rt << k	Desplazamiento lógico de *k* bits a la izquierda del valor en *Rt*
sllv (*shift left logical variable)*	sllv Rd, Rt, Rs	Rd ← Rt << Rs	Desplazamiento lógico de *k* bits a la izquierda en *Rt* la cantidad de bits en *Rs*
srl *(shift right logical)*	srl Rd, Rt, k	Rd ← Rt >> k	Desplazamiento lógico de *k* bits a la derecha del valor en *Rt*
srlv (*shift right logical variable)*	srlv Rd, Rt, Rs	Rd ← Rt >> Rs	Desplazamiento lógico de *k* bits a la derecha del valor en *Rt* la cantidad de bits en *Rs*
sra *(shift right arithmetic)*	sra Rd, Rt, k	Rd ← Rt >> k	Desplazamiento aritmético de *k* bits a la derecha del valor en *Rt*
srav *(shift right arith variable)*	srla Rd, Rt, Rs	Rd ← Rt >> Rs	Desplazamiento aritmético a la derecha del valor en *Rt* la cantidad de bits en *Rs*
rol *(rotate left)*	rol Rd, Rt, k	Rd[k..0] ← Rt[31..31-k+1], Rd[31..k] ← Rt[31-k..0],	Pseudoinstrucción. Desplazamiento circular a la izquierda del valor en *Rt* la cantidad de *k* bits.
ror *(rotate fight)*	ror Rd, Rt, k	Rd[31-k..k] ← Rt[31..k], Rd[31..31-k+1] ← Rt[k-1..0]	Pseudoinstrucción. Desplazamiento circular a la derecha del valor en *Rt* la cantidad de *k* bits.

- ¿Con qué instrucciones traducirá el ensamblador las pseudoinstrucciones *rol* y *ror*?. Escribe un código sencillo de prueba de ambos operadores y ensámblalos para comprobarlo.
- Las instrucciones de desplazamiento siguen el formado tipo R. ¿Cuál es la codificación en hexadecimal de la instrucción `ssl $t2,$t1,3`? Y de la instrucción `srl $t2,$t1,7`. ¿Cuál es el valor de cada campo? Ayúdate ensamblado un código de prueba.
- Descubre la palabra escondida. Dado el siguiente código, complétalo de tal manera que mediante instrucciones lógicas y de desplazamientos puedas escribir en la consola cada uno de los caracteres que se encuentran almacenados en cada byte del registre *$t1*.

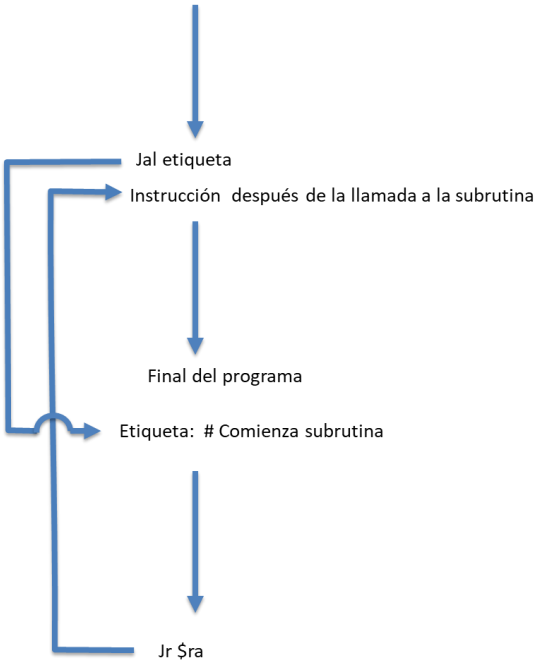
```
#Palabra escondida
li $t1, 1215261793

.......

move $a0, $t2
li $v0, 11
syscall

li $v0, 10
syscall
```

