**Práctica 2**

**Aritmética de enteros (1), operaciones lógicas, pseudoinstrucciones y Entrada/Salida**

**Actividad 1.**

➢ Probad el siguiente código: **Imprimir en la consola un valor entero.** Se utilizará la **función 1** (*print integer*). Si colocamos un 1 en el registro *$v0* se escribe el valor del entero contenido en *$a0* en la consola Run I/O

####################################

# #

# Código de la actividad 1 #

# Imprimir en consola #

# #

####################################

.text 0x00400000

addi $a0,$0,25 #Valor a escribir en $a0 addi $v0,$0,1 #Función 1, print integer

syscall #Escribe en consola $a0

Este ejemplo imprimirá en la consola 25 que es el valor contenido en *$a0.*

**Actividad 2.**

* Probad el siguiente código: **Leer de teclado e imprimir en consola.** Para leer se utiliza la función 5 (*read integer*). Si ponemos 5 en el registro *$v0* se lee un entero del teclado (con el cursor parpadeando el simulador queda a la espera que introduzcamos un entero y pulsemos *intro*). El valor leído se almacenará en *$v0*.

|  |
| --- |
| ######################################  # #  # Código de la actividad 2 #  # Leer el valor introducido por #  #teclado e imprimirlo en la consola #  # #  #####################################  .text  addi $v0,$0,5 #Función 5, read integer syscall #Valor leído en $v0    addi $a0,$v0,0 #Movemos el valor leído a $a0 addi $v0,$0,1 #Función 1, print integer  syscall #Escribimos en consola $a0 |

Este programa utiliza la función 5 y lee del teclado un valor y lo guarda en *$v0*. A continuación movemos este valor a *$a0* para escribirlo en pantalla utilizando la función

1.

**Actividad 3.**

* Probada el siguiente código: **Finalizar el programa.** Incorporamos la función 10 (*exit*), lo que hace esta función es salir del proceso o acabar el programa. De aquí en adelante lo utilizaremos para finalizar los programas.

|  |
| --- |
| ######################################  # #  # Código de la actividad 3 #  # Leer el valor introducido por #  # teclado, imprimirlo en la consola #  # y acabar el programa #  # #  #####################################  .text  addi $v0,$0,5 #Función 5, read integer syscall #Valor leído en $v0    addi $a0,$v0,0 #Movemos el valor leído a $a0 addi $v0,$0,1 #Función 1, print integer syscall #Escribimos en consola $a0  addi $v0,$0,10 #Función 10, exit  syscall #Acaba el programa |

**Cuestión 1.**

* Escribe el código para que se lea un valor *x* de teclado y escriba *x+1* en la consola.

**Cuestión 2.**

* Haz el código para que se lea un valor *x* de teclado y escriba *x-1* en la consola.

## 2. El conjunto de instrucciones: Aritmética de enteros

Las operaciones del MIPS con la Unidad Aritmética y Lógica (ALU) utilizan 3 operandos, dos de ellos son fuentes de entrada a la ALU y un tercero de salida que es el resultado de la operación realizada. El MIPS, por su diseño, sólo permite que los operandos de la ALU estén en el procesador. El destino del resultado de la operación será siempre un registro, en cuanto a los operandos fuente, uno de ellos ha de estar siempre en un registro, pero el otro puede provenir de un registro o ser un valor constante proporcionado por la propia instrucción. Esto da lugar a dos tipos de instrucciones aritméticas y lógicas dependiendo de dónde se encuentran los operandos fuente. De momento nos centraremos en las instrucciones aritméticas de enteros.

En la práctica 1 habéis estudiado la instrucción suma *addi* en la que uno de los operandos fuentes y el operando destino son registros y el otro operando fuente es una constante: *addi rt, rs, K* .

Veamos ahora el operador suma del MIPS que utiliza 3 registros como parámetros, se denomina *add* y tiene la siguiente sintaxis:

*add $rd, $rs, $rt*

En la figura se muestra un esquema de la operación:



*Figura 1. Representación gráfica de la instrucción add.*

Lo que hace esta instrucción es sumar el contenido de los registros *rs* y *rt* y almacena el resultado en *rd.*

**Cuestión 3.**

* ¿Cómo se escribe la instrucción que hace $t2=$t1+$t0?

