INSTRUCCIONS MIPS I MACROS MARS

INSTRUCCIONS MIPS	Enters	Coma flotant

Aritmètico-lògiques	addiu rt, rs, imm16
	addu rd, rs, rt
	and rd, rs, rt
	andi rt, rs, imm16
	div rs, rt
	divu rs, rt
	mult rs, rt
	multu rs, rt
	nor rd, rs, rt
	or rd, rs, rt
	ori rt, rs, imm16
	sll rd, rt, imm5
	sllv rd, rt, rs
	sra rd, rt, imm5
	srav rd, rt, rs
	srl rd, rt, imm5
	srlv rd, rt, rs
	subu rd, rs, rt
	xor rd, rs, rt
	xori rt, rs, imm16
		add.d fd, fs, ft
		add.s fd, fs, ft
		div.d fd, fs, ft
		div.s fd, fs, ft
		mul.d fd, fs, ft
		mul.s fd, fs, ft
		sub.d fd, fs, ft
		sub.s fd, fs, ft

Comparació	slt rd, rs, rt sltu rd, rs, rt slti rt, rs, imm16 sltiu rt, rs, imm16	c.eq.d fs, ft c.eq.s fs, ft c.le.d fs, ft c.le.s fs, ft c.lt.d fs, ft c.lt.s fs, ft

Memòria	lb rt, off16(rs)
	lbu rt, off16(rs)
	lh rt, off16(rs)
	lhu rt, off16(rs)
	lw rt, off16(rs)
	sb rt, off16(rs)
	sh rt, off16(rs)
	sw rt, off16(rs)
		lwc1 ft, off16(rs)
		swc1 ft, off16(rs)

Moviment de dades	lui rt, imm16 mfhi rd mflo rd mfc0 rt, rd mtc0 rd, rt	mfc1 rt, fs mov.s fd, fs mtc1 rd, fs
Instruccions de salt relatiu al PC	beq rs, rt, etiq bne rs, rt, etiq	bc1f etiq bc1t etiq
Instruccions de salt absolut	j etiq jal etiq jr rs	
Tractament d'excepcions	eret syscall	

MACROS MARS
b etiq
bge rsrc1, rsrc2, etiq
bgeu rsrc1, rsrc2, etiq
bgt rsrc1, rsrc2, etiq
bgtu rsrc1, rsrc2, etiq
ble rsrc1, rsrc2, etiq
bleu rsrc1, rsrc2, etiq
blt rsrc1, rsrc2, etiq
bltu rsrc1, rsrc2, etiq
la rdest, adr
li rdest, imm
move rdest, rsrc

INSTRUCCIONS MIPS

A continuació es detallen les instruccions MIPS ordenades alfabèticament. Abans es descriuen els elements que s'utilitzen en la seva sintaxi.

`rt`, `rs`, `rd`: Especifiquen un registre de propòsit general. En el format de la instrucció es representen en 5 bits, que es denoten com `ttttt`, `sssss` i `ddddd`, respectivament.

`imm16`: Especifica una expressió que avalua a una constant entera de 16 bits en Ca2. En el format de la instrucció aquest camp es denota amb 16 `i`'s.

`imm5`: Especifica una expressió que avalua a una constant natural de 5 bits. En el format de la instrucció aquest camp es denota amb 5 `i`'s.

`ft`, `fs`, `fd`: Especifiquen un registre de coma flotant. En el format de la instrucció es representen en 5 bits, que es denoten com `ttttt`, `sssss` i `ddddd`, respectivament.

`off16`: Especifica una expressió que avalua a una constant entera de 16 bits en Ca2. En el format de la instrucció aquest camp es denota amb 16 `o`'s.

