

# TEMA 2 “Lenguaje del computador MIPS”

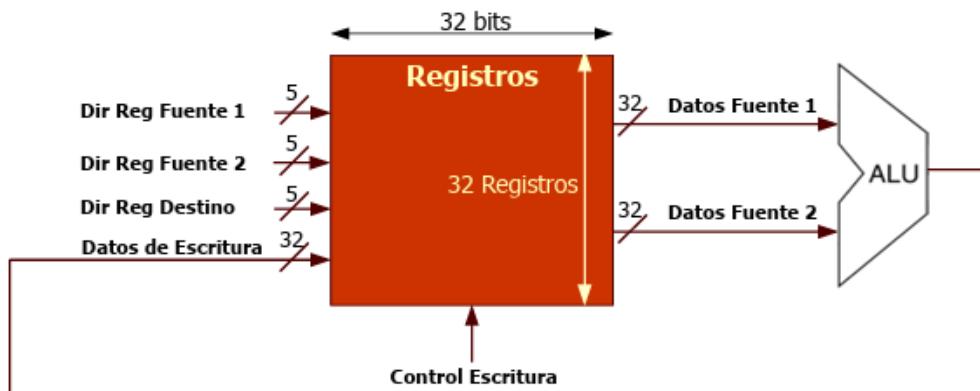
## 1. Introducción al procesador MIPS

### Arquitecturas con Acumulador

- Los primeros computadores tenían un único registro para instrucciones aritméticas (acumulador).
- La principal desventaja de este tipo de arquitectura es que todas las variables del programa deben estar ubicadas en memoria.
  - Mientras más accesos a memoria, más lentitud en la ejecución de los procesos.
- El siguiente paso es utilizar un banco de registros de propósito general en donde se encuentren todas las variables.

### Procesador MIPS

- Diseñado en la Universidad de Stanford por el equipo de John L. Hennessy.
- Procesador que utiliza una arquitectura de registros de propósito general.
  - La CPU contiene un banco de registros compuesto por 32 registros de 32 bits cada uno.
- Utiliza arquitectura RISC:
  - Instrucciones de tamaño fijo y presentadas en un reducido número de formatos.
  - Sólo las instrucciones de carga y almacenamiento acceden a la memoria de datos.
- El banco de registros está formado por 32 registros de 32 bits.
  - Dos puertos de lectura.
  - Un puerto de escritura.



- Esto proporciona mayor flexibilidad a la hora de realizar las instrucciones.
  - Implicará plantearse nuevos formatos de instrucción.

### 1.1. Operandos en registros

- El lenguaje ensamblador MIPS presenta la siguiente notación:  
add a, b, c # La suma de b y c se pone en a.
- Para realizar la misma operación utilizando el lenguaje ensamblador de la Computadora Mejorada sería:  
CRA  
ADD dir(b)  
ADD dir(c)  
STA dir(a)

- Otro ejemplo de lenguaje ensamblador MIPS sería:  
sub a, b, c # La resta de b y c se pone en a.
- En los ejemplos anteriores, las variables de las instrucciones MIPS se encuentran en registros.
- Los registros están numerados de 0 a 31, pero su representación simbólica está formada por dos caracteres precedidos por el símbolo '\$'. Por ejemplo:
  - \$s0, \$s1, ..., \$s7: Registros utilizados para valores salvados (se corresponden con variables de programas).
  - \$t0, \$t1, ..., \$t9: Registros utilizados para valores temporales.
  - \$zero: Registro cuyo valor siempre es cero -> registro no modifiable.
- Como ejemplo, supongamos que se quiere compilar la sentencia:  
 $f = (g + h) - (i + j)$  donde f, g, h, i y j están asignadas a los registros \$s0 a \$s4.  
add \$t0, \$s1, \$s2 # El registro \$t0 contiene g+h.  
add \$t1, \$s3, \$s4 # El registro \$t1 contiene i+j.  
sub \$s0, \$t0, \$t1 # f se carga con \$t0 - \$t1.

