Aritmètica en coma flotant

Índex

1 El joc d'instruccions de coma flotant

PROBLEMA 1 El codi següent escrit en llenguatge d'alt nivell correspon a una funció que calcula el perímetre d'una circumferència a partir del seu radi:

```
float perimetre (float radi)
{
    return (2.0 * 3.1415 * radi);
}
```

Es demana implementar aquesta funció en codi assemblador del MIPS R2000. Suposeu que els dos valors constants es troben ubicats en el segment de dades.

SOLUCIÓ Com que les dues constants de la funció (2.0 i 3.1415) estan ubicades en memòria, la directiva que permet codificar els valors en l'estàndard IEEE 754 de simple precisió és `.float`. D'altra banda, la variable `radi` es passarà a la subrutina per mitjà del registre `$f12` i el resultat calculat es deixarà al registre `$f0`. El codi en assemblador serà el següent:

```
            .data 0x10000000
            .float 2.0
            .float 3.1415

            .text 0x400000
            ...
perimetre:  la $t0, 0x10000000
            lwc1 $f4, 0($t0)        # lectura de 2.0
            lwc1 $f6, 4($t0)        # lectura de 3.1415
            mul.s $f0, $f4, $f6     # multiplica les constants
            mul.s $f0, $f0, $f12    # càlcul del perímetre
            jr $ra
```

■

PROBLEMA 2 Modifiqueu el codi assemblador del problema 1 perquè considere tots els valors involucrats codificats en doble precisió.

SOLUCIÓ En aquest cas haurem de tenir en compte que les variables seran de tipus `double` i, per tant, ocuparan 64 bits. Les instruccions de multiplicació també s'hauran de modificar per tal que operen amb variables d'aquest tipus. En el cas de les lectures de memòria, el MIPS R2000 només disposa d'instruccions per a llegir paraules de 32 bits, per la qual cosa la càrrega d'un valor expressat en doble precisió necessita dues instruccions `lwc1`.

```
            .data 0x10000000
            .double 2.0
            .double 3.1415926

            .text 0x400000
            ...
perimetre:  la $t0, 0x10000000
            lwc1 $f4, 0($t0)        # llegeix part baixa de 2.0
            lwc1 $f5, 4($t0)        # llegeix part alta de 2.0
            lwc1 $f6, 8($t0)        # llegeix part baixa de 3.1415
            lwc1 $f7, 12($t0)       # llegeix part alta de 3.1415
            mul.d $f0, $f4, $f6     # producte de les dues constants
            mul.d $f0, $f0, $f12    # càlcul del perímetre
            jr $ra
```

■

PROBLEMA 3 Determineu què fa el codi següent escrit en llenguatge d'assemblador del processador MIPS R2000:

```
    mfc1 $t0, $f0

    li $t1, 0x80000000
    and $t1, $t0, $t1
    li $t4, 31
    srlv $t1, $t1, $t4

    li $t2, 0x7F800000
    and $t2, $t0, $t2
    li $t4, 23
    srlv $t2, $t2, $t4

    li $t3, 0x007FFFFF
    and $t3, $t0, $t3
    li $t4, 0x00800000
    or $t3, $t3, $t4
```

SOLUCIÓ La primera instrucció del programa copia el contingut del registre de coma flotant `$f0` en el registre enter `$t0`. A continuació el codi va extraent, per mitjà de tres màscares de bits, els tres camps que conformen l'estàndard de codificació de nombres de coma flotant IEEE 754 de simple precisió. Així, en `$t1` deixa el bit de signe, en `$t2` els vuit bits de l'exponent i, finalment, en `$t3` els 23 bits de la mantissa. Després, mitjançant desplaçaments lògics cap a la dreta, alinea els bits obtinguts del signe i l'exponent en la part baixa del registre on s'han copiat. A la mantissa ja està en la part més baixa del registre, li afegeix el bit implícit.

