ESTRUCTURA DE COMPUTADORS
Grau en Enginyeria Informàtica

Sessió de laboratori número 3

FUNCIONS I CRIDES AL SISTEMA

Objectius

- Entendre els tres materials amb què es fa un programa en codi màquina (instruccions, dades i funcions de sistema).
- Construir funcions simples i cridar-les des d'un programa.
- Fer inventari d'instruccions per al control de flux.
- Conèixer i fer ús de les crides al sistema per mitjà de la instrucció màquina `syscall`.

Bibliografia

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

Introducció teòrica

Enters i caràcters

Ja sabem que en un computador les tires o paraules de bits no tenen un significat propi. Això significa que, per exemple, la paraula de bits `0x90324a00` es pot interpretar, segons el context, com una instrucció màquina, una adreça de memòria, un nombre enter sense signe, un nombre enter amb signe codificat en complement a dos (en aquest cast seria negatiu), un nombre real codificat segons la norma IEEE 754 (en aquest cas també seria negatiu), una cadena de 4 bytes de longitud, etcètera.

Encara que els computadors van ser dissenyats originalment per a fer gran quantitat de càlculs aritmètics, prompte van ser emprats a processar text. Gran nombre de computadors actuals utilitzen paraules de 8 bits per a representar caràcters segons el codi ASCII (*American Standard Code for Information Exchange*). Aquest codi té alguns trets que sempre he de tenir en compte: els codis per a lletres minúscules i majúscules només difereixen en un bit i, numèricament, la posició d'aquest bit fa que la diferència quantitativa entre els dos codis de cada lletra (majúscula i minúscula) siga de 32. Per exemple, el codi per a la lletra "Q" és 81 i per a la lletra "q" és 113 (noteu que 81+32=113, i que els dos codis en binari són,

respectivament, 1010001 i 1110001). Un altre valor important és el zero, anomenat *null*, que s'empra en C i Java per a marcar el final d'una cadena de caràcters.

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	⌂

Codi ASCII amb la representació dels primers 128 caràcters.

Dit això, cal afegir que, tanmateix, en l'actualitat *Unicode* és la codificació universal dels alfabets dels idiomes humans (llatí, grec, ciríl·lic, bengalí, etíop, tailandès, i un llarg etcètera). Hi ha tants alfabets en Unicode com símbols útils hi ha en ASCII. El llenguatge de programació Java empra Unicode per a codificar caràcters. Per omissió, utilitza 16 bits per a representar un caràcter. Per saber més d'Unicode podeu consultar www.unicode.org.

Ja sabem que el repertori d'instruccions del MIPS d'accés a memòria inclou la possibilitat d'accedir a tires de 32 bits (*word*) amb **`lw`** i **`sw`**, de 16 bits (*half*) amb **`lh`** i **`sh`** i de 8 bits (*byte*) amb **`lb`** i **`sb`**. Cal notar que les instruccions de lectura **`lh`** i **`lb`**, com que llegeixen menys de 32 bits, completen els bits que falten estenent el signe de la paraula llegida; val a dir, interpreten de manera implícita el valor llegit com un enter codificat en complement a dos.

Ara bé, ¿què ocorre si el que llegim de la memòria és un caràcter codificat en ASCII o Unicode? Doncs que no té cap sentit parlar de bit de signe perquè el valor llegit no s'ha d'interpretar com a un enter sinó com un caràcter. És per això que el repertori d'instruccions del MIPS R2000 també inclou les variants sense signe (*unsigned*) **`lhu`** i **`lbu`** per tal de llegir tires de bits més petites de 32 bits on no s'ha de fer cap extensió de signe. Aquestes

instruccions fan que els bits que falten per omplir el registre destinació es posen a zero. De fet, les instruccions **`lhu`** i **`lbu`** són més populars que **`lh`** i **`lb`**.

Control de flux d'execució en assemblador

Les instruccions de salt junt amb certes instruccions aritmètiques permeten construir les estructures condicionals i iteratives.

A baix nivell, podem distingir entre:

- salts incondicionals del tipus *seguir en l'adreça*; per exemple, la instrucció **`j eti`**.
- salts condicionals o bifurcacions *si (condició) seguir en l'adreça* on *adreça* assenyala la instrucció que s'executaria a continuació. En el joc del MIPS, tenim sis condicions per a salts condicionals: noteu que es poden fer tres parelles de condicions contràries (**=** i $\neq$, **>** i $\leq$, **<** i $\geq$).