`etiq`: Especifica una etiqueta del programa. Quan s'utilitza en una instrucció de salt relatiu al PC, en el format de la instrucció no s'especifica l'adreça absoluta d'`etiq`, sinó que es codifica en 16 bits el desplaçament que hi ha en nombre d'instruccions des de la següent instrucció fins a `etiq`. Aquest camp es denota amb 16 `o`'s. D'altra banda, quan s'utilitza en una instrucció de salt absolut, en el format de la instrucció s'especifiquen 26 bits, que corresponen als bits <27:2> de l'adreça d'`etiq`. Els 4 bits de més pes de l'adreça on hi ha la instrucció de salt i la del destí del salt han de coincidir. Els dos bits de menys pes del destí del salt són 0's i no s'especifiquen. En el format de la instrucció aquest camp es denota amb 26 `t`'s.

addiu	**Add Immediate**
`addiu:  0010 01ss ssst tttt iiii iiii iiii iiii`	
`addiu rt, rs, imm16`	rt = rs + SignExt(imm16)
La instrucció `addiu` realitza la suma aritmètica entera d'un registre i de l'extensió de signe d'una constant de 16 bits representada en Ca2. Existeix una instrucció anàloga (`addi`) que realitza la mateixa tasca, però amb la diferència que genera una excepció si es produeix sobreeiximent en la suma d'enters. La instrucció `addi` no la utilitzarem mai.	

addu	**Add**
`addu:  0000 00ss ssst tttt dddd d000 0010 0001`	
`addu  rd, rs, rt`	rd = rs + rt
La instrucció `addu` realitza la suma aritmètica entera de dos registres. Existeix una instrucció anàloga (`add`) que realitza la mateixa tasca, però amb la diferència que genera una excepció si es produeix sobreeiximent en la suma d'enters. La instrucció `add` no la utilitzarem mai.	

add.d	**Floating-Point Addition Double**
`add.d:  0100 0110 001t tttt ssss sddd dd00 0000`	
`add.d fd, fs, ft`	fd = fs + ft
La instrucció `add.d` realitza la suma en coma flotant de doble precisió de dos registres de coma flotant.	

add.s	**Floating-Point Addition Single**
`add.s:  0100 0110 000t tttt ssss sddd dd00 0000`	
`add.s fd, fs, ft`	fd = fs + ft
La instrucció `add.s` realitza la suma en coma flotant de simple precisió de dos registres de coma flotant.	

and	**AND**
`and:  0000 00ss ssst tttt dddd d000 0010 0100`	
`and  rd, rs, rt`	rd = rs AND rt
La instrucció `and` realitza el producte lògic per cada un dels dos bits que ocupen la mateixa posició als registres fonts.	

andi	**AND Immediate**
`andi:  0011 00ss ssst tttt iiii iiii iiii iiii`	
`andi rt, rs, imm16`	rt = rs AND ZeroExt(imm16)
La instrucció `and` realitza el producte lògic per cada un dels dos bits que ocupen la mateixa posició al registre font i a l'extensió de zeros d'una constant de 16 bits representada en Ca2.	

bc1f	**Branch Coprocessor 1 False**
`bc1f:  0100 0101 0000 0000 oooo oooo oooo oooo`	
`bc1f etiq`	$pc = etiq si cc = 0
La instrucció `bc1f` salta a `etiq` si el codi de condició de coma flotant `cc` és 0.	

bc1t	**Branch Coprocessor 1 True**
`bc1t:  0100 0101 0000 0001 oooo oooo oooo oooo`	
`bc1t etiq`	$pc = etiq si cc = 1
La instrucció `bc1t` salta a `etiq` si el codi de condició de coma flotant `cc` és 1.	

beq	**Branch On Equal**
`beq:  0001 00ss ssst tttt oooo oooo oooo oooo`	
`beq  rs, rt, etiq`	$pc = etiq si rs = rt
La instrucció `beq` salta a `etiq` si `rs` és igual a `rt`.	

bne	**Branch On Not Equal**
`bne:` `0001 01ss ssst tttt oooo oooo oooo oooo`	
`bne rs, rt, etiq`	$pc = etiq si rs <> rt
La instrucció `bne` salta a `etiq` si `rs` és diferent de `rt`.	

