ESTRUCTURA DE COMPUTADORES
Grado en Ingeniería Informática

Sesión de laboratorio número 3

FUNCIONES Y LLAMADAS AL SISTEMA

Objetivos

- Entender los tres materiales con que se hace un programa en código máquina (instrucciones, datos y funciones de sistema).
- Construir funciones simples y llamarlas desde un programa.
- Hacer inventario de instrucciones para el control de flujo.
- Conocer y hacer uso de las llamadas al sistema por medio de la instrucción máquina `syscall`.

Bibliografía

- D.A. Patterson y J. L. Hennessy, *Estructura y diseño de computadores*, Reverté, capítulo 2, 2011.

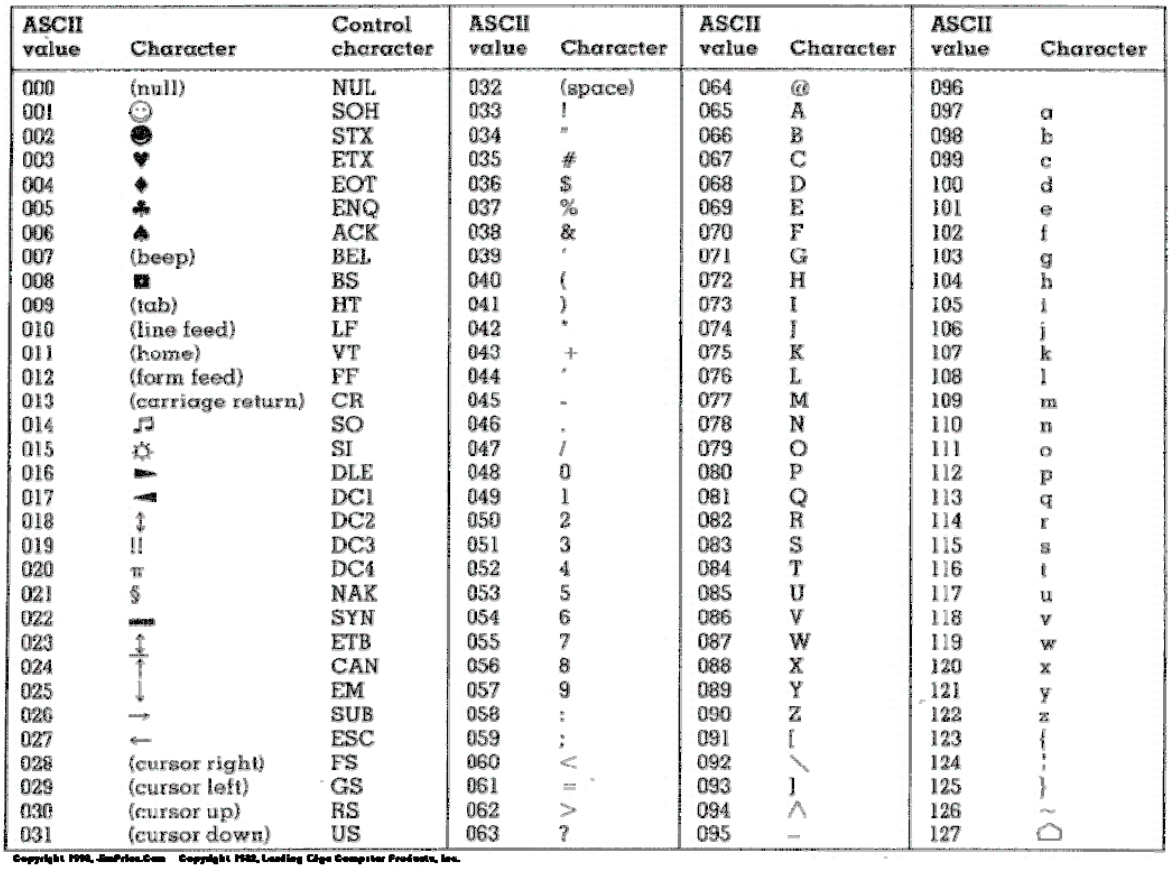
Introducción teórica

Enteros y caracteres

Ya sabemos que en un computador las cadenas o palabras de bits no tienen un significado propio. Eso significa que, por ejemplo, la palabra de bits `0x90324a00` puede interpretarse, según el contexto, como una instrucción máquina, una dirección de memoria, un número entero sin signo, un número entero con signo codificado en complemento a dos (en este caso sería negativo), un número real codificado según la norma IEEE 754 (en este caso también sería negativo), una cadena de 4 bytes de longitud, etc.