El joc d'instruccions només permet les comparacions **=** i $\neq$ entre dos registres i les comparacions **>**, $\leq$, **<** i $\geq$ entre un registre i el zero:

`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`**
$rs \neq rt$	$rs \leq 0$	$rs \geq 0$

Taula 1. Instruccions de bifurcació del MIPS

Aquest assortiment de condicions es pot ampliar amb ajuda de la instrucció aritmètica **`slt`** (*set on less than*) i les instruccions relacionades que estudiareu en el tema d'aritmètica d'enters. Així s'hi obtenen aquestes altres sis pseudoinstruccions:

`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`**
$rs \neq 0$	$rs \leq rt$	$rs \geq rt$

Taula 2. Pseudoinstruccions de bifurcació del MIPS

Vegeu la traducció d'un parell de pseudoinstruccions de salt en instruccions màquina en la taula següent:

Pseudoinstrucció	Instruccions màquina
`beqz rs,A`	**`beq rs,$zero,A`**
`bgt rs,rt,A`	**`slt $at,rt,rs`** **`bne $at,$zero,A`**

Taula 3. Traducció de les pseudoinstruccions **`beqz`** i **`bgt`** en instruccions del MIPS

Amb aquestes instruccions podeu construir estructures condicionals i iteratives equivalents a les que escriviu en alt nivell. Per exemple, si hi ha un bloc d'instruccions A1, A2… que només s'ha d'executar si el contingut d'un registre **$r** és negatiu, podeu triar una bifurcació que salte si es dóna la condició contrària (**$r ≥ 0**):

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

Per a iterar *n* vegades un bloc d'instruccions A1, A2…, trieu un registre **$r** i escriviu:

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

Vegeu en l'annex un quadre amb la traducció de diverses estructures de control de flux.

Funcions del programa

Les funcions del programa (*callee functions*) són la traducció dels mètodes de Java o les funcions de C. El parell d'instruccions **jal *eti*** (o crida a la funció) i **jr $ra** (el retorn de la funció), lligades al registre **$ra** (**$31**), donen el suport bàsic al flux d'execució.

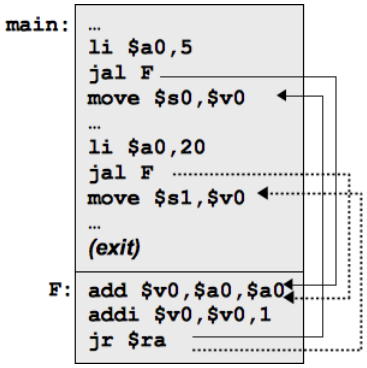

Figura 1. A l'esquerra, teniu l'esquema d'un programa **main()** que crida des de dos punts a una funció **F** que en alt nivell s'expressaria com **int F(int a){return 2*a+1}**. En tots dos casos, la instrucció **jal F** guarda en el registre **$ra** l'adreça de retorn, i per això la funció **F** acaba amb una instrucció **jr $ra**. Les fletxes de la figura en mostren el flux d'execució: la primera crida en continu ⟶ i la segona a traços ⋯▸
.
El programa principal, per la seua banda, acaba amb la crida al sistema **exit**.

El conveni d'ús del registres també preveu la separació entre registres del programa i registres de la funció. Els registres **$s0** a **$s7** estan orientats a servir de variables globals del programa, i els registres **$t0** a **$t9** a variables locals del procediment. El conveni diu:

- Si el programa principal utilitza un registre **$ti**, ha de preveure que qualsevol funció que cride podrà canviar-ne el contingut.

- Si una funció necessita escriure en un registre **$si**, haurà de preservar-ne el contingut previ i restaurar-lo abans d'acabar.
- Si una funció fa servir un registre **$ti**, haurà de tindre en compte que, entre dues execucions de la mateixa funció, qualsevol altra funció podrà modificar-ne el contingut.

El conveni també preveu la comunicació entre el programa principal i la funció, i la regula en funció del nombre i tipus de les dades intercanviades. Per exemple, si els arguments són de tipus enter i no hi ha més de quatre, aniran per ordre en els registres **$a0** a **$a3**. El valor retornat per la funció, si és un enter, s'escriurà en el registre **$v0**.

En resum: en els exercicis d'aquestes pràctiques, us convé seguir les regles de la taula següent a l'hora de programar. Aquestes regles les ampliarem més endavant per a permetre que les funcions del programa puguen cridar-se entre si.