Otra operación con el operador suma con tres registros es *addu*.  Esta instrucción tiene un significado similar a la instrucción *addiu* ya estudiada en la pràctica 1.

Con el operador resta con tres registros existen las instrucciones *sub* y *subu*. En la tabla 1 podéis ver las instrucciones comentadas:

|  |  |  |  |
| --- | --- | --- | --- |
| **Instrucción** | **Ejemplo** | **Significado** | **Comentarios** |
| **add** | add Rd,Rs,Rt | Rd  Rs + Rt | Suma Rs y Rt y coloca el resultado en Rd |
| **addi** | addi Rd,Rs,K | Rd  Rs + K | Suma Rs y una constante y coloca el resultado en Rd |
| **addu** | addu Rd,Rs,Rt | Rd  Rs + Rt | Suma Rs y Rt y coloca el resultado en Rd. Asume valores sin signo. |
| **addiu** | addiu Rd,Rs,K | Rd  Rs + K | Suma Rs y una constante y coloca el resultado en Rd. Asume valores sin signo. |
| **sub** | sub Rd,Rs,Rt | Rd  Rs – Rt | Resta Rs menos Rt y coloca el resultado en Rd |
| **subu** | subu Rd,Rs,Rt | Rd  Rs – Rt | Resta Rs menos Rt y coloca el resultado en Rd. Asume valores sin signo. |

*Tabla 1. Instrucciones de suma y resta del MIPS.*

Cabe destacar que no aparece en la tabla la instrucción *subi* ya que no forma parte del repertorio de instrucciones del MIPS.

**Actividad 4.**

Escribe y observa el siguiente código de partida sobre aritmética de enteros

|  |
| --- |
| ######################################  # #  # Código de la actividad 4 #  # Aritmética de enteros #  # #  #####################################    .text 0x00400000 addiu $t0, $zero, 25 addiu $t1, $zero, 5 sub $t2,$t0,$t1  addi $v0, $zero, 10 #Salir del programa  syscall |

Haz un análisis previo del código

* ¿Qué hace cada línea del código de partida?
* ¿En qué dirección de memoria se almacena la instrucción *sub*?

**Cuestión 4.**

* Ensambla y ejecuta el código. ¿Cuál es el valor final del registro *$t2*?

**3. Introducción al formato de instrucciones del MIPS: código máquina.**

Ya hemos comentado que el ensamblador traduce los programas escritos en ensamblador en un lenguaje inteligible por la máquina, que no es más que cadenas de ceros y unos. Esto lo hace codificando cada una de las instrucciones atendiendo al formato que siguen.

Todas las instrucciones MIPS requieren 32 bits para codificarse (1 palabra= 4 bytes= 32 bits). Los seis bits superiores de la palabra, es decir, los bits de más a la izquierda (bits 26-31) contienen el código de operación que especifica qué operación realizará la instrucción. Por tanto, hay 26=64 códigos de operación distintos, aunque, como iremos viendo a lo largo de las prácticas, hay más de 64 operaciones distintas. El resto de bits de la palabra se organizan en campos donde especificar los operandos de la instrucción. Hay tres formados de instrucción llamados tipo R, I, y J, aunque por ahora introduciremos sólo los formados R e I.

Las instrucciones *addi* y *addiu* siguen el mismo formato de instrucción, el denominado formato **tipo I**. Estas instrucciones utilizan como operandos fuente un registro y un dato inmediato y como operando destino un registro.

La forma general de codificación del formado tipo I es la que muestra la figura 2:

31 26 25 21 20 16 15 0

|  |  |  |  |
| --- | --- | --- | --- |
| *Operación* | *Rs* | *Rt* | *Inmediato* |