■

PROBLEMA 4 Alguns assembladors del MIPS admeten les pseudoinstruccions `li.s` i `li.d` que permeten carregar una constant en un registre de coma flotant. La traducció d'aquestes pseudoinstruccions és un seguit d'instruccions que formen la representació de la constant en un registre enter i en fan la transferència al banc de coma flotant, com mostra l'exemple següent:

```
# Traducció de li.s $f0, 2.75
        lui $at,0x4030
        mtc1 $at, $f0
```

Escriviu una traducció possible de les pseudoinstruccions següents en seqüències d'instruccions del processador:

1. `li.s $f0, +0.0`

2. `li.d $f0, -0.0`

3. `li.d $f0, -2.75`

4. `li.s $f0, +Inf`

SOLUCIÓ

1. Traducció de `li.s $f0, +0.0`

```
mtc1 $zero, $f0
```

2. Traducció de `li.d $f0, -0.0`

```
lui $at, 0x8000
mtc1 $at, $f1
mtc1 $zero, $f0
```

3. Traducció de `li.d $f0, -2.75`

```
lui $at,0xC006
mtc1 $at, $f1
mtc1 $zero, $f0
```

4. Traducció de `li.s $f0, +Inf`

```
lui $at,0x7F80
mtc1 $at, $f0
```

■

PROBLEMA 5 Considereu el codi següent escrit en llenguatge d'alt nivell:

```
do {
    f = f/1707.0;
} while (f<0.0001);
```

Es demana implementar aquest codi en llenguatge d'assemblador del MIPS R2000. Suposeu que la variable `f` es troba continguda inicialmente al registre `$f18`.

SOLUCIÓ El codi en llenguatge d'assemblador que proposem per a implementar el codi anterior és el següent:

```
        li.s $f4, 1707.0        # $f4 conté el valor 1707.0
        li.s $f6, 0.0001        # $f6 conté el valor 0.0001
bucle:  div.s $f18, $f18,$f4    # divisió i assignació de f
        c.lt.s $f18, $f6        # comparació del while
        bc1t bucle              # tornar al bucle si f<0.0001
```

Les dues primeres pseudoinstruccions carreguen els valors constants 1707.0 i 0.0001 en dos registres temporals del coprocessador matemàtic. Cal fer notar que la traducció d'aquestes dues pseudoinstruccions a instruccions màquina és duta pel programa assemblador; aquesta traducció implica, també, la traducció de les constants del programa a les cadenes de bits corresponents codificades segons les normes de l'estàndard IEEE 754.

A continuació el bucle es limita a fer la divisió i l'assignació de la variable `f` i avaluar la condició. Aquesta condició de "menor que" es comprova amb la instrucció `c.lt.s` (les lletres `lt` indiquen *lower than*). Si la condició s'acompleix, l'indicador de condició `FPc` del registre d'estat del processador (bit número 23) es posa a 1 i, en cas contrari, a 0. L'última instrucció, `bc1t`, saltarà a la primera instrucció del bucle quan la comprovació pose l'indicador `FPc` a 1.

■

PROBLEMA 6 El conveni de programació en llenguatge d'assemblador del MIPS R2000 proposa l'ús dels registres de coma flotant següent:

Nom	Ús
`$f0`	Retorn de funció (part real)
`$f2`	Retorn de funció (part imaginària)
`$f4..$f10`	Registres temporals
`$f12,$f14`	Pas de paràmetres
`$f16,$f18`	Registres temporals
`$f20..$f30`	Registres a preservar

on cada registre d'aquests 16 es pot tractar com un registre de 64 bits on s'emmagatzema un nombre en format IEEE 754 de doble precisió (DP) o com un registre de 32 bits per a nombres en simple precisió (SP). Vegeu que el conveni només referencia els registres amb índex parells (`$f0`, `$f2`, ..., `$f30`); per tant els registres senars (`$f1`, `$f3`, ..., `$f31`) només s'utilitzen per a contenir la part alta de valors en doble precisió.

Noteu que la distribució dels registres preveu el càlcul de variable complexa, i per això les funcions poden tornar en `$f2` la part imaginària del resultat. Les funcions de variable real només tornaran els seus valors en `$f0`. En conseqüència, les funcions prendran com a paràmetres els valors continguts en `$f12` i `$f14` i desaran el resultat computat en els registres `$f0` (part real) i `$f2` (part imaginària, si escau). Finalment, considerem sis registres temporals: `$f4`, `$f6`, `$f8`, `$f10`, `$f16` i `$f18`.