## 1.2. Operandos en memoria

- Cuando aparecen variables con estructuras complejas, con un número de elementos superior al número de registros (por ejemplo, una tabla), hay que definir un formato nuevo para poder trabajar con estos datos.
- MIPS debe incluir instrucciones que transfieran datos entre memoria y registros.
  - En este caso, un registro contiene la dirección base de la tabla, es decir, la dirección de su primer elemento. Este registro es denominado '*registro índice*'.
- La memoria principal utilizada por MIPS es de 32 bits por palabra y utiliza 32 bits para ser direccionada.
  - El elemento mínimo direccionable es el byte. Como cada palabra contiene 4 bytes, los dos últimos bits de dirección se encargarán de seleccionar el byte dentro de la palabra.
  - Por tanto, las direcciones de palabras consecutivas difieren en 4.
- La instrucción de transferencia que mueve datos de memoria a algún registro se denomina *lw (load word)*.
- Su instrucción complementaria, encargada de transferir datos de un registro a memoria, se denomina *sw (store word)*.
- Como ejemplo, supongamos A una tabla de 100 palabras y que el compilador tiene asociadas las variables g y h a los registros \$s1 y \$s2. Además, supongamos que la dirección base de A se encuentra en \$s3. Si queremos encontrar la correspondencia en ensamblador de:

$$g = h + A[8];$$

Tenemos lo siguiente:

```
lw $t0, 32($s3) # El registro $t0 se carga con A[8].
add $s1, $s2, $t0 # g = h + A[8].
```

Donde la dirección de A[8] se consigue sumando la dirección base (que está en \$s3) más el número del elemento que se quiere acceder (como las direcciones van de 4 en 4 -> 4 x 8 = 32).

## 2. Formatos de instrucción

Las instrucciones MIPS son de 32 bits y están divididas en los siguientes campos:

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Donde

- $op$ : Código de operación.
  - $rs$ : Primer registro operando fuente.
  - $rt$ : Segundo registro operando fuente.
  - $rd$ : Registro operando destino.
  - $shamt$ : Tamaño del desplazamiento (*SHift AMounT*).
  - $funct$ : Función. Selecciona la variante específica de la operación  $op$ .

- En las instrucciones `lw` y `sw` sólo se usan dos direcciones de registro. Para especificar la constante que se sumará al registro índice podríamos utilizar el campo del registro que no usamos, pero sería insuficiente.
  - Por eso es necesario definir un nuevo formato de instrucción:

op	rs	rt	dirección
6 bits	5 bits	5 bits	16 bits

- El compromiso de simplicidad requiere que, de haber más de un formato, sean de la misma longitud.
  - El formato que usa dos registros operando fuente y un registro operando destino se llama *Tipo-R* (de registro), mientras que el último descrito se denomina *Tipo-I*.
  - Los tres primeros campos de ambos formatos coinciden.
    - En el caso del Tipo-R, el campo rt se refiere al segundo registro operando fuente, mientras que en el Tipo-I se refiere al operando destino.

### **3. Instrucciones para la toma de decisiones**

- El lenguaje ensamblador MIPS incluye dos instrucciones de toma de decisiones, similares a una sentencia *if* con un *go to*.  
    beq registro1, registro2, L1
    - La instrucción *beq (branch if equal)* realiza un salto a la sentencia L1 si el valor del registro1 es igual al del registro2.  
        bne registro1, registro2, L1
    - La instrucción *bne (branch if not equal)* realiza un salto a la sentencia L1 si el valor del registro1 no es igual al del registro2.
  - Estas instrucciones suelen ir combinadas con una instrucción de comparación. Concretamente *slt (set on less than)* se comporta del siguiente modo:  
    slt \$t0, \$s1, \$s2 # \$t0 = 1 si \$s1 < \$s2  
                      # \$t0 = 0 en caso contrario
  - Imaginemos que queremos realizar lo siguiente:  
    if (*i*<*j*) *f*=*g*+*h*;  
    *f*=*f*-*j*;

Donde los valores de f.....i están en los registros \$s0.....\$s4.