Registres	Ús
$s0…$s7	El codi del programa principal
$a0…$a3	Pas de paràmetres del programa a les funcions
$t0…$t9	El codi de la funció
$v0	Retorn de resultats de les funcions als programes

Taula 4. Regles del conveni d'ús dels registres per part de les aplicacions

Les crides al sistema

Les operacions d'escriptura en la consola o de lectura del teclat suposen accedir a parts del computador que, per raons de seguretat i d'eficiència, no estan visibles per als programes corrents. La majoria dels computadors, reals o simulats, disposen de perifèrics i d'un sistema operatiu (per rudimentari que siga aquest) que ofereix un catàleg de funcions. Més endavant, en el temes referents a l'entrada/eixida, n'estudiareu els detalls.

En un MIPS, aquestes funcions de sistema es poden emprar mitjançant la instrucció **syscall**. Cada funció es distingeix per un número que la identifica anomenat índex, pot acceptar una sèrie d'arguments i desa un possible resultat.

Il·lustrem amb un exemple el mecanisme de crida. El codi següent llegeix un nombre enter des del teclat i el copia en l'adreça de memòria etiquetada amb el nom **valor**:

```
li $v0, 5          # Índex de la crida read_int
syscall            # Crida al sistema read_int
sw $v0, valor      # Copia el nombre enter en memòria
```

El catàleg de funcions del sistema simulat en PCSpim el teniu a la taula següent:

Servei	Codi de la crida	Arguments	Resultat
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		

Tanmateix, en aquesta pràctica heu de treballar només amb les quatre funcions referides en la taula següent. Noteu que l'índex que identifica el tipus de servei sempre s'indica en el registre **$v0**, algunes crides atenen els paràmetre contingut en el registre **$a0** i, si tornen resultat, ho fan sempre en **$v0**:

Nom	$v0	Descripció	Arguments	Resultat
print_int	*1*	Imprimeix el valor d'un enter	**$a0** = enter per a imprimir	—
read_int	*5*	Llig el valor d'un enter	—	**$v0** = enter llegit
exit	*10*	Acaba el procés	—	—
print_char	*11*	Imprimeix un caràcter	**$a0** = caràcter per a imprimir	—

Taula 5. Funcions del sistema que cal utilitzar en aquesta pràctica.

Per tant, l'ús de les funcions del sistema és ben paregut al de les funcions del programa; la diferència més notable és que el codi de les funcions del sistema està amagat, és independent dels programes, és comú per a tots ells i preserven el contingut dels registres globals i locals. En definitiva, no és necessari conéixer l'adreça on es troben les funcions del sistema per a poder fer-ne ús.

Exercicis de laboratori

Exercici 1: Les crides al sistema i les funcions dels programes

Obriu i observeu el codi següent contingut al fitxer "02_exer_01.s". Noteu que només hi ha el segment de codi (**.text**) i no trobem cap comentari. Els comentaris els haureu d'afegir segons aneu entenent què fa 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 lloc, heu de detectar quines instruccions pertanyen al programa principal i quines a una funció de nom **`Mult`**.

- Quines són les dues últimes instruccions del programa principal?
- Quina és l'última instrucció de la funció?
- Busqueu-hi les quatre crides al sistema emprades en el programa. Què fa cadascuna?
- Busqueu un bucle dins de la funció. Quantes vegades s'executa aquest bucle?
- Què fa la funció exactament?

Carregueu el programa i executeu-lo. Notareu que l'entrada/eixida per la consola és molt pobre.

- Sabeu executar el programa complet? En executar-lo, tingueu en compte que el programa demana l'entrada de dos nombres pel teclat i després imprimeix un resultat. Ara bé, no hi haurà cap missatge que us indique que s'està esperant una entrada del teclat.
- Sabeu fer una execució pas a pas?

Tècnica experimental: ús dels *breakpoints*. És molt útil per a detenir el programa en un punt on convé inspeccionar els registres o la memòria sense haver-hi d'anar pas a pas des del principi. Hi ha prou a indicar-li al simulador l'adreça de la instrucció on ha d'aturar-se l'execució. Feu servir la tècnica anterior per a detenir l'execució dins de **`Mult`** i observar el valor de l'adreça de retorn contingut en el registre **`$ra`**. Haureu d'indicar com a punt de trencament del flux d'execució l'adreça de la instrucció **`jr $ra`**.

- Quin és el valor de l'adreça de retorn?
- A quina instrucció del programa apunta?