### Tipo I

6 bits 5 bits 5 bits 16 bits

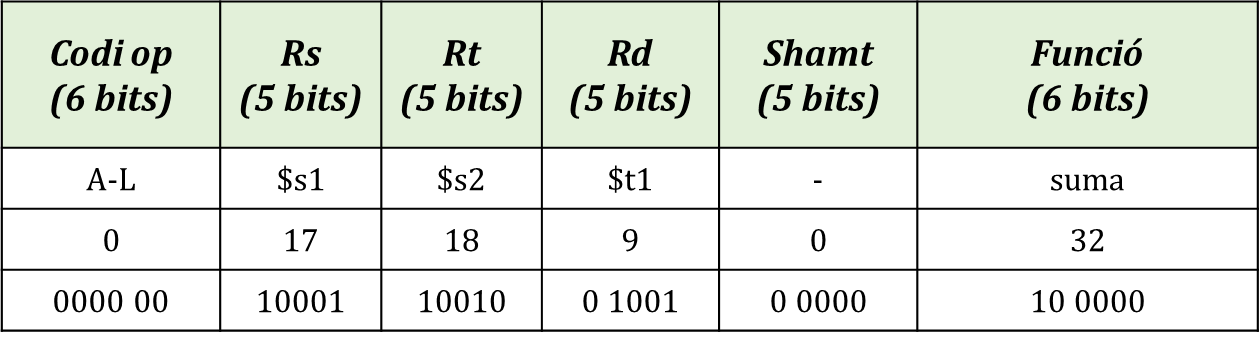
*Figura 2. Codificación del formato tipo I.*

Como hemos dicho, el primer campo, es decir, los 6 bits superiores (26-31), corresponden al código de operación y especifican qué operación realizará la instrucción. Los siguientes dos campos indican el número de los registros mediante 5 bits ya que disponemos de 32 registros. *Rs* es el registro fuente y *Rt* es el registro destino. Los 16 bits restantes los utilizamos para especificar el dato inmediato.

**Actividad 5.**

* Observad el código de partida *Aritmética de enteros* de la actividad 4. ¿Cómo se codifica la primera instrucción? Hacedlo a mano (el código de operación de la instrucción *addiu*es 0x09 o 001001 en binario)
* Confirmad con el programa ensamblado que el código máquina es el mismo.

Las instrucciones que utilizan 3 registros como parámetros siguen otro formato de instrucción. La instrucción *add $t1, $s1, $s2*se codifica de la siguiente manera:



*Figura 3. Codificación de la instrucción add $t1, $s1, $s2.*

En hexadecimal quedaría como: 0x02324820.

El primer campo corresponde al código de operación e indica que se trata de una operación aritmética o lógica. Todas las instrucciones aritméticas o lógicas tienen siempre el valor 000000 para el código de operación. Los siguientes tres campos corresponden a los tres registres operandos, *Rs* y *Rt* como registros fuente y *Rd* como registro destino. El quinto campo de 5 bits (shamt) no tiene sentido en las instrucciones de suma o resta y su valor será siempre 0. El campo función indica la operación aritmética o lógica a realizar (suma, resta...), en nuestro caso se trata de una suma y por lo tanto el valor es 32 (0x20 o 10 0000 en binario).

**Cuestión 5.**

* ¿Cómo se codifica la última instrucción de resta del código de partida *Aritmética de enteros* de la actividad 4? Hacedlo a mano (el campo función de la resta es 0x22 o 10 0010 en binario)
* Confirmad en el MARS con el programa ensamblado que el código máquina es el mismo.

La forma general del formato tipo R es el siguiente:

31 26 25 21 20 16 15 11 11 6 5 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *Operación* | *Rs* | *Rt* | *Rd* | *shamt* | *Función* |

### Tipo R

6 bits 5 bits 5 bits 5 bits 5 bits 6 bits

*Figura 3. Codificación del formato tipo R.*

Cómo hemos dicho, los 6 bits superiores (26-31) de la palabra corresponden al código de operación, este campo tiene siempre el valor 0x0. Los siguientes 3 campos especifican los registres fuente *Rs*, *Rt* y el registro destino *Rd* mediante 5 bits. El quinto campo, *shamt*, es la cantidad de desplazamiento (utilizado en las instrucciones de desplazamiento). El último campo de 6 bits es la *función* y nos sirve para distinguir las distintas instrucciones que siguen el formato tipo R.

**Cuestión 6.**

* Escribe el código que realice las siguientes acciones haciendo uso de las instrucciones estudiadas:

$t0=5

$t1=$t0+10

$t2=$t0+$t1

$t3=$t1-30

* Ensámblalo y ejecútalo y comprueba que el contenido de los registros es correcto.
* A la vista del código escrito, ¿es necesario que forme parte del repertorio de instrucciones la instrucción *subi*?