c.eq.d	**Compare Equal Double**
`c.eq.d:` `0100 0110 001t tttt ssss s000 0011 0010`	
`c.eq.d fs, ft`	cc=1 si fs = ft; altrament cc=0
La instrucció `c.eq.d` activa el codi de condició `cc` si els registres de coma flotant de doble precisió `fs` i `ft` són iguals; altrament el codi de condició es posa a 0.	

c.eq.s	**Compare Equal Single**
`c.eq.s:` `0100 0110 000t tttt ssss s000 0011 0010`	
`c.eq.s fs, ft`	cc=1 si fs = ft; altrament cc=0
La instrucció `c.eq.s` activa el codi de condició `cc` si els registres de coma flotant de simple precisió precisió `fs` i `ft` són iguals; altrament el codi de condició es posa a 0.	

c.le.d	**Compare Less Than Equal Double**
`c.le.d:` `0100 0110 001t tttt ssss s000 0011 1110`	
`c.le.d fs, ft`	cc=1 si fs <= ft; altrament cc=0
La instrucció `c.le.d` activa el codi de condició `cc` si el registre de coma flotant `fs` és més petit o igual (en doble precisió) que `ft`; altrament el codi de condició es posa a 0.	

c.le.s	**Compare Less Than Equal Single**
`c.le.s:` `0100 0110 000t tttt ssss s000 0011 1110`	
`c.le.s fs, ft`	cc=1 si fs <= ft; altrament cc=0
La instrucció `c.le.s` activa el codi de condició `cc` si el registre de coma flotant `fs` és més petit o igual (en simple precisió) que `ft`; altrament el codi de condició es posa a 0.	

c.lt.d	**Compare Less Than Double**
`c.lt.d:` `0100 0110 001t tttt ssss s000 0011 1100`	
`c.lt.d fs, ft`	cc=1 si fs < ft; altrament cc=0
La instrucció `c.lt.d` activa el codi de condició `cc` si el registre de coma flotant `fs` és més petit (en doble precisió) que `ft`; altrament el codi de condició es posa a 0.	

c.lt.s	**Compare Less Than Single**
`c.lt.s:` `0100 0110 000t tttt ssss s000 0011 1100`	
`c.lt.s fs, ft`	cc=1 si fs < ft; altrament cc=0
La instrucció `c.lt.s` activa el codi de condició `cc` si el registre de coma flotant `fs` és més petit (en simple precisió) que `ft`; altrament el codi de condició es posa a 0.	

div	**Divide**
`div:` `0000 00ss ssst tttt 0000 0000 0001 1010`	
`div rs, rt`	**lo = rs / rt; hi = rs % rt**
La instrucció `div` calcula el quocient i el residu de la divisió entera entre dos registres. El quocient s'emmagatzema al registre lo i el residu al registre hi. El signe del residu coincideix amb el signe del dividend. No es genera excepció per sobreeiximent del quocient. En cas de divisió per 0 el resultat resta indefinit.	

divu	**Unsigned Divide**
`divu:` `0000 00ss ssst tttt 0000 0000 0001 1011`	
`divu rs, rt`	lo = rs / rt; hi = rs % rt
La instrucció `divu` calcula el quocient i el residu de la divisió natural entre dos registres. El quocient s'emmagatzema al registre `lo` i el residu al registre `hi`. En cas de divisió per 0 el resultat resta indefinit.	

div.d	**Floating-Point Divide Double**
`div.d:` `0100 0110 001t tttt ssss sddd dd00 0011`	
`div.d fd, fs, ft`	fd = fs / ft
La instrucció `div.d` calcula el quocient de la divisió de dos registres de coma flotant en doble precisió.	

div.s	**Floating-Point Divide Single**
`div.s:` `0100 0110 000t tttt ssss sddd dd00 0011`	
`div.s fd, fs, ft`	fd = fs / ft
La instrucció `div.s` calcula el quocient de la divisió de dos registres de coma flotant en simple precisió.	