Aunque los computadores fueron diseñados originalmente para realizar gran cantidad de cálculos aritméticos, pronto fueron utilizados para procesar texto. Gran número de computadores actuales utilizan palabras de 8 bits para representar caracteres según el código ASCII (*American Standard Code for Information Exchange*). Este código tiene algunas características que siempre debemos tener en cuenta: los códigos para letras minúsculas y mayúsculas sólo difieren en un bit y, numéricamente, la posición de dicho bit hace que la diferencia cuantitativa entre los dos códigos de cada letra (mayúscula y minúscula) sea de 32.

Por ejemplo, el código para la letra “Q” es 81 y per a la letra “q” es 113 (nótese que 81+32=113, y que los dos códigos en binario son, respectivamente, 1010001 y 1110001). Otro valor importante es el cero, llamado *null*, que se utiliza en C y Java para marcar el final de una cadena de caracteres.



ASCII value	Character	Control character	ASCII value	Character	ASCII value	Character	ASCII value	Character
000	(null)	NUL	032	(space)	064	@	096	
001	☺	SOH	033	!	065	A	097	a
002	☻	STX	034	"	066	B	098	b
003	♥	ETX	035	#	067	C	099	c
004	♦	EOT	036	$	068	D	100	d
005	♣	ENQ	037	%	069	E	101	e
006	♠	ACK	038	&	070	F	102	f
007	(beep)	BEL	039	'	071	G	103	g
008	◘	BS	040	(	072	H	104	h
009	(tab)	HT	041	)	073	I	105	i
010	(line feed)	LF	042	*	074	J	106	j
011	(home)	VT	043	+	075	K	107	k
012	(form feed)	FF	044	,	076	L	108	l
013	(carriage return)	CR	045	-	077	M	109	m
014	♫	SO	046	.	078	N	110	n
015	☼	SI	047	/	079	O	111	o
016	►	DLE	048	0	080	P	112	p
017	◄	DC1	049	1	081	Q	113	q
018	↕	DC2	050	2	082	R	114	r
019	!!	DC3	051	3	083	S	115	s
020	¶	DC4	052	4	084	T	116	t
021	§	NAK	053	5	085	U	117	u
022	▬	SYN	054	6	086	V	118	v
023	↨	ETB	055	7	087	W	119	w
024	↑	CAN	056	8	088	X	120	x
025	↓	EM	057	9	089	Y	121	y
026	→	SUB	058	:	090	Z	122	z
027	←	ESC	059	;	091	[	123	{
028	(cursor right)	FS	060	<	092	\	124	¦
029	(cursor left)	GS	061	=	093	]	125	}
030	(cursor up)	RS	062	>	094	^	126	~
031	(cursor down)	US	063	?	095	_	127	⌂

Figura 1. Código ASCII con la representación de los primeros 128 carácteres.

Dicho esto, cabe añadir que, sin embargo, en la actualidad *Unicode* es la codificación universal de los alfabetos de los idiomas humanos (latín, griego, cirílico, bengalí, etíope, tailandés, y un largo etcétera). Hay tantos alfabetos en Unicode como símbolos útiles hay en ASCII. El lenguaje de programación Java utiliza Unicode para codificar caracteres. Por omisión, utiliza 16 bits para representar un carácter. Para saber más sobre Unicode consultar www.unicode.org.