Donde los valores de i,...,j estan en los rango s  $\text{lt } \$t0 \; \$s3 \; \$s4 \quad \# \; \$t0=1 \; si \; \$s3 < \$s4$

~~See \$t0, \$30, \$34 // \$t0=PSI \$30 < \$34  
beq \$t0 \$zero 11 # Salta a l1 si \$t0 = \$zero~~

`seq $t0, $zero, $t1 # Satu  
add $s0 $s1 $s2 # f=0+h`

add \$s0, \$s1, \$s2 # f=g+i  
l1: sub \$s0 \$s0 \$s3 # f=f-i

- En la primera instrucción, \$t0=1 si  $i < j$  y \$t0=0 en caso contrario. Lo que queremos es que realice  $f = g + h$  cuando se cumpla la condición y, cuando no lo haga, se la salte. De ahí que utilicemos la instrucción beq, comparando \$t0 con el registro \$zero.

## 4. Instrucciones inmediatas

- Debido a que los programas utilizan muchas veces constantes, sería interesante poder utilizarlas de forma eficiente.
- Por tanto, se crean versiones de las instrucciones aritméticas en donde uno de los operandos es una constante, que viene codificada en la misma instrucción.
  - Utilizando el formato Tipo-I, rs sería el registro donde está el primer operando, rt el registro destino y la constante estaría codificada en los 16 bits destinados a la dirección.

```
addi $s0, $s1, 5    # $s0 = $s1 + 5
slti $t2, $s3, 8    # $t2 = 1 si $s3 < 8
```

- Puede ocurrir que queramos utilizar constantes de más de 16 bits. Para ello se utiliza la instrucción lui (*load upper inmediante*). Esta instrucción carga los 16 bits codificados en la propia instrucción en los 16 bits más significativos del registro destino.
- Por ejemplo, 270000d = **00000000000000100** 0001111010110000b.
  - Los 16 bits más significativos son **000000000000100b** = **4d**.
  - Los 16 bits menos significativos son **0001111010110000b** = **7856d**.
- Para poder cargar la constante 270000d en un registro, se podría hacer:

```
lui $s0,4      # $s0=00000000000000100 000000000000000000
addi $s0,$s0,7856 # $s0=00000000000000100 0001111010110000
```

## 5. Saltos incondicionales

Dentro de un programa, al dividir su curso en varios rumbos, es inevitable tener que saltar a otras partes del programa de forma incondicional.

- La instrucción de salto incondicional es j (*jump*).
   
j Fin # Ir a Fin

Como interesa aprovechar el mayor número de bits posible para establecer la dirección de salto, y teniendo en cuenta que ningún registro es utilizado, se plantea el tercer y último formato de instrucción, que es el formato *Tipo-J*:

op	dirección objetivo
6 bits	26 bits

## 6. Operaciones lógicas

Además de las operaciones aritméticas, hay una serie de operaciones lógicas dentro del repertorio MIPS. Operaciones de desplazamiento lógico:

```
sll $s0, $s1, 4  # Transfiere el valor de $s1 desplazado
                  # 4 bits a la izquierda a $s0
srl $s0, $s1, 4  # Transfiere el valor de $s1 desplazado
                  # 4 bits a la derecha a $s0
```

Operaciones AND lógica y OR lógica:

```
and $t0, $t1, $t2 # $t0 = $t1 & $t2
or $t0, $t1, $t2 # $t0 = $t1 | $t2
```

Operaciones AND lógica y OR lógica inmediatas:

```
andi $t0, $t1, 0xFF00
ori $t0, $t1, 0xFF00
```

La AND lógica es útil a la hora de enmascarar bits.

- Se seleccionan algunos bits y el resto se ponen a cero.

Ejemplo de funcionamiento de la AND lógica:

and \$t0, \$t1, \$t2

\$t2	0000 0000 0000 0000 0000 1101 1100 0000
\$t1	0000 0000 0000 0000 0011 1100 0000 0000
\$t0	0000 0000 0000 0000 0000 1100 0000 0000

La OR lógica es útil a la hora de incluir bits en una palabra.

- Activa una serie de bits a 1, dejando el resto inalterado.

Ejemplo de funcionamiento de la OR lógica:

or \$t0, \$t1, \$t2

\$t2	0000 0000 0000 0000 0000 1101 1100 0000
\$t1	0000 0000 0000 0000 0011 1100 0000 0000
\$t0	0000 0000 0000 0000 0011 1101 1100 0000

## 7. Carga y almacenamiento de bytes

A la hora de manejar caracteres, MIPS utiliza código ASCII.