**Cuestión 7.**

* Escribe el código que lea dos números x e y, y obtenga por pantalla el valor de la suma x+y.

## 4. Repertorio de instrucciones: instrucciones lógicas

El MIPS incorpora un conjunto de operadores lógicos que permiten operar con los bits individuales de los operandos fuentes. Las instrucciones de operaciones lógicas pueden operar bit a bit con los dos operandos como registros fuentes y en ese caso siguen el formato tipo R, o con un operando fuente registro y un valor inmediato, en ese caso siguen el formado tipo I. Exactamente igual que con las instrucciones aritméticas estudiadas. Las instrucciones lógicas son: *and, andi, or, ori, xor, xori nor.* En la tabla 2 se muestran las instrucciones lógicas del MIPS:

|  |  |  |  |
| --- | --- | --- | --- |
| **Instrucción** | **Ejemplo** | **Significado** | **Comentarios** |
| **and** | and Rd,Rs,Rt | Rd  Rs & Rt | 3 registros operandos; AND lógica |
| **or** | or Rd,Rs,Rt | Rd  Rs | Rt | 3 registros operandos; OR lógica |
| **xor** | xor Rd,Rs,Rt | Rd  Rs  Rt | 3 registros operandos; XOR lógica |
| **nor** | nor Rd,Rs,Rt | Rd  (Rs | Rt) | 3 registros operandos; NOR lógica |
| **and inmediata** | andi Rd,Rs,10 | Rd  Rs & 10 | AND lógica registros y constantes |
| **or inmediata** | ori Rd,Rs,10 | Rd  Rs | 10 | OR lógica registros y constantes |
| **xor inmediata** | xori Rd, Rs,10 | Rd  Rs 10 | XOR lógica registros y constantes |

*Tabla 2. Instrucciones lógicas del MIPS.*

Hasta ahora habéis utilizado la instrucción *addi*para poner un valor inmediato en un registro: *addi rd, $zero, inmediato.*

**Cuestión 8.**

* ¿Podríamos utilizar la instrucción lógica *ori* para dar un valor inicial a un registro en lugar de la instrucción *addi*?
* ¿Cómo escribirías la instrucción que hace *$t2*=7 utilizando *ori*?

**Cuestión 9.**

* Escribe el código que haga la operación lógica OR de *$t1* y *$t2* y lo guarde en *$t3*, la operación lógica AND de *$t1* y *$t2* y lo guarde en *$t4*, y la operación lógica XOR de *$t1* y *$t2* y lo guarde en *$t5*. Escribe en la ventana de registros, tras ensamblarlo, los siguientes valores para los registros $t1=0x55555555 y $t2= 0xAAAAAAAA. Ejecuta el código y estudia los resultados.

## 5. Ayudas a la programación: más sobre entrada y salida

Hasta ahora habéis visto que podéis leer de teclado y escribir en la consola valores enteros con la instrucción *syscall* utilizando las funciones 1 y 5. También podéis escribir por consola valores hexadecimales mediante la **función 34** y valores en binario mediante la **función 35** (mirad en *Mars Help> MIPS>Syscalls*).

**Actividad 6.**

|  |
| --- |
| ######################################  # #  # Código de la actividad 6 #  # Imprimir en consola #  # valores hexadecimales #  # #  #####################################  # Imprimir en consola valores hexadecimales  .text 0x00400000  ori $a0,$0,0xABC #En $a0 el valor a escribir ori $v0,$0,34 #Función 34, print hexadecimal syscall #Escribe en consola el valor $a0 |

* Ensambla, ejecuta el código y observa el resultado.

**Cuestión 10.**

* Escribe el código que lee un valor entero por teclado y escribe el mismo valor en binario por la consola.

## 6. Pseudoinstrucciones

Si tenéis intención de inicializar un registro $t0=10, hasta ahora habéis escrito *addi*

*$t0,$0,10* o *ori $t0,$0,10*. Ahora estudiaremos una manera alternativa de hacerlo.

El ensamblador dispone de lo que se denominan *pseudoinstrucciones*, que son pseudooperadores que no existen en el conjunto real de instrucciones del MIPS pero nos permiten escribir de una manera distinta o de una manera más fácil ciertas operaciones muy comunes. Lo que hace el ensamblador es traducir las pseudoinstrucciones en una o varias instrucciones reales.