Ya sabemos que el repertorio de instrucciones del MIPS de acceso a memoria incluye la posibilitad de acceder a cadenas de 32 bits (*word*) con **`lw`** y **`sw`**, de 16 bits (*half*) con **`lh`** y **`sh`** y de 8 bits (*byte*) con **`lb`** y **`sb`**. Cabe notar que las instrucciones de lectura **`lh`** y **`lb`**, dado que leen menos de 32 bits, completan los bits que faltan extendiendo el signo de la palabra leída; es decir, interpretan de manera implícita el valor leído como un entero codificado en complemento a dos.

Ahora bien, ¿qué ocurre si lo que leemos de la memoria es un carácter codificado en ASCII o Unicode? Pues que no tiene ningún sentido hablar de bit de signo puesto que el valor leído no debe interpretarse como un entero si no como un carácter. Es por eso que el repertorio de instrucciones del MIPS R2000 también incluye les variantes sin signo (*unsigned*) **`lhu`** y **`lbu`** para leer cadenas de bits más pequeñas de 32 bits donde no se debe hacer ninguna extensión de signo. Estas instrucciones hacen que los bits que faltan para rellenar el registro destino se pongan a cero. De hecho, las instrucciones **`lhu`** y **`lbu`** son más populares que **`lh`** y **`lb`**.

Control de flujo de ejecución en ensamblador

Las instrucciones de salto junto a ciertas instrucciones aritméticas permiten construir las estructuras condicionales e iterativas.

A bajo nivel, podemos distinguir entre:

- saltos incondicionales del tipo *seguir en la dirección*; por ejemplo, la instrucción **`j eti`**.
- saltos condicionales o bifurcaciones *si (condición) seguir en la dirección* donde *dirección* señala la instrucción que se ejecutaría a continuación. En el juego del MIPS, tenemos seis condiciones para saltos condicionales: nótese que se pueden hacer tres parejas de condiciones contrarias (**=** y $\neq$, **>** y $\leq$, **<** y $\geq$).

El juego de instrucciones sólo permite las comparaciones **=** y $\neq$ entre dos registros y las comparaciones **>**, $\leq$, **<** y $\geq$ entre un registro y el cero:

`beq rs,rt,A`	**`bgtz rs,A`**	**`bltz rs,A`**
rs = rt	*rs > 0*	*rs < 0*
`bne rs,rt,A`	**`blez rs,A`**	**`bgez rs,A`**
rs ≠ rt	*rs ≤ 0*	*rs ≥ 0*

Tabla 1. Instrucciones de bifurcación del MIPS

Este surtido de condiciones puede ampliarse con ayuda de la instrucción aritmética **`slt`** (*set on less than*) y las instrucciones relacionadas que se estudiarán en el tema de aritmética de enteros. Así se obtienen estas otras seis pseudoinstrucciones:

`beqz rs,A`	**`bgt rs,rt,A`**	**`blt rs,rt,A`**
rs = 0	*rs > rt*	*rs < rt*
`bnez rs,A`	**`ble rs,rt,A`**	**`bge rs,rt,A`**
rs ≠ 0	*rs ≤ rt*	*rs ≥ rt*

Tabla 2. Pseudoinstrucciones de bifurcación del MIPS

Veamos la traducción de un par de pseudoinstrucciones de salto en instrucciones máquina en la tabla siguiente:

Pseudoinstrucción	Instrucciones máquina
beqz rs,A	**beq rs,$zero,A**
bgt rs,rt,A	**slt $at,rt,rs** **bne $at,$zero,A**

Tabla 3. Traducción de las pseudoinstrucciones **beqz** y **bgt** en instrucciones del MIPS

Con estas instrucciones pueden construirse estructuras condicionales e iterativas equivalentes a las que escritas en alto nivel. Por ejemplo, si hay un bloque de instrucciones A1, A2… que sólo han de ejecutarse si el contenido de un registro **$r** es negativo, se puede utilizar una bifurcación que salte si se da la condición contraria (**$r ≥ 0**):

```
            bgez $r,L
            A1
            A2
            …
L:
```

Para iterar *n* veces un bloque de instrucciones A1, A2…, se puede utilizar un registro **$r** y escribir:

```
            li $r,n
bucle:      A1
            A2
            …
            addi $r,$r,-1
            bgtz $r,bucle
```

En el anexo puede consultarse un cuadro con la traducción de diversas estructures de control de flujo.