Seguint aquest conveni, construïu una biblioteca de funcions de coma flotant que implementen els càlculs següents:

1. `float F2C_s(float a)`. Fa la conversió d'una temperatura t_F, expresada en graus Fahrenheit, a t_C, expressada en graus Celsius, segons la fòrmula:

$$t_C = \frac{5.0}{9.0} \times (t_F - 32.0)$$

2. `float sqr_s(float a)`. Calcula el quadrat de dos valors reals en simple precisió.

3. `double add_sd(float a, double b)`. Calcula la suma de dos valors reals de tipus distint: el paràmetre `a` (ubicat en `$f12`) és de simple precisió i el paràmetre b (en el registre `$f14`) és de doble precisió. El resultat de la funció és un valor real de doble precisió.

4. `addcx_s`. Calcula la suma de dos valors complexos de simple precisió. Com que és un cas complicat, no feu molt de cas del conveni: considereu que el primer sumand és `$f12` (part real) i `$f13` (part imaginària) i que el segon sumand està en `$f14` i `$f15`.

5. `float max_s (float a, float b)`. Calcula el màxim de dos valors reals en simple precisió.

6. `double max_d (double a, double b)`. Calcula el màxim de dos valors reals expressats en doble precisió.

SOLUCIÓ Per a cada funció de les esmentades més amunt, cal dissenyar el codi en assemblador corresponent al seu còmput. Noteu que, si no es diu res que ho contradiga, caldrà respectar el conveni d'utilització dels registres.

1. Funció `sqr_s`: el codi calcula primer el quocient $5.0/9.0$ emprant valors reals (no enters), després computa la resta que hi ha entre parèntesis i, finalment, fa la multiplicació:

```
F2C_s:  li.s $f4, 5.0          # Constant 5.0 en $f4
        li.s $f6, 9.0          # Constant 9.0 en $f6
        div.s $f4, $f4, $f6    # Càlcul de la fracció 5.0/9.0
        li.s $f6, 30.0         # Constant 32.0 en $f4
        sub.s $f0, $f12, $f6   # Càlcul tF - 32.0
        mul.s $f0, $f0, $f4    # Multiplicació final
        jr $ra
```

2. Funció `sqr_s`: en aquest cas el codi és molt senzill, ja que n'hi haurà prou amb multiplicar el paràmetre d'entrada (registre `$f12`) per si mateix i retornar el valor calculat en el registre `$f0`:

```
sqr_s:  mul.s $f0, $f12, $f12  # Elevem al quadrat
        jr $ra
```

3. Funció `add_sd`: com que el resultat ha de ser un valor en doble precisió, cal fer una conversió prèvia del paràmetre de simple precisió:

```
add_sd: cvt.d.s $f0, $f12       # Conversió de simple a doble
        add.d $f0, $f0, $f14    # Suma en doble precisió
        jr $ra
```

4. Funció `addcx_s`: en aquest cas haurem de sumar les parts reals entre si i les imaginàries entre sí, i deixar el resultat en `$f0` i `$f2`, respectivament:

```
addcx_s:  add.s $f0, $f12, $f14   # Càlcul part real
          add.s $f2, $f13, $f15   # Càlcul part imaginària
          jr $ra
```

5. Funció `max_s`: s'ha de fer una comparació entre els dos paràmetres de la funció; la instrucció que farem servir és `c.le.s` (estableix si el primer registre és menor que el segon). Si la condició s'acompleix aleshores l'indicador de condició `FPc` del registre de l'estat del coprocessador matemàtic es posa a 1 i, consegüentment, la instrucció `bc1t` provocarà un salt:

```
max_s:      c.le.s $f12, $f14  # Comparació menor que
            bc1t f14_guanya    # Si salta $f14 és major
f12_guanya: mov.s $f0, $f12    # Retorna el valor de $f12
            jr $ra
f14_guanya: mov.s $f0, $f14    # Retorna el valor de $f14
            jr $ra
```

