**Ensamblador**

**Introducción**

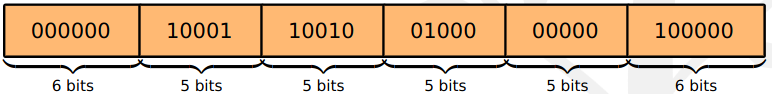
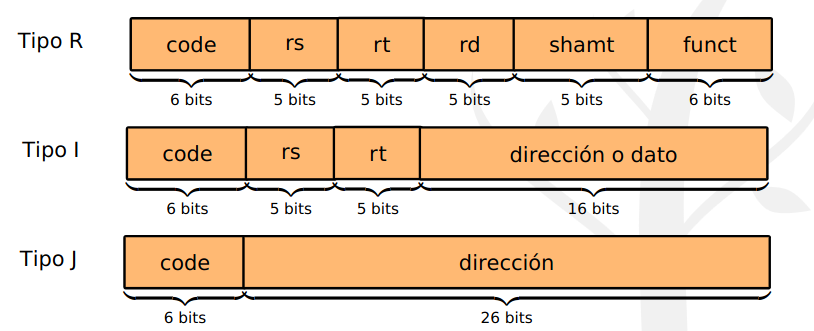
* El lenguaje **ensamblador** es una representación simbólica del código máquina en binario que utiliza el ordenador.
* **PC:** Contador de programa. Indica la dirección de la próxima línea de código a ejecutar.
* **IR:** Registro de instrucción: Contiene el contenido de la última línea de código ejecutada.
* Los programas se dividen en dos partes:
  + .text : sección obligatoria, contiene el código. Empieza siempre con:

.globl main

.main:

* + .data: opcional (normalmente necesaria). Contiene declaraciones de variables
    - Las variables se declaran con var: .tipo

**Instrucciones MIPS**

* Longitud de instrucción, registros y palabras fija de **32 bits[[1]](#footnote-0).**
* Una **instrucción** está organizada en **campos**, distintos según el tipo de operación.
  + **Tipo R:** Instrucciones aritmético/lógicas. Ejemplo: add $t0, $s1, $s2
  + **Tipo I:** Instrucciones de transferencia de datos, saltos, condicionales, operandos inmediatos
  + **Tipo J:** Instrucciones de salto incondicional.

**Direccionamiento**

* Indica la localización de los operandos de la instrucción.
* **Modos:**
  + **Inmediato:** el operando es una constante en un campo de la instrucción.
    - Operaciones AL de tipo inmediato. Ejemplo: addi $t1, $t0, 20
  + **Registro:** El operando está en un registro cuya ubicación se indica en la instrucción.
    - Operaciones aritmético-lógicas. Ejemplo: add $t2, $t0, $t1
  + **Base con desplazamiento:** El operando está en una dirección de memoria, que se obtiene de la suma del desplazamiento + el contenido de un registro base.
    - Operaciones de transferencia de datos. Ejemplo: lw $t1, 20($t0)
  + **Relativo al PC**: El operando está en una dirección de memoria, que se obtiene de la suma del contador del programa + el desplazamiento.
    - Operaciones de salto condicional. Ejemplo: beq $t1, $t0, -25
  + **Pseudodirecto:** El operando está en una dirección de memoria, obtenida como la concatenación de 26 bits indicados en la instrucción y los 4 bits más significativos del contador de programa. Ejemplo: j salto
    - Operaciones de salto incondicional.Ejemplo: j 1200

**Tipos de variables**

* **.ascii** “string a almacenar”: Almacena lista de caracteres
* **.asciiz** “string a almacenar”: Almacena lista de caracteres acabando con 0. Cada caracter ocupa 1 byte, se debe considerar también el espacio del \0.
* **.word** palabra, palabra2, …: Almacena lista de palabras en posiciones contiguas. Cada palabra ocupa 4 bytes.
* **.space** n: Reserva n bytes de espacio en la memoria.
* **.float f1, f2, …:** Almacena lista de números en punto flotante
* **.double d1, d2, …:** Almacena lista de números en punto flotante con prec doble