Funciones del programa

Las funciones del programa (*callee functions*) son la traducción de los métodos de Java o las funciones de C. La pareja de instrucciones **jal *eti*** (o llamada a función) y **jr $ra** (retorno de función), ligadas al registro **$ra** (**$31**), dan el soporte básico al flujo de ejecución.

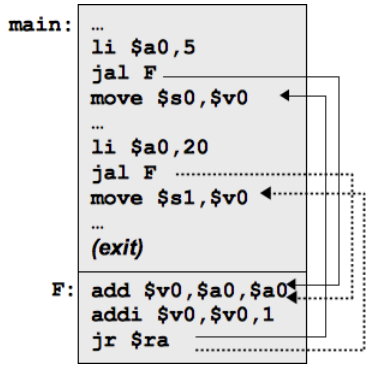


Figura 2. A la izquierda, aparece el esquema de un programa **main()** que llama desde dos puntos a una función **F** que en alto nivel se expresaría como **int F(int a){return 2*a+1}**. En los dos casos, la instrucción **jal F** guarda en el registro **$ra** la dirección de retorno, y por eso la función **F** acaba con una instrucción **jr $ra**. Las flechas de la figura muestran el flujo de ejecución: la primera llamada en continuo ⟶ y la segunda a trazos ⋯▸.
El programa principal, por su parte, acaba con la llamada al sistema **exit**.

El convenio de uso de los registros también contempla la separación entre registros del programa y registros de la función. Los registros **$s0** a **$s7** están orientados a servir de variables globales del programa, y los registros **$t0** a **$t9** a variables locales del procedimiento. El convenio dice:

- Si el programa principal utiliza un registro **$ti**, ha de prever que cualquier función que llame podrá cambiar su contenido.
- Si una función necesita escribir en un registro **$si**, deberá de preservar su contenido previamente y restaurarlo antes de terminar.
- Si una función utiliza un registro **$ti**, deberá tener en cuenta que, entre dos ejecuciones de la misma función, cualquier otra función podrá modificar su contenido.

El convenio prevé, a su vez, la comunicación entre el programa principal y la función, y la regula considerando el número y tipo de datos intercambiados. Per ejemplo, si los argumentos son de tipo entero y no hay más de cuatro, irán por orden en los registros **$a0** a **$a3**. El valor retornado por la función, si es un entero, se escribirá en el registro **$v0**.

En resumen: en los ejercicios de estas prácticas conviene seguir las reglas de la tabla siguiente a la hora de programar. Estas reglas se ampliarán más adelante para permitir que las funciones del programa puedan llamarse entre sí.

Registros	Úso
$s0…$s7	El código del programa principal
$a0…$a3	Paso de parámetros del programa a las funciones
$t0…$t9	El código de la función
$v0	Retorno de resultados de las funciones a los programas

Tabla 4. Reglas del convenio de uso de los registros por parte de las aplicaciones

Las llamadas al sistema

Las operaciones de escritura en la consola o de lectura del teclado suponen acceder a partes del computador que, por razones de seguridad y de eficiencia, no están visibles para los programas corrientes. La mayoría de los computadores, reales o simulados, disponen de periféricos y de un sistema operativo (por rudimentario que sea éste) que ofrece un catálogo de funciones. Más adelante, en temas referentes a la entrada/salida, estudiaremos los detalles.

En un MIPS, estas funciones de sistema se pueden invocar mediante la instrucción **syscall**. Cada función se distingue por un número que la identifica llamado índice, puede aceptar una serie de argumentos y devuelve un posible resultado.