eret	**Exception Return**
`eret:` `0100 0010 0000 0000 0000 0000 0001 1000`	
`eret`	llegiu descripció
La instrucció `eret` activa el bit `EXL` del `Status Register` del coprocessador 0 i retorna a la instrucció apuntada pel registre `EPC` del coprocessador 0.	

j	**Jump**
`j:` `0000 10tt tttt tttt tttt tttt tttt tttt`	
`j etiq`	$pc = etiq
La instrucció `j` salta incondicionalment a `etiq`.	

jal	**Jump And Link**
`jal:` `0000 11tt tttt tttt tttt tttt tttt tttt`	
`jal etiq`	$ra = $pc+4; $pc = etiq
La instrucció `jal` salva l'adreça de la següent instrucció a `$ra` i salta incondicionalment a `etiq`.	

jr	**Jump Register**
`jr:` `0000 00ss sss0 0000 0000 0000 0000 1000`	
`jr rs`	$pc = rs
La instrucció `jr` salta incondicionalment a l'adreça especificada pel registre `rs`.	

lb	**Load Byte**
`lb:` `1000 00ss ssst tttt oooo oooo oooo oooo`	
`lb rt, off16(rs)`	rt = SignExt(M_{byte}[rs+SignExt(off16)])
La instrucció `lb` llegeix un byte de memòria a l'adreça indicada per la suma del registre `rs` i l'extensió de signe d'`off16`. La dada es guarda al registre destí fent extensió de signe.	

lbu	**Load Unsigned Byte**
`lbu:` `1001 00ss ssst tttt oooo oooo oooo oooo`	
`lbu rt, off16(rs)`	rt = ZeroExt(M_{byte}[rs+SignExt(off16)])
La instrucció `lbu` llegeix un byte de memòria a l'adreça indicada per la suma del registre `rs` i l'extensió de signe d'`off16`. La dada es guarda al registre destí fent extensió de zeros.	

lh	**Load Half**
`lh:` `1000 01ss ssst tttt oooo oooo oooo oooo`	
`lh rt, off16(rs)`	rt = SignExt(M_{half}[rs+SignExt(off16)])
La instrucció `lh` llegeix un half (16 bits) de memòria a l'adreça indicada per la suma del registre `rs` i l'extensió de signe d'`off16`. La dada es guarda al registre destí fent extensió de signe.	

lhu	**Load Unsigned Half**
`lhu:` `1001 01ss ssst tttt oooo oooo oooo oooo`	
`lhu rt, off16(rs)`	rt = ZeroExt(M_{half}[rs+SignExt(off16)])
La instrucció `lhu` llegeix un half (16 bits) de memòria a l'adreça indicada per la suma del registre `rs` i l'extensió de signe d'`off16`. La dada es guarda al registre destí fent extensió de zeros.	

lui	**Load Upper Immediate**
`lui:` `0011 1100 000t tttt iiii iiii iiii iiii`	
`lui rt, imm16`	$rt_{31:16}$ = imm16 $rt_{15:0}$ = 0
La instrucció `lui` carrega la constant de 16 bits `imm16` a la meitat alta del registre destí. També posa a 0 la meitat baixa del registre destí.	

lw	**Load Word**
`lw:` `1000 11ss ssst tttt oooo oooo oooo oooo`	
`lw rt, off16(rs)`	rt = M_{word}[rs+SignExt(off16)]
La instrucció `lw` llegeix una paraula (32 bits) de memòria a l'adreça indicada per la suma del registre `rs` i l'extensió de signe d'`off16`.	

lwc1	**Load Word Coprocessor 1**
`lwc1:` `1100 01ss ssst tttt oooo oooo oooo oooo`	
`lwc1 ft, off16(rs)`	ft = M_{word}[rs+SignExt(off16)]
La instrucció `lwc1` llegeix una paraula (32 bits) de memòria a l'adreça indicada per la suma del registre `rs` i l'extensió de signe d'`off16`. La dada es guarda al registre de coma flotant `ft`.	