- Esa es la razón por la que el elemento mínimo direccionable sea el byte.

Para cargar/almacenar bytes entre registros y memoria, existen instrucciones equivalentes a lw y sw, pero que solamente transfieren un único byte.

lb \$t0, 2(\$s1) # Carga un byte de memoria

- La instrucción lb (*load byte*) carga el byte seleccionado de memoria y lo transfiere a los 8 bits menos significativos del registro destino.  
sb \$t1, 4(\$s2) # Guarda un byte en memoria
- La instrucción sb (*store byte*) carga los 8 bits menos significativos del registro fuente y los transfiere al byte seleccionado en memoria.

## 8. Instrucciones de llamada a procedimientos

Para llamar a un procedimiento o subrutina, el lenguaje ensamblador MIPS incluye la instrucción jal (*jump and link*).

- Esta instrucción salta a la dirección donde continúa el programa, guardando la dirección de retorno.

jal DirecciónDelProcedimiento

El salto lo efectúa igual que la instrucción j. La dirección de retorno (encontrada en el PC) es almacenada en el registro \$ra, que es utilizado específicamente para esta tarea.

Para poder realizar el salto de retorno, necesitamos una instrucción que realice un salto a la dirección contenida en un registro. Esa instrucción es jr (*jump registry*).

jr \$t1 # Salta a la dirección contenida en \$t1

## 9. Pseudoinstrucciones

El ensamblador MIPS acepta variaciones de las instrucciones de su repertorio, aunque estas variaciones no estén implementadas en el hardware.

La instrucción *move* transfiere el valor de un registro a otro.

```
move $t0, $t1      # $t0 se carga con el valor de $t1
```

En este caso el ensamblador convierte esta instrucción en la siguiente instrucción sí implementada:

```
add $t0, $t1, $zero # $t0 = $t1 + 0
```

Otro caso parecido es el de la instrucción *blt* (*branch if less than*)

```
blt $s1, $s2, L1  #Salta a L1 si $s1<$s2
```

El ensamblador MIPS convierte esta instrucción en las siguientes instrucciones:

```
slt $at, $s1, $s2 # $at=1 si $s1<$s2  
bne $at, $zero, L1 # Salta a L1 si $at != 0
```

Del mismo modo se podrían realizar *ble* (*branch if less than or equal to*), *bgt* (*branch if greater than*) y *bge* (*branch if great than or equal to*).

El único coste para poder realizar estas instrucciones es la reserva de un registro *\$at*, para ser usado por el ensamblador.

Por último, otra de las pseudoinstrucciones más usadas es *li* (*load immediate*).

Esta instrucción se ocupa de cargar una constante de 32 bits en un registro destino:

```
li $s1, 270000  # $s1 = 2700000
```

El ensamblador MIPS divide la constante de 32 bits en dos bloques de 16 bits. Los más significativos los carga mediante *lui* en el registro destino y los 16 menos significativos los transfiere mediante *ori*:

```
lui $s1, 4  
ori $s1, 7856
```

(Los datos 4 y 7856 combinados forman 2700000, tal y como se pudo ver en la ilustración de *lui* una serie de transparencias más atrás).

## Registros del procesador MIPS

Nº de reg	Nombre	Uso	Preservado en las llamadas
0	\$zero	Valor constante cero	No aplicable
1	\$at	Reservado por el ensamblador	No
2-3	\$v0-\$v1	Valores para resultados de las llamadas a procedimiento y evaluación de expresiones	No
4-7	\$a0-\$a3	Argumentos	No
8-15	\$t0-\$t7	Temporales	No
16-23	\$s0-\$s7	Salvados	Sí
24-25	\$t8-\$t9	Más temporales	No
26-27	\$k0-\$k1	Reservados al núcleo del sistema operativo	No
28	\$gp	Puntero global	Sí
29	\$sp	Puntero de pila	Sí
30	\$fp	Puntero de bloque de activación	Sí
31	\$ra	Dirección de retorno	Sí