A continuación se ilustra con un ejemplo el mecanismo de llamada. El código lee un número entero desde el teclado y lo copia en la dirección de memoria etiquetada con el nombre **valor**:

```
li $v0, 5          # Índice de la llamada read_int
syscall            # Llamada al sistema read_int
sw $v0, valor      # Copia el número entero en memoria
```

El catálogo de funciones del sistema simulado en PCSpim se encuentra en la tabla siguiente:

Servicio	Código de la llamada	Argumentos	Resultado
print_int	1	$a0 = integer	
print_float	2	$f12 = float	
print_double	3	$f12 = double	
print_string	4	$a0 = string	
read_int	5		integer (in $v0)
read_float	6		float (in $f0)
read_double	7		double (in $f0)
read_string	8	$a0 = buffer $a1 = length	
sbrk	9	$a0 = amount	address (in $v0)
exit	10		

Tabla 5. Tabla de llamadas al sistema.

Sin embargo, en esta práctica debemos trabajar sólo con las cuatro funciones referidas en la tabla siguiente. Nótese que el índice que identifica el tipo de servicio siempre se indica en el registro **$v0**, algunas llamadas toman el parámetro contenido en el registro **$a0** y, si devuelven un resultado, lo hacen siempre en **$v0**:

Nombre	$v0	Descripción	Argumentos	Resultado
print_int	*1*	Imprime el valor de un entero	**$a0** = entero a imprimir	—
read_int	*5*	Lee el valor de un entero	—	**$v0** = entero leido
exit	*10*	Acaba el proceso	—	—
print_char	*11*	Imprime un carácter	**$a0** = carácter a imprimir	—

Tabla 6. Funciones del sistema que deben utilizarse en esta práctica.

Por tanto, el uso de las funciones del sistema es muy parecido al de las funciones del programa; la diferencia más notable es que el código de las funciones del sistema está oculto, es independiente de los programas, es común para todos ellos y preserva el contenido de los registros globales y locales. En definitiva, no es necesario conocer la dirección donde se encuentran las funciones del sistema para poder utilizarlas.

Ejercicios de laboratorio

Ejercicio 1: Las llamadas al sistema y las funciones de los programas

Abra y observe el código siguiente contenido en el fichero “02_exer_01.s”. Note que sólo se encuentra el segmento de código (**.text**) y no hay ningún comentario. Los comentarios deberá añadirlos conforme vaya entendiendo lo que hace el programa.

```
          .globl __start
          .text 0x00400000
__start:  li $v0,5
          syscall
          move $a0,$v0
          li $v0,5
          syscall
          move $a1,$v0
          jal Mult
          move $a0,$v0
          li $v0,1
          syscall
          li $v0,10
          syscall
Mult:     li $v0, 0
          beqz $a1, MultRet
MultFor:  add $v0, $v0, $a0
          addi $a1, $a1, -1
          bne $a1, $zero, MultFor
MultRet:  jr $ra
```

En primer lugar, debe detectar qué instrucciones pertenecen al programa principal y cuáles a una función de nombre **Mult**.

- ¿Cuáles son las dos últimas instrucciones del programa principal?
- ¿Cuál es la última instrucción de la función?
- Busque las cuatro llamadas al sistema utilizadas en el programa. ¿Qué hace cada una?
- Busque un bucle dentro de la función. ¿Cuántas veces se ejecuta este bucle?
- ¿Qué hace la función exactamente?

Cargue el programa y ejecútelo. Note que la entrada/salida por la consola es muy pobre.

- ¿Sabe ejecutar el programa completo? Al ejecutarlo, tenga en cuenta que el programa pide la entrada de dos números por el teclado y luego imprime un resultado. Ahora bien, no habrá ningún mensaje que indique que se está esperando una entrada del teclado.
- ¿Sabe hacer una ejecución paso a paso?