2. Funciones del programa

Al igual que en lenguajes en alto nivel, el ensamblador permite programar subrutinas o procedimientos. Una subrutina o subprograma es un fragmento de código diseñado para hacer una tarea determinada la cual puede ser invocada desde cualquier lugar del programa principal e incluso des de otra subrutina. Esta subrutina puede llamarse tantas veces como se desee. Las subrutinas permiten escribir programas más compactos y estructurados y por lo tanto más legibles al evitar repeticiones de código innecesarios y obtener código reutilizable. Las subrutinas a veces reciben información a través de un conjunto de parámetros con los valores de los cuales harán las operaciones o cálculos necesarios. Si el resultado de las operaciones es un valor se devolverá al finalizar la subrutina. En este caso la subrutina recibe el nombre de ***función***. Si la subrutina solamente ejecuta una secuencia de instrucciones sin devolver ningún valor se denomina ***procedimiento***. Los registros `$a0`, `$a1`, `$a2`, `$a3` son los utilizados para pasar los parámetros a la subrutina y los registres `$v0`, `$v1` son los utilizados para devolver valores de una función. Si se requieren más se utilizará la pila, pero de momento no se estudiará, ya la comentaremos más adelante.

Las subrutinas tienen que tener un punto de entrada y un punto de regreso al programa que los invoca y se han de ejecutar de manera independiente al programa principal. El lenguaje ensamblador del MIPS proporciona dos instrucciones para diseñar subrutinas.
La primera instrucción es la jal (jump and link) esta instrucción salta a la dirección donde se encuentra la subrutina (modifica el registro PC) y guarda en el registro $ra la dirección de regreso al programa que la llama. Es necesario guardar la dirección de regreso porque se puede llamar a la misma subrutina desde varios lugares del programa. Esta instrucción se escribe: `jal DireccionDeLaSubrutina`. La instrucción jal sigue el formado tipo J estudiado en la práctica anterior al introducir la instrucción j.
Cuando se escribe una subrutina en MIPS se etiqueta la primera instrucción con el nombre que se le quiere dar a la subrutina. Posteriormente se llamará a la subrutina utilizando el nombre de la etiqueta la cual especificará la dirección donde se encuentra.
La segunda instrucción es jr (jump register) y se escribe `jr $ra`, es la instrucción de regreso de la subrutina y se colocará al final de esta. Esta instrucción salta a la dirección almacenada en $ra (modifica el registro PC).
Por ahora estudiaremos exclusivamente funciones hoja que son aquellas que no llaman a otra función. Los procedimientos anidados que son aquellos que llaman a otras subrutinas las estudiaremos más adelante.

En la siguiente figura podéis observar la semántica de una llamada a una función:



Código de partida. En el siguiente código se ha escrito una función que escribe en la consola el valor entero pasado por argumento, a continuación hace un salto de línea.

```
            # Prueba de llamada a una función
            .text

            li $a0, '>'          #Comienza programa principal
            li $v0,11
            syscall
            li $v0,5
            syscall              #Leer un enter

            addi $t1,$v0,10
            move $a0, $t1        #argumento a pasar en $a0
            jal imprim           #llamamos a la función
            add $t1, $t1,$t1

            move $a0, $t1        #argumento a pasar en $a0
            jal imprim           #llamamos a la función

            li $v0,10            #Acaba el programa
            syscall

  #---------Funcions---------

imprim:     addi $v0,$0,1        #comienza la función
            syscall              #Escribe un valor
            li $a0, '\n'         #Salto de línia
            li $v0,11
            syscall
            jr $ra               #Vuelta al programa principal
```

- Ensamblad el programa y observad en qué direcciones se colocan las instrucciones. Ejecutadlo a pasos. Fijaos en la ventana de registros como van cambiando los contenidos de los registros PC y $ra.
- Escribe una función con instrucciones suma que devuelve el cuádruplo del número entero que se le pasa. Escribid el programa principal que lea el número del teclado y escriba el cuádruplo en la consola aprovechando la función *imprim*.

3. Convenio de los registros del MIPS

Una situación que podría ocurrir y hay que evitar es que la función a la que se llama modifique los datos del programa principal (por ejemplo escribiendo en registros). El programa puede necesitar los datos una vez se ha vuelto de la función. Por esta razón los diseñadores del MIPS establecieron unos convenios que los programadores tendrían que seguir para evitar borrar datos erróneamente. Las reglas afectan a los registres $s0,..,$s7 y $t0,...,$t7:

- El programa que llama tiene que suponer que la función llamada puede modificar los registros temporales $t0,...,$t7. Además, el programa que llama tiene que saber que las funciones utilizan los registros $v0 y $v1 para devolver valores. Por lo tanto, es responsabilidad del programa que llama almacenar en otro lugar los valores que quiera preservar.