mfhi	**Move From Hi**
`mfhi:` `0000 0000 0000 0000 dddd d000 0001 0000`	
`mfhi rd`	rd = $hi
La instrucció `mfhi` copia el registre `$hi` al registre destí.	

mflo	**Move From Lo**
`mflo:` `0000 0000 0000 0000 dddd d000 0001 0010`	
`mflo rd`	rd = $lo
La instrucció `mflo` copia el registre `$lo` al registre destí.	

mfc0	**Move From Coprocessor 0**
`mfc0:` `0100 0000 000t tttt dddd d000 0000 0000`	
`mfc0 rt, rd`	rt = rd (c0)
La instrucció `mfc0` copia el registre rd, pertanyent al coprocessador 0, cap al registre de propòsit general rt.	

mfc1	**Move From Coprocessor 1**
`mfc1:` `0100 0100 000t tttt ssss s000 0000 0000`	
`mfc1 rt, fs`	rt = fs
La instrucció `mfc1` copia el registre fs, pertanyent al coprocessador 1, cap al registre de propòsit general rt.	

mov.s	**Move Floating-Point Single**
`mov.s:` `0100 0110 0000 0000 ssss sddd dd00 0110`	
`mov.s fd, fs`	fd = fs
La instrucció `mov.s` copia el registre de coma flotant en simple precisió `fs` a `fd`.	

mtc0	**Move To Coprocesor 0**
`mtc0:` `0100 0001 000t tttt dddd d000 0000 0000`	
`mtc0 rt, rd`	rd (c0) = rt
La instrucció `mtc0` copia el registre de propòsit general rt cap al registre rd, pertanyent al coprocessador 0.	

mtc1	**Move To Coprocessor 1**
`mtc1:` `0100 0101 000t tttt ssss s000 0000 0000`	
`mtc1 rt, fs`	fs = rt
La instrucció `mtc1` copia el registre de propòsit general rt cap al registre fs, pertanyent al coprocessador 1.	

mult	**Multiply**
`mult:` `0000 00ss ssst tttt 0000 0000 0001 1000`	
`mult rs, rt`	lo = low(rs*rt); hi = high(rs*rt)
La instrucció `mult` calcula el producte enter de dos registres. Els 32 bits de menys pes s'emmagatzemen al registre `lo` i els 32 bits de més pes al registre `hi`.	

multu	**Unsigned Multiply**
`multu:` `0000 00ss ssst tttt 0000 0000 0001 1001`	
`multu rs, rt`	lo = low(rs*rt); hi = high(rs*rt)
La instrucció `mult` calcula el producte natural de dos registres. Els 32 bits de menys pes s'emmagatzemen al registre `lo` i els 32 bits de més pes al registre `hi`.	

mul.d	**Floating-Point Multiply Double**
`mul.d:` `0100 0110 001t tttt ssss sddd dd00 0010`	
`mul.d fd, fs, ft`	fd = fs * ft
La instrucció `mul.d` calcula el producte en coma flotant de doble precisió de dos registres de coma flotant.	

mul.s	**Floating-Point Multiply Single**
`mul.s:` `0100 0110 000t tttt ssss sddd dd00 0010`	
`mul.s fd, fs, ft`	fd = fs * ft
La instrucció `mul.s` calcula el producte en coma flotant de simple precisió de dos registres de coma flotant.	

nor	**NOR**
`nor:` `0000 00ss ssst tttt dddd d000 0010 0111`	
`nor rd, rs, rt`	rd = rs NOR rt
La instrucció `nor` realitza la complementació de la suma lògica per cada un dels dos bits que ocupen la mateixa posició als registres fonts.	

or	**OR**
`or:  0000 00ss ssst tttt dddd d000 0010 0101`	
`or  rd, rs, rt`	rd = rs OR rt
La instrucció `or` realitza la suma lògica per cada un dels dos bits que ocupen la mateixa posició als registres fonts.	

ori	**OR Immediate**
`ori:  0011 01ss ssst tttt iiii iiii iiii iiii`	
`ori  rt, rs, imm16`	rt = rs OR ZeroExt(imm16)
La instrucció `ori` realitza la suma lògica per cada un dels dos bits que ocupen la mateixa posició al registre font i a l'extensió de zeros de la constant de 16 bits, que és un operand de la instrucció.	