Técnica experimental: uso de los *breakpoints*. Es muy útil para detener el programa en un punto donde conviene inspeccionar los registros o la memoria sin tener que ir paso a paso desde el principio. Simplemente se le indica al simulador la dirección de la instrucción donde ha de detenerse la ejecución. Utilice la técnica anterior para detener la ejecución dentro de `Mult` y observar el valor de la dirección de retorno contenida en el registre `$ra`. Deberá indicar como punto de ruptura del flujo de ejecución la dirección de la instrucción `jr $ra`.

- ¿Cuál es el valor de la dirección de retorno?
- ¿A qué instrucción del programa apunta?

Ejercicio 2: Creación de funciones

Vamos a mejorar el diálogo del programa anterior a través de la consola. Esta mejora consiste en asociar el símbolo de una letra a cada valor que se lea o escriba. Así, puede nombrar el multiplicando como 'M', el multiplicador como 'Q' y el producto como 'R'. Debe escribir dos funciones que añadirá al programa del apartado anterior:

- Para la introducción de valores por el teclado. La función `Input` tiene como argumento el símbolo de la letra que vamos a escribir en la consola. La función debe escribir en la consola este símbolo seguido del carácter '`=`' y después leer un entero (el multiplicador o el multiplicando). La función debe devolver este valor leído.
- Para la impresión del resultado. La función `Prompt` tiene dos argumentos: la letra y el resultado (número entero) que se debe imprimir. La función debe escribir la letra, el carácter "`=`", el valor del resultado y el carácter de final de línea LF (*line feed*, valor 10 del código ASCII).

Para mayor claridad, expresaremos estas dos funciones en pseudocódigo:

```
int Input(char $a0) {
        print_char($a0);
        print_char('=');
        $v0=read_int();
        return($v0); }
```

```
void Prompt(char $a0, int $a1) {
        print_char($a0);
        print_char('=');
        print_int($a1);
        print_char('\n');
        return; }
```

Nótese que los argumentos recibidos por las funciones están almacenados en registros. Por ejemplo, la función `Input` recibe el carácter a imprimir en el registro `$a0`; de manera similar, `Prompt` recibe los dos argumentos (un carácter y un entero) en los registres `$a0` y `$a1`. Este detalle es muy importante: esta manera de pasar los parámetros a las funciones se llama por valor. En la práctica siguiente modificaremos este ejemplo haciendo que las variables se ubiquen en la memoria principal y pasando como argumentos su dirección de memoria.

Cuando tenga hecha la codificación de las dos funciones `Input` y `Prompt`, deberá de reescribir completamente el cuerpo del programa principal para que rotule el multiplicando con la letra

"M", el multiplicador con la "Q" y el resultado del producto con "R". El diálogo resultante debe aparecer en la consola como en la figura siguiente:

```
A=Input('M');
B=Input('Q');
C=Mult(A,B);
Prompt('R',C);
Exit();
```

```
M=215
Q=875
R=188125
```

Figura 3. A la izquierda el pseudocódigo del programa principal que debe escribir y a la derecha un ejemplo de diálogo resultante. En negrita, aparece el texto escrito por el programa. En cursiva, el texto tecleado por el usuario.

Ejercicio 3: Instrucciones condicionales

Nótese que la función **`Mult`** sólo funciona correctamente si el multiplicador Q es positivo. Pruebe a ejecutar el programa con *Q=–5*: el bucle de la función se alargará y deberá detener el programa. Para ello puede pulsar la combinación de teclas CTRL+C o bien pulsar el icono del menú rotulado con la palabra *Stop*.

En este apartado se le pide modificar ligeramente el programa principal para que si *Q<0*, en lugar de calcular *R=Mult(M,Q)* calcule *R=Mult(–M,–Q)*, es decir cambie el signo de ambos argumentos antes de llamar a la función a fin de mantener el resultado correcto. Si expresamos esta acción en pseudocódigo para una mayor claridad tenemos el siguiente:

```
M=Input('M');
Q=Input('Q');
If (Q<0)
      M=–M;
      Q=–Q;
R=Mult(M,Q);
Prompt('R',R);
Exit();
```

Figura 4. Una manera de resolver la limitación de **`Mult`** y poder operar con multiplicadores negativos

El punto fundamental aquí es descubrir como cambiar el signo de un número entero.