Exercici 2: Creació de funcions

Anem a millorar el diàleg del programa anterior a través de la consola. Aquesta millora consisteix a associar el símbol d'una lletra a cada valor que llegiu o escriviu. Així, podeu batejar el multiplicand com 'M', el multiplicador com 'Q' i el producte com 'R'. Heu d'escriure dues funcions que haureu d'afegir al programa de l'apartat anterior:

- Per a la introducció de valors pel teclat. La funció **`Input`** té com a argument el símbol de la lletra que anem a escriure en la consola. La funció ha d'escriure en la consola aquest símbol seguit per la lletra '**=**' i després llegir un enter (el multiplicador o el multiplicand). La funció ha de tornar aquest valor llegit.
- Per a la impressió del resultat. La funció **`Prompt`** té dos arguments: la lletra i el resultat (nombre enter) que cal imprimir. La funció ha d'escriure la lletra, el símbol "**=**", el valor del resultat i el caràcter de final de línia LF (*line feed*, valor 10 del codi ASCII).

Per a major claredat, expressarem aquestes dues funcions en pseudocodi:

```
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; }
```

Noteu que els arguments rebuts per les funcions estan emmagatzemats en registres. Per exemple, la funció **`Input`** rep el caràcter a imprimir en el registre **`$a0`**; de manera similar, **`Prompt`** rep els dos arguments (un caràcter i un enter) en els registres **`$a0`** i **`$a1`**. Aquest detall és molt important: aquesta manera de passar els paràmetres a les funcions s'anomena per valor. En la pràctica següent modificarem aquest exemple fent que les variables s'ubiquen en la memòria principal i passant com a arguments la seua adreça de memòria.

Quan tingueu feta la codificació de les dues funcions **`Input`** i **`Prompt`**, haureu de reescriure completament el cos del programa principal perquè retole el multiplicand amb la lletra "M", el multiplicador amb la "Q" i el resultat del producte amb "R". El diàleg resultant ha d'aparèixer en la consola com en la figura següent:

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

M=*215*
Q=*875*
R=188125

Figura 2. A l'esquerra teniu l'esquema en pseudocodi del programa principal que heu d'escriure i a la dreta un exemple de diàleg resultant. En negreta, apareix el text escrit pel programa. En cursiva, el text teclejat per l'usuari.

Exercici 3: Instruccions condicionals

Noteu que la funció **`Mult`** només funciona correctament si el multiplicador Q és positiu. Proveu a executar el programa amb *Q=–5*: el bucle de la funció s'allargarà i caldrà detenir el programa. Per tal d'aturar l'execució podeu polsar combinació de tecles ^C o bé polsar la icona del menú retolada amb la paraula *Stop*.

En aquest apartat us demanem modificar lleugerament el programa principal per tal que si *Q<0*, en comptes de calcular *R=Mult(M,Q)* calcule *R=Mult(–M,–Q)*, això és, canvie el signe d'ambdós arguments abans de cridar la funció per tal de mantenir el resultar correcte. Si expressem aquesta acció en pseudocodi per a major claredat tenim el següent:

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

Figura 3. Una manera de resoldre la limitació de **`Mult`** i poder operar amb multiplicadors negatius

El punt fonamental ací és esbrinar com canviar el signe d'un nombre enter.

Qüestions diverses

Es tracta de qüestions de llapis i paper, però en alguns casos podeu comprovar-les amb el simulador. Podeu resoldre-les en el laboratori, si us sobra temps, o resoldre-les a casa.

Instruccions i pseudoinstruccions

1. Si calguera una pseudoinstrucció **`ca2 rt,rs`** que fera l'operació *rt = complement_a_2(rs)*, com es traduiria? Hi ha alguna pseudoinstrucció estàndard del MIPS equivalent a **`ca2`**?

2. Amb l'ajuda del simulador, proveu a carregar codi on aparega la pseudoinstrucció **`li $1,20`** o **`li $at,20`**. Què en diu el simulador?

3. Com es traduirà una hipotètica pseudoinstrucció com **`beqi $t0,4,eti`** (saltar a **`eti`** si *$t0=4*)

4. Com es traduirà la pseudoinstrucció **`b eti`**, (*branch,* salt incondicional a **`eti`**) en instruccions de bifurcació condicional del format I, sense usar la instrucció **`j`** (*jump*)?

5. Podeu explicar la diferència entre les crides print_char(100) i print_integer(100)?

6. I quina és la diferència entre print_char('A') i print_integer('A')?

7. En la Taula 3 teniu la traducció de dos de les sis pseudoinstruccions de Taula 2. Quina és la traducció de les quatre que hi falten?