sb	**Store Byte**
`sb:  1010 00ss ssst tttt oooo oooo oooo oooo`	
`sb  rt, off16(rs)`	M_{byte}[rs+SignExt(off16)] = rt
La instrucció `sb` escriu el byte de menys pes del registre `rt` a memòria a l'adreça indicada per la suma del registre `rs` i l'extensió de signe d'`off16`.	

sh	**Store Half**
`sh:  1010 01ss ssst tttt oooo oooo oooo oooo`	
`sh  rt, off16(rs)`	M_{half}[rs+SignExt(off16)] = rt
La instrucció `sh` escriu la meitat baixa (16 bits) del registre `rt` a memòria a l'adreça indicada per la suma del registre `rs` i l'extensió de signe d'`off16`.	

sll	**Shift Left Logical**
`sll:  0000 00ss ssst tttt dddd diii ii00 0000`	
`sll  rd, rt, imm5`	rd = rt << imm5
La instrucció `sll` realitza un desplaçament de bits a l'esquerra del registre rt. El nombre de bits a desplaçar s'indica amb la constant imm5.	

sllv	**Shift Left Logical Variable**
`sllv:  0000 00ss ssst tttt dddd d000 0000 0100`	
`sllv rd, rt, rs`	rd = rt << rs
La instrucció `sll` realitza un desplaçament de bits a l'esquerra del registre rt. El nombre de bits a desplaçar ve indicat pels 5 bits de menys pes del registre `rs`.	

slt	**Set Less Than**
`slt:  0000 00ss ssst tttt dddd d000 0010 1010`	
`slt  rd, rs, rt`	rd = 1 si rs < rt; rd = 0 si rs >= rt
La instrucció `slt` posa a 1 el registre destí si `rs` < `rt` (comparació d'enters); altrament, el registre destí es posa a 0.	

sltu	**Set Less Than Unsigned**
`sltu:` `0000 00ss ssst tttt dddd d000 0010 1011`	
`sltu rd, rs, rt`	rd = 1 si rs < rt; rd = 0 si rs >= rt
La instrucció `sltu` posa a 1 el registre destí si `rs` < `rt` (comparació de naturals); altrament, el registre destí es posa a 0.	

slti	**Set Less Than Immediate**
`slti:` `0010 10ss ssst tttt iiii iiii iiii iiii`	
`slti rt, rs, imm16`	rt = 1 si rs < SignExt(imm16); rt = 0 si rs >= SignExt(imm16)
La instrucció `slti` posa a 1 el registre destí si rs < SignExt(imm16) (comparació d'enters); altrament, el registre destí es posa a 0.	

sltiu	**Set Less Than Immediate Unsigned**
`sltiu:` `0010 10ss ssst tttt iiii iiii iiii iiii`	
`sltiu rt, rs, imm16`	rt = 1 si rs < SignExt(imm16); rt = 0 si rs >= SignExt(imm16)
La instrucció `sltiu` posa a 1 el registre destí si `rs` < SignExt(`imm16`) (comparació de naturals); altrament, el registre destí es posa a 0.	

sra	**Shift Right Arithmetic**
`sra:` `0000 00ss ssst tttt dddd diii ii00 0011`	
`sra rd, rt, imm5`	rd = rt >> imm5
La instrucció `sra` realitza un desplaçament aritmètic (amb extensió de signe) de bits a la dreta del registre `rt`. El nombre de bits a desplaçar s'indica amb la constant `imm5`.	

srav	**Shift Right Arithmetic Variable**
`srav:` `0000 00ss ssst tttt dddd d000 0000 0111`	
`srav rd, rt, rs`	rd = rt >> rs
La instrucció `srav` realitza un desplaçament aritmètic (amb extensió de signe) de bits a la dreta del registre `rt`. El nombre de bits a desplaçar ve indicat pels 5 bits de menys pes del registre `rs`.	