Cuestiones diversas

Se trata de cuestiones de lápiz y papel, pero en algunos casos puede comprobarlas con el simulador. Puede resolverlas en el laboratorio, si le sobra tiempo, o resolverlas en casa.

Instrucciones y pseudoinstrucciones

1. Si se precisara una pseudoinstrucción **`ca2 rt,rs`** que hiciera la operación *rt = complemento_a_2(rs)*, ¿cómo se traduciría? ¿Hay alguna pseudoinstrucción estándar del MIPS equivalente a **`ca2`**?

2. Con ayuda del simulador, pruebe a cargar código donde aparezca la pseudoinstrucción **`li $1,20`** o **`li $at,20`**. ¿Qué dice el simulador?

3. ¿Cómo se traducirá una hipotética pseudoinstrucción como **`beqi $t0,4,eti`** (saltar a **`eti`** si *$t0=4*)

4. ¿Cómo se traducirá la pseudoinstrucción **`b eti`**, (*branch,* salto incondicional a **`eti`**) en instrucciones de bifurcación condicional del formato I, sin usar la instrucción **`j`** (*jump*)?

5. ¿Puede explicar la diferencia entre las llamadas print_char(100) y print_integer(100)?

6. Y ¿Cuál es la diferencia entre print_char('A') y print_integer('A')?

7. En la Tabla 3 tiene la traducción de dos de las seis pseudoinstrucciones de Tabla 2. ¿Cuál es la traducción de las cuatro que faltan?

Ejercicios adicionales con el simulador

Puede hacerlos en el laboratorio, si le sobra tiempo, o acabarlos en casa.

Ejercicio 4: Iteraciones

1. Haga los cambios necesarios en el programa principal para que se repita el cálculo M×Q hasta que alguno de los dos operandos introducidos por el teclado valga cero, es decir, se trata de repetir la multiplicación mientras los dos operandos sean diferentes de cero. Eso mismo expresado en pseudocódigo:

```
repeat
  M=Input('M');
  Q=Input('Q');
  R=Mult(M,Q);
  Prompt('R',R);
while ((M≠0) && (Q≠0));
Exit();
```

2. Diseñe un programa que pida un número *n* y escriba la tabla de multiplicar de *n*, desde *n*×1 hasta *n*×10. Para hacer la programación más sencilla puede utilizar una función **`PromptM`** el pseudocódigo de la cual se expresa a continuación:

```
void PromptM(int x, int y, int r) {
  print_int(x);
  print_char('x');
  print_int(y);
  print_char('=');
  print_int(r);
  print_char('\n');
}
```

Ejercicio 5: Selector

Escriba la función **`void PrintChar(char c)`**, que imprime en la consola un carácter siguiendo el estilo de C: entre comillas y mostrando los casos especiales **`'\n'`** (carácter ASCII número 10) y **`'\0'`** (carácter ASCII número 0).

```
void PrintChar(int x) {
        putchar('"');  /* comilla */
        switch (x){
                case 0:  print_char('\'); print_char('0'); break;
                case 10:    print_char('\'); print_char('n'); break;
                default: print_char(x);
        }
        putchar('"'); /* comilla */
}
```

Anexo

Ejemplos de control de flujo

En la tabla siguiente,

- Los símbolos *cond*, *cond1*, etc., hacen referencia a las seis condiciones simples (= y ≠, > y ≤, < y ≥) que relacionan dos valores contenidos en registros. El asterisco indica condición contraria; por ejemplo, si *cond* = "**>**" tenemos *cond** = "≤".
- En la columna de alto nivel, los símbolos *A*, *B*, etc. indican sentencias simples o compuestas; en la columna de bajo nivel, los símbolos **A**, **B**, etc. representan los bloques de instrucciones equivalentes en ensamblador.

Condicionales.