Exercicis addicionals amb el simulador

Podeu fer-los en el laboratori, si us sobra temps, o acabar-los a casa.

Exercici 4: Iteracions

1. Feu els canvis necessaris en el programa principal que tal de repetir el càlcul M×Q fins que algun dels dos operands introduïts pel teclat valga zero, això és, es tracta de repetir la multiplicació mentre els dos operands siguen diferents de zero. Això mateix expressat en pseudocodi:

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

2. Dissenyeu un programa que demane per un nombre *n* i escriga la taula de multiplicar de *n*, des de *n*×1 fins *n*×10. Per tal de fer la programació més senzilla podeu emprar una funció `PromptM` el pseudocodi de la qual s'expressa tot seguit:

```
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');
}
```

Exercici 5: Selector

Escriviu la funció `void PrintChar(char c)`, que imprimeix en la consola un caràcter seguint l'estil de C: entre cometes i mostrant els casos especials `'\n'` (caràcter ASCII número 10) i `'\0'` (caràcter ASCII número 0).

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

Annex

Exemples de control de flux

En la taula següent,

- Els símbols *cond*, *cond1*, etc., fan referència a les sis condicions simples (**=** i ≠, **>** i ≤, **<** i ≥) que relacionen dos valors continguts en registres. L'asterisc indica condició contrària; per exemple, si *cond* = "**>**" tenim *cond** = "≤".
- En la columna d'alt nivell, els símbols *A*, *B*, etc. indiquen sentències simples o compostes; en la columna de baix nivell, els símbols **A**, **B**, etc. representen els blocs d'instruccions equivalents en assemblador.

Condicionals.

Alt nivell	Assemblador

if (*cond1*)
A;
else if (*cond2*)
B;
else
C;
D;
	`if:` ***bif (cond1*)*** `elseif`
	`A`
	`j endif`
	`elseif:` ***bif (cond2*)*** `else`
	`B`
	`j endif`
	`else:` `C`
	`endif:` `D`
	
	`if:` ***bif (cond1)*** `then`
	bif (cond2) `elseif`
	`j else`
	`then:` `A`
	`j endif`
	`elseif:` `B`
	`j endif`
	`else:` `C`
	`endif:` `D`

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

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**

Selectors

Alt nivell	Assemblador

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`

Iteracions

Alt nivell	Assemblador

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* vegades /* *n*>0 */
A;
B;
	``
	li $r,n
	bucle: A
	addi $r,$r,-1
	bgtz $r,bucle
	B
	``

Crides al sistema del PCSpim

$v0	Nom	Descripció	Arguments	Resultat	Equivalent Java	Equivalent C
1	*print_integer*	Imprimeix (*) el valor d'un enter	**$a0** = enter per a imprimir	—	`System.out.print(int $a0)`	`printf("%d",$a0)`
2	*print_float*	Imprimeix (*) el valor d'un *float*	**$f12** = float per a imprimir	—	`System.out.print(float $f0)`	`printf("%f",$f0)`
3	*print_double*	Imprimeix (*) el valor d'un *double*	**$f12** = double per a imprimir	—	`System.out.print(double $f0)`	`printf("%Lf",$f0)`
4	*print_string*	Imprimeix una cadena de caràcters acabada en nul ('\0')	**$a0** = punter a la cadena	—	`System.out.print(int $a0)`	`printf("%s",$a0)`
5	*read_integer*	Llig (*) el valor d'un enter	—	**$v0** = enter llegit		
6	*read_float*	Llig (*) el valor d'un *float*	—	**$f0** = *float* llegit		
7	*read_double*	Llig (*) el valor d'un *double*	—	**$f0** = *double* llegit		
8	*read_string*	Llig una cadena de caràcters (de llargària limitada) fins trobar un '\n' i la desa en el buffer acaba en nul ('\0')	**$a0** = punter al buffer d'entrada **$a1** = nombre màxim de caràcters de la cadena			
9	*sbrk*	Reservar un bloc de memòria del *heap*	**$a0** = longitud del bloc en bytes	**$v0** = adreça base del bloc de memòria		`malloc(integer n);`
10	*exit*		—	—		`exit(0);`
11	*print_character*		**$a0** = caràcter per a imprimir			`putc(char c);`
12	*read_character*			**$a0** = caràcter llegit		`getc();`

NOTES

L'asterisc en Imprimeix* i Llig* indiquen que, a més a més de l'operació d'entrada/eixida, hi ha un canvi de representació de binari a alfanumèric o d'alfanumèric a binari.