Por ejemplo, la pseudoinstrucción ***li* (*load immediate*)** sirve para cargar un dato en un registro. Si en lugar de *addi $t0,$0,10* escribís *li $t0,10*, el ensamblador traducirá esta pseudoinstrucción por *addi $t0,$0,10*.

➔ De ahora en adelante cada vez que queráis que un programa en ensamblador cargue una constante en un registro podéis utilizar la pseudoinstrucción *li*. Es más general y más legible que *addiu*.

Si queréis copiar el contenido de un registro a otro, hasta ahora habéis escrito, por ejemplo, *addi $t1,$t0,0*. Ahora podéis utilizar la pseudoinstrucción *move $t1,$t0* con el mismo efecto.

Si queréis invertir cada uno de los bits del registro *$t2* y guardarlo en el registro *$t1*, hasta ahora habríais escrito *nor $t1, $t2, $zero*. Ahora podrías escribir en su lugar *Not $t1, $t2*.

Si queréis hacer una resta con el valor inmediato k, hasta ahora habríais escrito, por ejemplo*, addi $t1,$t0,-k*. Ahora podéis reescribirlo utilizando la pseudoinstrucción *subi* como: *subi $t1, $t0, k*.

**Actividad 7.**

➢ Comprueba cómo se traducen las siguientes pseudoinstrucciones al ensamblar el programa. Mira la columna de la izquierda del código nombrada *Basic* en la ventana *Text segment* una vez ensamblado el programa.

|  |
| --- |
| ####################################  # #  # Código de la actividad 7 #  # Traducción de pseudoinstrucción #  # por el ensamblador #  # #  ####################################    li $t1, 4 move $t2, $t1 not $t3, $t2  subi $t4,$t1,1 |

## 7. Constantes grandes

Recordad que el formado de instrucciones del MIPS tiene 32 bits y que las instrucciones que permiten operar con datos inmediatos (las de formato tipo I) sólo tienen 16 bits para alojar el dato inmediato. Esto quiere decir que si necesitamos cargar un registro con un dato que ocupe más de 16 bits no podemos hacerlo con una instrucción con formado tipo I. Este proceso requerirá una serie de pasos.

**Actividad 8.**

* Prueba el siguiente código

|  |
| --- |
| ####################################  # #  # Código de la actividad 8 #  # Cargar datos muy grandes #  # #  ####################################    li $t0,0x10000000 li $t1,0x10000100 |

* ¿Con qué instrucciones tiene que traducir el ensamblador la pseudoinstrucción *li*?
* ¿Qué hace la instrucción *lui*? Buscad en la web o en la ayuda de las instrucciones básicas del MARS.

**Cuestión 11.**

* Escribid el código (3 líneas entre instrucciones y pseudoinstrucciones) que hace las siguientes acciones:

$t0=5

$t1=$t0+10

$t2=$t1-30

* Ensamblad y ejecutad el programa. Comprobad que el resultado final es ($t0=5, $t1=15, $t2=-15) es correcto.

## 8. Entrada/salida de caracteres

Hasta ahora todos los datos que habéis utilizado han sido valores enteros. Los computadores también permiten manipular datos alfanuméricos fácilmente legibles por los usuarios. El sistema que permite la codificación de los caracteres en valores binarios se denomina ASCII (American Standard Code for Information Interchange). En ASCII todos los caracteres están representado por un número comprendido entre 1 y 127 almacenado en 8 bits. Las codificaciones ASCII se muestran en la siguiente tabla:

Imatge que conté text, menú, captura de pantalla, nombre

Descripció generada automàticament

*Figura 1: Codificaciones ASCII.*

Utilizando esta tabla se pueden codificar cadenas de caracters. Por ejemplo, “Hola” en caracteres ASCII se codifica con el valor hexadecimal 0x486F6C61 (“H”= 0x48, “o”=0x6F, “l”=0x6C y “a”=0x61).

También se pueden representar en ASCII valores numéricos. Por ejemplo, el número 12 en binario es 0xC=11002. Si se representa como una cadena de caracteres, utilizando la codificación ASCII sería 0x3132 (“1”=0x31 y “2”=0x32).