srl	**Shift Right Logical**
`srl:` `0000 00ss ssst tttt dddd diii ii00 0010`	
`srl rd, rt, imm5`	rd = rt >>> imm5
La instrucció `srl` realitza un desplaçament lògic (amb extensió de zeros) de bits a la dreta del registre `rt`. El nombre de bits a desplaçar s'indica amb la constant `imm5`.	

srlv	**Shift Right Logical Variable**
`srlv:` `0000 00ss ssst tttt dddd d000 0000 0110`	
`srlv rd, rt, rs`	rd = rt >>> rs
La instrucció `srlv` realitza un desplaçament lògic (amb extensió de zeros) de bits a la dreta del registre `rt`. El nombre de bits a desplaçar ve indicat pels 5 bits de menys pes del registre `rs`.	

subu	**Subtract**
`subu:` `0000 00ss ssst tttt dddd d000 0010 0011`	
`subu rd, rs, rt`	rd = rs - rt
La instrucció `subu` realitza la resta entera de dos registres. Existeix una instrucció anàloga (`sub`) que realitza la mateixa tasca, però amb la diferència que genera una excepció si es produeix sobreeiximent en la resta d'enters. La instrucció `sub` no la utilitzarem mai	

sub.d	**Floating-Point Subtract Double**
`sub.d:` `0100 0110 001t tttt ssss sddd dd00 0001`	
`sub.d fd, fs, ft`	fd = fs - ft
La instrucció `sub.d` calcula la resta en coma flotant de doble precisió de dos registres de coma flotant.	

sub.s	**Floating-Point Subtract Single**
`sub.s:` `0100 0110 000t tttt ssss sddd dd00 0001`	
`sub.s fd, fs, ft`	fd = fs - ft
La instrucció `sub.s` calcula la resta en coma flotant de simple precisió de dos registres de coma flotant.	

sw	**Store Word**
`sw:` `1010 11ss ssst tttt oooo oooo oooo oooo`	
`sw rt, off16(rs)`	M_{word}[rs+SignExt(off16)] = rt
La instrucció `sw` escriu el registre `rt` a memòria a l'adreça indicada per la suma del registre `rs` i l'extensió de signe d'`off16`.	

swc1	**Store Word Coprocessor 1**
`swc1:` `1100 01ss ssst tttt oooo oooo oooo oooo`	
`swc1 ft, off16(rs)`	M_{word}[rs+SignExt(off16)] = ft
La instrucció `swc1` escriu el registre de coma flotant `ft` a memòria a l'adreça indicada per la suma del registre `rs` i l'extensió de signe d'`off16`.	

syscall	**System Call**
`syscall:` `0000 0000 0000 0000 0000 0000 0000 1100`	
`syscall`	llegiu descripció
La instrucció `syscall` crida al sistema operatiu generant una excepció. El registre `$v0` ha de contenir el paràmetre de la crida al sistema operatiu.	

xor	**XOR**
`xor:` `0000 00ss ssst tttt dddd d000 0010 0110`	
`xor rd, rs, rt`	rd = rs XOR rt
La instrucció `xor` realitza la suma exclusiva per cada un dels dos bits que ocupen la mateixa posició als registres fonts.	

xori	**XOR Immediate**
`xori:  0011 01ss ssst tttt iiii iiii iiii iiii`	
`xori rt, rs, imm16`	rt = rs XOR ZeroExt(imm16)
La instrucció `xori` realitza la suma exclusiva per cada un dels dos bits que ocupen la mateixa posició al registre font i a l'extensió de zeros de la constant de 16 bits `imm16`.	

MACROS MARS

A continuació es descriuen les macros MARS ordenades alfabèticament. Abans es detallen els elements que s'utilitzen en la seva sintaxi.

`rsrc1`, `rsrc2`, rdest: Especifiquen un registre de propòsit general.

`imm`: Especifica una expressió que avalua a una constant entera.