**Registros**

| Número | Nombre | Uso convencional |
| --- | --- | --- |
| $0 | $zero | Valor constante 0 |
| $1 | $at | Reservado por ensamblador |
| $2-$3 | $v0-$v1 | Resultados y evaluación de expresiones. |
| $4-$7 | $a0-$a3 | Argumentos de función. Usados por **syscall**. |
| $8-$15 | $t0-$t7 | Temporales. |
| $16-$23 | $s0-$s7 | Salvados. |
| $24-$25 | $t8-t9 | Temporales |
| $26-27 | $k0-$k1 | Reservado para núcleo de SO |
| $28 | $gp | Puntero global. N/A |
| $29 | $sp | Puntero de pila (ver pila) |
| $30 | $fp | Puntero de bloque de activación. N/A |
| $31 | $ra | Dirección de retorno de subrutina (ver jal) |

**Instrucciones aritmético-lógicas**

* **add/sub/and/or $1, $2, $3:** Almacena en $1 el valor de la operación correspondiente de $2 y $3.
* **addi/subi/andi/ori $1, $2, cte:** Almacena en $1 el valor de la operación correspondiente de $2 y la constante.
* **sll $t0,$s1,4**: Almacena en $t0 el resultado de desplazar $s1 4 posiciones a la izquierda.
* Para producto y división se usan **HI** y **LO**, registros para operar con operandos de 64 bits.
  + **div $1, $2** almacena en HI el resto y en LO el cociente de dividir $1 entre $2
  + **mult $1, $2** almacena en LO el resultado de multiplicar $1 por $2
  + **mfhi/mflo $1** copian el valor de HI en $1 y viceversa.
* **slti $1, $2, cte.**: Almacena en $1 un 1 si el valor del registro $2 es menor que cte. Se usa junto con beq y bne para crear blt, bgt, ble, bge.

**Instrucciones de carga**

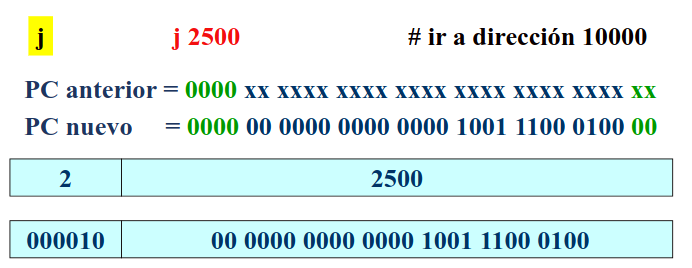
* **lui** **$1, cte10:** Carga la constante en los 16 bits más significativos del registro $1. La etiqueta puede tener signo y se carga en CA2. Parte de la.
* **lw $1, d($2):** Carga en el registro $1 la palabra almacenada en la dir. de memoria [(direccion almacenada en $2) + d]. d debe ser múltiplo de 4.
* **sw $1, d($2):** Carga en la dir. de memoria [(direccion almacenada en $2) + d] la palabra almacenada en $1. d debe ser múltiplo de 4.
* **lb $1, d($2):** Carga en el registro $1 el byte almacenado en la dir. de memoria [(direccion almacenada en $2) + d], sobreescribiendo el contenido de $1. d no es necesariamente múltiplo de 4.
  + Tener en cuenta que es little endian. Por ejemplo, si en la dirección que indica ($2) hay 0x1b2b3b4b, lb $1, 3($2) cargará 0x0000002b
* **sb $1, d($2):** Carga en la dir. de memoria [(direccion almacenada en $2) + d] el primer byte almacenado en $1 . d no es necesariamente múltiplo de 4.

**Pseudoinstrucción[[2]](#footnote-1) la [load adress]**

* **la $tn, variable:** carga en el registro $tn la dirección de memoria donde está almacenada la variable.
* Una dirección de memoria ocupa 32 bits, y una instrucción en total puede ocupar 32 bits, de los cuales solo 16 están disponibles.
* La ejecución de está pseudoinstrucción se divide en dos instrucciones, cargándose los bits de 16 en 16.
  + **lui $tn, [primeros 16 bits direccion memoria variable]**: Carga la constante en los 16 bits más signicativos del registro $tn.
  + **ori $tn, $tn, 0x000[ultimos 16 bits]:** Almacena en tn el resultado de una operación OR entre tn y lo. Permite cargar en t0 los 16 bits menos significativos.
* Ejemplo: siendo a=12345678 y b=fecdba98, y sus direcciones de memoria 0x10010000 y 0x1001004, respectivamente.

