Práctica 2
Aritmética de enteros (1), operaciones lógicas y Entrada/Salida

Objetivos

- Tomar contacto en algunas funciones del sistema de entrada y salida
- Conocer instrucciones aritméticas de enteros
- Conocer las instrucciones lógicas
- Introducir la codificación de instrucciones

Materiales

Simulador MARS y un código fuente de partida

Desarrollo práctico

1. Input y Output

El simulador del MIPS contiene un sistema operativo rudimentario que permite en los programas pedir operaciones de E/S. De esta manera podemos interactuar con el usuario haciendo lecturas de datos desde el teclado o imprimiendo valores en la pantalla. Esto se hará usando la instrucción `syscall` (mirad en la ventana `Mars Help> MIPS>Syscalls`). La instrucción `syscall` nos permite llamar a los servicios del sistema que nos darán acceso a los distintos dispositivos de entrada y de salida como pueda ser la consola del usuario, el teclado y otros dispositivos externos. El valor que haya lo registro *$v0* nos permite seleccionar una entre todas las funciones de los sistemas de entrada y salida. En algunos casos habrá que pasar parámetros o recibirlos del sistema de entrada y salida, en esos casos se usarán determinados registros. Por lo tanto, la secuencia de invocación a una función del sistema será colocar en *$v0* el identificativo de la función y a continuación hacer la llamada a la instrucción `syscall`.

Primer ejemplo: imprimir en la consola un valor entero. Se utilizará la función 1 (`print integer`). Si metemos un 1 en el registro *$v0* se escribe el valor del entero contenido en *$a0* en la consola Run I/O

```
# Imprimir en consola

.text 0x0040000
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*.

Segundo ejemplo: Leer de teclado e imprimir en consola. Para leer se utiliza la función 5 (`read integer`). Si metemos 5 en lo registro *$v0* se lee un entero del teclado. El valor leído se almacenará en *$v0*.

```
# 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.

Tercer ejemplo: 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.

```
# 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

addi $v0,$0,10      #Función 10, exit
syscall             #Acaba el programa
```

Ejercicios

- Haz un código que lee un valor *x* de teclado y escribe *x+1* en la consola.
- Haz un código que lee un valor *x* de teclado y escribe *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 donde 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:

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

- ¿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 siguiente 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.

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

Código de partida: Aritmética de enteros

```
# 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
```

Análisis previo del código

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

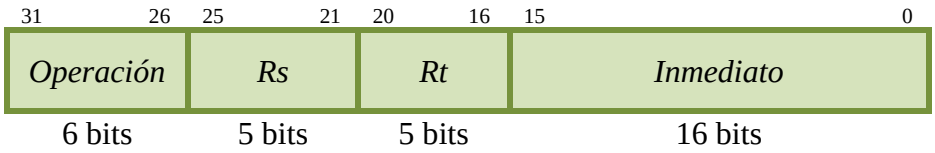
Todas las instrucciones MIPS requieren 32 bits para codificarse (1 palabra= 4 bytes= 32 bits). Los seis bits superiores de la palabra, es decir, los bits de más a la izquierda (bits 26 -31) contienen el código de operación que especifica qué operación hará la instrucción. Hay 2^6=64 códigos de operación 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 un mismo formado de instrucción, el 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 el siguiente:

Tipo I



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

- Observa el código de partida *Aritmética de enteros* del apartado anterior. ¿Cómo se codifica la primera instrucción? Hacedlo a mano (el código de operación de la instrucción `addiu` es 0x09)
- Confirmad con el programa ensamblado que el código máquina es el mismo.

Las instrucciones que utilizan 3 registros como parámetros siguen un formato de instrucción distinto. La instrucción `add $t1, $s1, $s2` se codifica así:

Codi op (6 bits)	Rs (5 bits)	Rt (5 bits)	Rd (5 bits)	Shamt(5 bits)	Función (6 bits)
A-L	$s1	$s2	$t1	-	suma
0	17	18	9	0	32
0000 00	10001	10010	0 1001	0 0000	10 0000

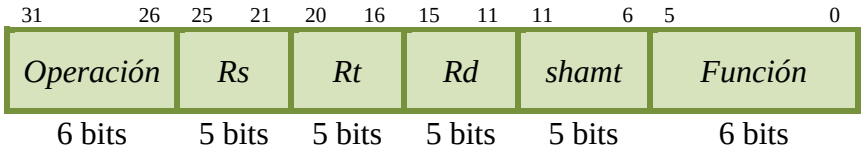
En hexadecimal quedaría como: `0x02324020`.

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* especifica al 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 una suma y por lo tanto el valor es 32 (0x20).

- ¿Cómo se codifica la última instrucción de resta del código de partida *Aritmética de enteros* que acabamos de escribir? Hacedlo a manos (el campo función de la resta es 0x22)
- Confirmad con el programa ensamblado que el código máquina es el mismo.

La forma general del formato tipo R es el siguiente:

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. El último campo de 6 bits es la función y nos sirve para distinguir las distintas instrucciones de formado tipos R.

- Notad que hay 64 instrucciones distintas con formato tipo R. ¿Por qué?

Ejercicios

- Escribe el código que hace estas acciones haciendo uso de las instrucciones estudiadas:

  $t0=5
  $t1=$t0+10
  $t2=$t0+$t1

$t3=$t1-30

- Ensambla 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`?
- Haz el código que lea dos números x e y, y obtén por pantalla el valor de la suma `x+y`
- Haz el código que lee dos números x e y, y obtén por pantalla `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 con los dos operandos como registros fuentes y en ese caso siguen el formado tipo R, o con un operando fuente registro y un valor inmediato como el otro operando fuente. En ese caso siguen el formado tipo I. Las instrucciones lógicas son: `and`, `andi`, `or`, `ori`, `xor`, `xori nor`.

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

- ¿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*?
- ¿Sería o no ventajoso en cuanto al coste temporal el hecho de utilizar el operador lógico `ori` en vez de la instrucción `addi`?. Razónalo.
- Reescribid el código de partida *Aritmética de enteros* del apartado 2 cambiando *addi* o *addiu* por *ori*:
- ¿Cuál es el valor del código de operación de la instrucción `ori`?

En la siguiente tabla 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

Ejercicios

- 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.
- Supón que $t1=0x0000FACE, utilizando únicamente las instrucciones lógicas de la tabla anterior, escribe el código que reordene los bits de *$t1* de manera que en *$t2* aparezca el valor 0x0000CAFE. Ensambla y escribe en la ventana de registros $t1=0x0000FACE. Ejecuta y comprueba que el código es correcto.

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`). Ejemplo:

```
# 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
```

Ejercicios a entregar

- Modifica el código del último ejercicio del apartado 4 para que aparezca en la pantalla el contenido del registro $t2=0000CAFE.
- Escribe el código que lee un valor entero por teclado y escribe el mismo valor en binario por la consola.

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.