- El programa que llama ha de asumir que los valores que estén en lo registres $s0,..,$s7 no se modificarán por la función llamada. Esto significa que si la función necesita utilizar alguno de ellos tendrá que preservar su valor antes de usarlos y restaurar el valor original antes de devolver en el programa que lo ha llamado. El mismo ocurre con los registros de paso de parámetros $a0,...,$a3, el programa que llama supone que la función no modificará su contenido.

Cómo ha resumen, en la mesa siguiente se muestra los convenios y como se deben utilizar los registros implicados en las llamadas a funciones:

Registro	Uso
$s0,..,$s7	Utilizados por el programa principal. Preservados en las llamadas.
$t0,...,$t7	Utilizados en las funciones. No preservados en las llamadas.
$a0,...,$a3	Paso de parámetros a las funciones.
$v0, $v1	Valor de retorno de las funciones.

- Observad el último código escrito, ¿se ajusta al convenio de MIPS de utilización de registros? De ahora en adelante haced servir el convenio.

4. Multiplicación por desplazamientos y sumas

Una buena razón para utilizar instrucciones de desplazamiento es la mayor comprensión de la multiplicación y la división por una constante.
Las instrucciones de desplazamiento a la izquierda nos sirven para hacer multiplicaciones por potencias naturales de 2 y las instrucciones de desplazamiento a la

derecha para hacer divisiones por potencias naturales de 2. Estas instrucciones de desplazamiento utilizadas junto con instrucciones de sumas nos permiten hacer una multiplicación por cualquier constante.
Vamos por pasos, observad el siguiente código de partida en el que se da una función que hace la multiplicación de un entero por 4 utilizando instrucciones de desplazamiento. El programa pide el entero por teclado y escribe la solución en consola.

```
            # Multiplicación por 4
            .text
            li $a0, '>'
            li $v0,11
            syscall
            li $v0,5
            syscall             #Leer un entero

            move $a0, $v0       #parámetro a pasar en $a0
            jal mult4           #llamamos a la función mult4
            move $a0, $v0
            jal imprim          #Llamamos a la función imprim
            li $v0,10           #Acaba el programa
            syscall

#---------------Funcions-----------------#

imprim:     addi $v0,$0,1       #función imprim
            syscall             #Escribe el valor en $a0
            li $a0, '\n'        #Salto de línia
            li $v0,11
            syscall
            jr $ra              #Vuelve al programa principal

mult4:      sll $v0, $a0, 2     #Función multiplica entero per 4
            jr $ra
```

- Ensambla y prueba el programa.
- Modifica el código en el que ahora haya una función multi5 que multiplique por 5 y muestre el resultado por consola. Comprobar que el resultado es correcto.
- Modifícalo ahora para tener una nueva función mult10 que multiplique por 10. Comprobar que el resultado es correcto.

Acabáis de ver que la multiplicación por un número que es potencia de 2 se puede hacerse sencillamente utilizando el operador de desplazamiento lógico a la izquierda. La multiplicación por un número que no es potencia entera de 2 también resulta sencilla si combinamos el desplazamiento lógico a la izquierda con instrucciones de suma. De esta manera podemos multiplicar cualquier número por cualquier constante. Lo que hay que tener con cuenta es el desglose de la constante a multiplicar en potencias de 2. Por ejemplo, el valor 11 se escribe como $11= 2^3+2^1+2^0$. Una multiplicación de un valor N por la constante 11 es $Nx11= Nx2^3+Nx2^1+N$, es decir, se traducirá en 2 operaciones de desplazamientos y dos sumas, donde los exponentes indican los bits a desplazar.

- Escribe una función que multiplique por 60. Escribe el programa principal que lea una cantidad de minutos y devuelve por consola la cantidad en segundos.
- Modifica el código de tal manera que ahora lo que lea sea una cantidad de hora y muestre por consola la cantidad de segundos.