<table>
<tr><th>Alto nivel</th><th>Ensamblador</th></tr>
<tr><td>if (cond1)
  A;
else if (cond2)
  B;
else
  C;
D;</td><td>
<pre>
if: bif (cond1*) elseif
        A
        j endif
elseif: bif (cond2*) else
        B
        j endif
else: C
endif: D
</pre>
<pre>
if: bif (cond1) then
        bif (cond2) elseif
        j else
then: A
        j endif
elseif: B
        j endif
else: C
endif: D
</pre>
</td></tr>
<tr><td>if (cond1 && cond2)
  A;
B;</td><td>
<pre>
if: bif (cond1*) endif
        bif (cond2*) endif
        A
endif: B
</pre>
</td></tr>
</table>

if (*cond1*||*cond2*) A; B;	`if: bif (cond1) then` `bif (cond2*) endif` `then: A` `endif: B` `if: bif (cond1*) endif` `bif (cond2*) endif` `A` `endif: B`

Selectores

Alto nivel	Ensamblador
switch (exp){ case X : A; break; case Y : case Z : B; break; default: C; } D;	`bif (exp != X) caseY` `caseX: A` `j endSwitch` `caseY: bif (exp != Y) default` `caseZ: bif (exp != Z) default` `B` `j endSwitch` `default: C` `endSwitch: D` `bif (exp == X) caseX` `bif (exp == Y) caseY` `bif (exp == Z) caseZ` `j default` `caseX: A` `j endSwitch` `caseY:` `caseZ: B` `j endSwitch` `default: C` `endSwitch: D`

Iteraciones

Alt nivel	Ensamblador
while (*cond*) A; B;	`while: bif (cond*) endwhile` `A` `j while` `endwhile B`
do A; while (*cond*) B;	`do: A` `bif (cond) do` `B`

do A; if(*cond1*) continue; B; if(*cond2*) break; C; while (*cond3*) D;	``` do: A bif (cond1) while B bif (cond2) enddo C while: bif (cond3) do enddo: D ```
iterar *n* veces /* *n*>0 */ A; B;	``` li $r,n bucle: A addi $r,$r,-1 bgtz $r,bucle B ```

Llamadas al sistema del PCSpim

$v0	Nom bre	Descripción	Argumentos	Resultado	Equivalente Java	Equivalente C
1	*print_integer*	Imprime (*) el valor de un entero	**$a0** = entero a imprimir	—	`System.out.print(int $a0)`	`printf("%d",$a0)`
2	*print_float*	Imprime (*) el valor de un *float*	**$f12** = float a imprimir	—	`System.out.print(float $f0)`	`printf("%f",$f0)`
3	*print_double*	Imprime (*) el valor de un *double*	**$f12** = double a imprimir	—	`System.out.print(double $f0)`	`printf("%Lf",$f0)`
4	*print_string*	Imprime una cadena de caracteres acabada en nul ('\0')	**$a0** = puntero a la cadena	—	`System.out.print(int $a0)`	`printf("%s",$a0)`
5	*read_integer*	Lee (*) el valor de un entero	—	**$v0** = enter leido		
6	*read_float*	Lee (*) el valor de un *float*	—	**$f0** = *float* leido		
7	*read_double*	Lee (*) el valor de un *double*	—	**$f0** = *double* leido		
8	*read_string*	Lee una cadena de caracteres (de longitud limitada) hasta encontrar un '\n' y la deja en el buffer acaba en nul ('\0')	**$a0** = puntero al buffer de entrada **$a1** = nombre máximo de caracteres de la cadena			
9	*sbrk*	Reservar un bloque de memoria del *heap*	**$a0** = longitud del bloque en bytes	**$v0** = dirección base del bloque de memoria		`malloc(integer n);`
10	*exit*		—	—		`exit(0);`
11	*print_character*		**$a0** = carácter a imprimir			`putc(char c);`
12	*read_character*			**$a0** = carácter leido		`getc();`

NOTAS

El asterisco en Imprime* y Lee* indica que, además de la operación de entrada/salida, hay un cambio de representación de binario a alfanumérico o de alfanumérico a binar