Con la instrucción *syscall* podemos tener entrada de caracteres por teclado y salida de caracteres por la consola con las **funciones 11** (*print character*) y **12** (*read character*).

**Actividad 9**

* Probad el siguiente código que escribe en consola un carácter leído del teclado.

|  |
| --- |
| ####################################  # #  # Código de la actividad 9 #  # Programa ECHO #  # #  ####################################    li $v0,12 #Función 12. Read character syscall #Carácter leído en $v0    move $a0,$v0 #Carácter a escribir en $a0  li $v0,11 #Función 11. Print character  syscall  li $v0, 10 #Función 10. Acaba programa  syscall |

Podéis cargar directamente un carácter en un registro poniéndolo entre comillas simples, por ejemplo, *li $a0,'x'*o cargarlo utilizando su valor ASCII: *li $a0,0x78*.

**Cuestión 12.**

* El carácter ‘\n‘ es el carácter de nueva línea y provoca que la salida por consola desde el programa comience en una línea nueva. Modifica el código anterior para que al leer un carácter muestre la salida en una línea diferente.

**Cuestión 13.**

* Con el fin de facilitar la claridad de ejecución, modifica el código anterior para que muestre ‘>’ o ‘?’ antes de leer el carácter como una manera de solicitarlo.

**Cuestión 14.**

* Escribe el código que imprime por consola el valor ASCII en hexadecimal del carácter leído.

## 9. Etiquetas

En el ensamblador del MIPS una cadena de caracteres seguida de “:” es una etiqueta. Una etiqueta es un identificador que representa un valor y puede ser utilizado en cualquier línea de código. Las etiquetas no son equivalentes a variables, las etiquetas simplemente son marcas en el programa. El valor representado por una etiqueta puede ser una dirección de memoria y la podemos utilizar en cualquier lugar de nuestro programa.

En el ejemplo siguiente podéis observar dos etiquetas *eti1* y *eti2*, cada una de ellas toma el valor de la dirección en la que se encuentra la instrucción que señalan. Así *eti1* toma el valor de la dirección 0x00400000 que es dónde se encuentra la instrucción *addi $v0,$0,5* y *eti2* el de 0x00400008, dónde se encuentra *addi $a0,$v0,15*.

####################################

# #

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | # Código de la actividad 10 | | | # |
|  | # Pruebas de etiquetas | | | # |
|  | #  ####################  .text  li $a0, '>' li $v0,11 syscall | | #  ################  #Función 11. Print character | |
| eti1: | addi $v0,$0,5 syscall | | #Función 5. Read integer | |
| eti2: | addi $a0,$v0,15 addi $v0,$0,1 | | #Función 1. Print integer | |
|  | syscall  li $a0, '\n'  li $v0,11 #Función 11. Print character  syscall  la $t1, eti1 #Carga en $t1 la dirección de eti1 La $t2, eti2 #Carga en $t2 la dirección de eti2    move $a0,$t1  li $v0,34 #Función 34. Print en hexadecimal syscall    li $a0, '\n' li $v0,11 #Función 11. Print character syscall  move $a0,$t2  li $v0,34 #Función 34. Print en hexadecimal | | | |
|  | syscall  li $v0,10 syscall | #Función 10: termina el programa | | |

La pseudoinstrucción ***la* *(load address)*** sirve para cargar un dato de 32 bits que especifica una dirección en un registro. En el ejemplo *la $t1, eit1* indica que se cargue en lo registro *$t1* el valor de la dirección de la etiqueta *eti1*.

**Actividad 10.**

* Ejecuta el código y comprueba el valor correcto de las etiquetas.
* ¿Qué hace el código?
* ¿En qué instrucciones básicas se ha traducido la pseudoinstrucción *la*?

# Resumen

* Hay instrucciones que hacen la suma, la resta y operaciones lógicas.
* Las instrucciones aritméticas y lógicas tienen tres operandos, pero hay instrucciones en que los tres operandos son todo registros y otros en que uno de los operandos es un valor constante.
* Hay una instrucción, *syscall*, que permite realizar operaciones de entrada y salida.
* En el ensamblador se proporcionan pseudoinstrucciones para facilitarnos la programación y hacer más legible el código resultante.

* Las etiquetas pueden contener direcciones o valores numéricos, depende de la instrucción que los utiliza.