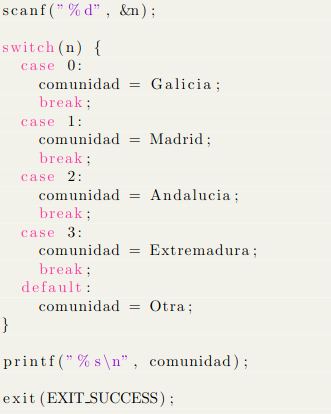
| Código (editor) | Código (máquina) | Contenido PC | $t0 | $a1 | $a2 |
| --- | --- | --- | --- | --- | --- |
| la $t0, a | lui $t0, 0x1001 | 00400004 | 10010000 | 00000000 | 000000000 |
| ori $t0, $t0, 0x0000 | 00400008 | 10010000 | 00000000 | 000000000 |
| lw $a1, 0($t0) | lw $a1, 0($t0) | 00400012 | 10010000 | 12345678 | 000000000 |
| la $t0, b | lui $t0, 0x1001 | 00400016 | 10010000 | 12345678 | 000000000 |
| ori $t0, $t0, 0x0004 | 00400020 | 10010004 | 12345678 | 000000000 |
| lw $a2, 0($t0) | lw $a2, 0($t0) | 00400024 | 10010004 | 12345678 | fecdba98 |

**Saltos**

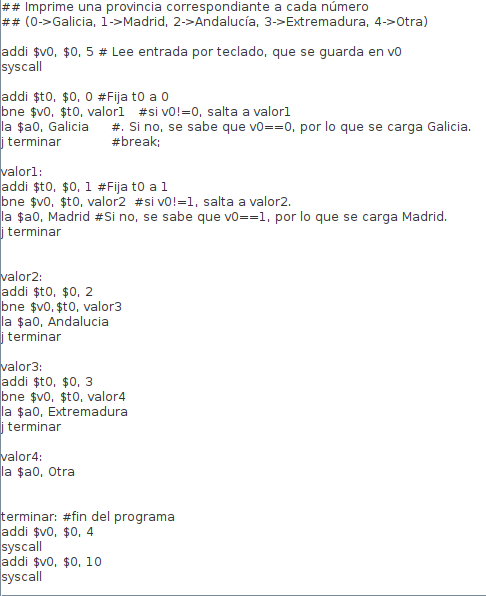
* Hacen que el programa salte a otra instrucción que no sea la siguiente en memoria.
* **Salto incondicional (j, jr, jal):** Saltan siempre.
  + No varían los 4 primeros bits del PC, almacenan el valor de la constante en los 26 siguientes, y los últimos 2 bits son siempre 00.
  + 
  + **jr $s1** avanza a la dirección contenida en $s1. Es de tipo R.



* **Salto condicional (beq, bne):** Saltan si se cumple una condición
  + **beq $s1,$s2,25:** Si $s1==$s2, se suma al PC el valor de la etiqueta multiplicada por 4, en CA2 (admite valores negativos)



**Ejemplo: Implementación de switch()**



**Subrutinas**

* Se realizan con saltos y pasando parámetros mediante registros.
  + Los parámetros de entrada se facilitan en **$a0-$a3**
  + Los resultados se guardan en **$v0-$v1**
* Se usa la instrucción **jal**, que realiza un salto y además guarda la dirección de retorno en el registro 31 ($ra)
  + Para volver al programa principal, se utilizaría **jr $ra**

**Pilas**

* Parte de la memoria utilizada como una estructura, utilizada para mantener el estado del sistema al llamar a una subrutina.
  + Existen dos posibles convenios: puede ser el invocador o el invocado el responsable de guardar/restaurar.
  + El contenido de los registros temporales ($t0-$t9) debe ser guardado antes de llamar a la subrutina.
  + El contenido de los registros salvados ($s0-$s7) debe ser preservado por la subrutina utilizando la pila.
* Crece de posiciones altas a bajas de memoria. La última posición ocupada es indicada por el **stack pointer** (registro 29, $sp)
* Para realizar **push:**
  + Se resta 4 al puntero de la pila [addi $sp,$sp,-4]
  + Se guarda el valor mediante sw [sw $t0,0($sp)]
  + Se debe restar antes y guardar después. De esta forma, la primera dirección de la pila (0x7000000) estará vacía siempre.
* Para realizar **pop:**
  + Se recupera el valor mediante lw [lw $t0,0($sp)]
  + Se suma 4 al puntero de la pila [addi $sp,$sp,4]

**Syscall**

* Syscall realiza una llamada al sistema operativo para realizar una función, determinada por el contenido del registro $v0.