`adr`: Especifica una expressió que avalua a una constant natural.

b		**Branch**
`b etiq`	$pc = etiq	`beq $zero, $zero, etiq`

bge		**Branch On Greater Or Equal**
`bge rsrc1, rsrc2, etiq`	$pc = etiq si rsrc1 >= rscr2	`slt $at, rsrc1, rsrc2` `beq $at, $zero, etiq`

bgeu		**Branch On Greater Or Equal Unsigned**
`bgeu rsrc1, rsrc2, etiq`	$pc = etiq si rsrc1 >= rscr2	`sltu $at, rsrc1, rsrc2` `beq $at, $zero, etiq`

bgt		**Branch On Greater**
`bgt rsrc1, rsrc2, etiq`	$pc = etiq si rsrc1 > rscr2	`slt $at, rsrc2, rsrc1` `bne $at, $zero, etiq`

bgtu		**Branch On Greater Unsigned**
`bgtu rsrc1, rsrc2, etiq`	$pc = etiq si rsrc1 > rscr2	`sltu $at, rsrc2, rsrc1` `bne $at, $zero, etiq`

ble		**Branch On Lesser Or Equal**
`ble rsrc1, rsrc2, etiq`	$pc = etiq si rsrc1 <= rscr2	`slt $at, rsrc2, rsrc1` `beq $at, $zero, etiq`

bleu		**Branch On Lesser or Equal Unsigned**
`bleu rsrc1, rsrc2, etiq`	$pc = etiq si rsrc1 <= rscr2	`sltu $at, rsrc2, rsrc1` `beq $at, $zero, etiq`

blt		**Blanch On Lesser**
`blt rsrc1, rsrc2, etiq`	$pc = etiq si rsrc1 < rscr2	`slt $at, rsrc1, rsrc2` `bne $at, $zero, etiq`

bltu		**Branch On Lesser Unsigned**
`bltu rsrc1, rsrc2, etiq`	$pc = etiq si rsrc1 < rscr2	`sltu $at, rsrc1, rsrc2` `bne $at, $zero, etiq`

la		**Load Address**
`la  rdest, adr`	rdest = adr	`lui $at, hi(adr)` `ori rdest, $at, lo(adr)`

li		**Load Immediate**

`li  rdest, imm`	rdest = imm	`si imm = imm16 ->`
		`addiu rdest, $szero, imm16`
		`---`
		`si imm = imm32 ->`
		`lui $at, hi(imm32)`
		`ori rdest, $at, lo(imm32)`

move		**Move**
`move  rdest, rsrc`	`rdest = rsrc`	`addu rdest, $zero, rsrc`

DENOMINACIó DELS 32 REGISTRES DE PROPòSIT GENERAL

MIPS té 32 registres de 32 bits cadascun per a propòsit general. En el llenguatge assemblador es poden fer constar de dues maneres: pel número o pel nom (però als programes de l'assignatura usarem exclusivament el nom):

Número	Nom	Utilització
`$0`	`$zero`	Sempre val zero, no modificable
`$1`	`$at`	Reservat a l'expansió de macros (convé no usar-lo)
`$2-$3`	`$v0-$v1`	Resultat de subrutines (sols usarem $v0)
`$4-$7`	`$a0-$a3`	*Arguments* de subrutines
`$8-$15`	`$t0-$t7`	*Temporals*, no preservats per les crides a subrutines
`$16-$23`	`$s0-$s7`	*Saved* ("segurs"), temporals preservats per les crides a subrutines
`$24-$25`	`$t8-$t9`	*Temporals*, no preservats per les crides a subrutines
`$26-$27`	`$k0-$k1`	Reservats per al nucli (*Kernel*) del SO (convé no usar-los)
`$28`	`$gp`	*Global pointer*, explicat al Tema 3 (no l'usarem)
`$29`	`$sp`	*Stack pointer*, conté l'adreça del cim de la pila
`$30`	`$fp`	*Frame pointer* (no l'estudiem)
`$31`	`$ra`	*Return address*, adreça de retorn de subrutina