5. Multiplicación por desplazamientos, sumas y restas

La idea que hemos utilizado hasta ahora de realizar la multiplicación por una constante mediante instrucciones de sumas y desplazamientos la podemos extender a la multiplicación por una constante según el algoritmo de Booth que nos permite multiplicar tanto por valores positivos como negativos. Ahora una multiplicación por una constante constará de instrucciones de desplazamientos, de suma y de restas. Por ejemplo, el valor 11 en binario utilizando 8 bits se escribe como $11=00001011_2$. Al recodificarlo según el algoritmo de Booth con los valores 1, 0 y -1 quedan como 11=000+1-1+10-1. A la hora de aplicar el algoritmo el '+' significa una suma y el '-' una resta. Cuando lo expresamos en potencias de 2 recodificado por Booth se transforma en $11= 2^4-2^3+2^2-2^0$. En este caso la multiplicación por la constante 11 según Booth se transforma en 3 instrucciones de desplazamiento, 2 de suma y 2 de resta. Vemos que en este caso concreto han aumentado el número de instrucciones; en otros casos la multiplicación se simplificará, todo dependerá de la estructura de bits del valor a recodificar.

- Escribe la función que multiplique por la constante 11 según el algoritmo de Booth y comprueba que el resultado es correcto.

6. Multiplicación y división generales

En el caso de que se requiera de operaciones de multiplicación o división de dos valores enteros cualquier, el lenguaje ensamblador del MIPS proporciona varias instrucciones de multiplicación y división. El resultado de estas instrucciones se coloca en una pareja especial de registros de 32 bits no incluida en los 32 registros de propósito general del MIPS. Estos registros especiales se denominan *hi* y *lo*.
La interpretación del contenido de estos dos registros depende de la operación realizada. En el caso de la multiplicación la parte alta del resultado se deja en *hi* y la parta baja en *lo*. Conviene notar que el tamaño de la palabra es de 32 bits, por lo tanto, si el registro contiene algún valor distinto de cero significa que puede haberse producido desbordamiento en la multiplicación. ¿Cuándo se producirá realmente desbordamiento? En el supuesto de que el resultado tenga que ser positivo, si el valor de *hi* es distinto de cero claramente se habrá producido desbordamiento. Si el resultado es negativo no habrá desbordamiento cuando todos los bits del registro están a unos y además el bit de mayor peso de *lo* también lo es.
En el caso de la división en *hi* se coloca el resto y en *lo* el cociente. La división por cero no se puede hacer en MIPS, es una operación indefinida y no se detecta por el ensamblador, es por lo tanto responsabilidad del programador evitar que se produzca.
Para poder extraer los contenidos de los registros *hi* y *lo* para operar con ellos el MIPS proporciona unas instrucciones especiales de acceso a estos registros. En la tabla siguiente aparecen todas las instrucciones relacionadas con la multiplicación y división:

Instrucción	Ejemple	Significado	Comentarios
mult	mult Rs, Rt	[hi,lo] ←Rs * Rt	Multiplicación de *Rs* y *Rt*. Deja el resultado en [*hi*,*lo*]
mflo	mflo Rd	Rd ← lo	Mueve el valor de *lo* al registro *Rd*
mfhi	mfhi Rd	Rd ← hi	Mueve el valor de *hi* al registro *Rd*
multu	multu Rs, Rt	[hi,lo] ←Rs * Rt	Multiplicación sin signo de *Rs* y *Rt*. Deja el resultado en [*hi*-*lo*]

div	div Rs, Rt	[hi,lo] ← Rs / Rt	División de Rs por Rt. Deja el resultado en [*hi*,*lo*]. El cociente en *lo* y el residuo en *hi*.
divu	divu Rs, Rt	[hi,lo] ← Rs / Rt	División sin signo de Rs por Rt. Deja el resultado en [*hi*,*lo*]. El cociente en *lo* y el residuo en *hi*.

Una cuestión importante a tener en cuenta con las operaciones de multiplicación o división es su coste temporal. Este coste puede llegar a ser muy apreciable siendo el coste de la división superior al de la multiplicación. Claramente dependerá de la implementación del procesador concreto y en muchos casos del tamaño de los operandos. Este coste será muy superior al conseguido en las operaciones de multiplicación o división con las instrucciones de suma y desplazamiento. Por lo cual tendríamos que tenerlo presente si tenemos que multiplicar o dividir por constantes sencillas.

Ejercicio a entregar

- Escribe el código que lee el valor x y escribe por pantalla la solución de la ecuación: $5x^2 + 2x + 3$.

Resumen

- Se utiliza una etiqueta en la primera instrucción de una función para darle nombre.
- Hay una serie de convenios en el uso de registros que hay que seguir en la programación de funciones del programa.
- La multiplicación y la división se puede hacer utilizando instrucciones de sumas, restas y desplazamientos o las propias que aporta el repertorio de instrucciones del MIPS.