1. **print\_int:** Imprime como entero el contenido de $a0
2. **print\_float:** Imprime como float el contenido de $f12
3. **print\_double:** imprime como float el contenido de $f12 y $f13 unidos
4. **print\_string**: Imprime como string el contenido en la posición que indica $a0
5. **read\_int:** Pide por teclado un entero que se almacena en $a0
6. **read\_float:** Pide por teclado un float que se almacena en $f0
7. **read\_double:** Pide por teclado un float que se almacena en $f0 y $f1
8. **read\_string:** Pide por teclado un string que se almacena en $a0 y su longitud en $a1

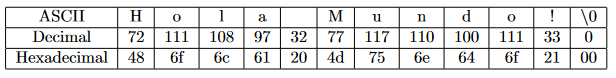
**10. exit:** Finaliza la ejecución.

**Arrays**

* Los elementos de un vector se almacenan de forma contigua
* El elemento [0] es el primero en escribirse
* Para cargar el elemento [i] de un vector, se utiliza lw $registro 4\*i($registro), sustituyendo 4 por el tamaño en memoria del elemento

| Código | Contenido de registros |
| --- | --- |
| la $t0, vector | $t0: dirección de vector |
| lw $s0, 0($t0) | $s0: valor [0] de vector |
| add $s1, $0, $s0 | $s1: valor[0] de vector |
| lw $s0, 4($t0) | $s0: vector[1] |
| add $s1, $s1, $s0 | $s1: vector[0]+vector[1] |
| lw $s0, 8($t0) | $s0: vector[2] |
| add $s1, $s1, $s0 | $s1: vector[0] +vector[1]+vector[2] |
| lw $s0, 12($t0) | $s0: vector[3] |
| add $s1, $s1, $s0 | $s1: vector[0]+vector[1] + vector[2] + vector[3] |
| la $t0, resultado | $t0: direccion de resultado |
| sw $s1, 0($t0) | Guarda en resultado el valor de s1 |
| la $a0, mensaje | $a0: direccion de mensaje |
| addi $v0, $0, 4 | $v0: 4 |
| syscall | Imprime mensaje |
| add $a0, $s1, $0 | $a0: valor de s1 |
| addi $v0, $0, 1 | v0: 1 |
| syscall | Imprime s1 |
| addi $v0, $0, 10 | v0: 10 |
| syscall | finaliza programa |

**Memoria**

* Los números en memoria se almacenan en hexadecimal, en complemento a 2 con signo. Cada word ocupa 4 bytes (32 bits).
* Los strings se almacenan en memoria contigua, divididas en palabras de hasta 4 bytes [32 bits]. Cada char ocupa un byte.
  + Dentro de cada palabra los caracteres están de derecha a izquierda (caracter de la izquierda en posicion menos significativa). Sigue una configuración **Little Endian**. Ejemplo:



|  | b3 | b2 | b1 | b0 |
| --- | --- | --- | --- | --- |
| 0x10010000 | 61 | 6c | 6f | 48 |
| 0x10010004 | 6e | 75 | 4d | 20 |
| 0x10010008 | 00 | 21 | 6f | 64 |

(se en lugar de escribir .asciiz “hola mundo” se escribe directamente .word 0x616c6f48, gárdase a palabra nese mismo orden, 0x616c6f48)

1. (32 bits=4 bytes) Esto implica que todas las direcciones de memoria deben ser múltiplos de 4, y que la dirección de una palabra es la de su primer byte. [↑](#footnote-ref-0)
2. Otras pseudoinstrucciones: li (carga de 32 bits), blt/ble/bgt/bge (variaciones de bne y slti), move [↑](#footnote-ref-1)