6. Funció `max_d`: aquest cas és similar a l'anterior, amb l'exepció que els valors vénen expressats en doble precisió i, per tant, la instrucció de comparació ara serà `c.le.d`. A més, noteu que els valors manejats en el codi són de 64 bits (suffix `.d` en la instrucció de moviment de dades):

```
max_d:      c.le.d $f12, $f14  # Comparació menor que
            bc1t f14_guanya    # Si salta $f14 és major
f12_guanya: mov.d $f0, $f12    # Retorna el valor de $f12
            jr $ra
f14_guanya: mov.d $f0, $f14    # Retorna el valor de $f14
            jr $ra
```

■

PROBLEMA 7 Escriviu una subrutina `es_NaN` que, donat un nombre real en el registre `$f12`, torne en `$v0` un 1 si és un NaN o un 0 en cas contrari.

SOLUCIÓ La subrutina analitza l'exponent per tal d'esbrinar si val 255 (tot a uns). En el cas que tinga aquest valor i, a més a més, la mantissa siga diferent de zero, aleshores serà un NaN.

```
es_NaN: mfc1 $t0, $f12          # copia $f12 en $t0
        li $t1, 0x7F800000      # màscara per a l'exponent
        and $t2, $t0, $t1       # exponent en $t2
        bne $t1, $t2, noes      # si exp<>255 no és NaN
        li $t1, 0x007FFFFF      # màscara per a la mantissa
        and $t2, $t0, $t1       # mantissa en $t2
        beq $t2, $zero, noes    # si mantissa==0 no és NaN
sies:   li $v0, 1
        j fi
noes:   li $v0, 0
fi:     jr $ra
```

■

PROBLEMA 8 En la secció de dades següent es defineixen un nombre real i un enter. Es demana escriure el codi corresponent per a que sume aquests nombres i deixe el resultat de la suma en l'adreça definida amb l'etiqueta `resultat`.

```
            .data
enter:      .word 2
real:       .float 3.5
resultat:   .float 0.0
            .text
            ....
            .end
```

Per a implementar el programa heu d'utilitzar només registres temporals del banc de registres d'enters (`$t0`, ...,`$t9`) i del banc de registres de reals (`$f4`,...,`$f10`).

SOLUCIÓ Per a poder sumar els dos números, han de trobar-se els dos codificats amb l'estàndard IEEE754, per tant han d'estar en el banc de registres de reals.

```
.text
lwc1 $f6, enter    # lectura de la variable "enter"
cvt.s.w $f6,$f6    # conversió de la variable "enter" a coma flotant
lwc1 $f4, real     # lectura de la variable "real"
add.s $f4,$f4,$f6  # suma en coma flotant
swc1 $f4,resultat  # escriptura del resultat
.end
```

Noteu que el codi anterior ha llegit la variable entera de la memòria i l'ha desat en el banc de registres de coma flotant per al seu tractament. Una alternativa a `lwc1 $f6, enter`, on es passa el valor de la variable pel banc de registres d'enters, seria la següent:

```
lw $t0, enter      # llig la variable "enter"
mtc1 $t0, $f6      # copia el valor en un registre de coma flotant
```

■

PROBLEMA 9 Escriviu un segment de programa que calcule la suma dels enters ubicats en els registres `$t0` i `$t1` utilitzant instruccions de coma flotant i deixe el resultat en el registre `$v0`.

SOLUCIÓ Aquesta és una situació que es dóna algunes vegades quan programem en un llenguatge d'alt nivell: hi ha dos enters que volem sumar, però la suma es fa sobre els valors reals equivalents als enters. Finalment, el resultat d'aquesta suma es torna a convertir en un valor enter. Un exemple senzill en C que il·lustra aquesta situació (ús de l'anomenat *casting*) és el següent:

```
int a,b,c;
...
c = (int) ( (float) a + (float) b )
```

Des del punt de vista del microprocessador, aquesta operació implica un moviment dels valors enters (32 bits) continguts en els registres `$t0` i `$t1` als registres `$f4` i `$f6` (per exemple) del coprocessador matemàtic (els valors reals que emprarem són de simple precisió). Després d'aquest moviment de dades, s'ha de fer una conversió de tipus (d'enter a real), ja que la instrucció `mtc1` es limita a copiar, tal qual, el valor del registre font en el destí. Una vegada feta la suma, el resultat real s'ha de convertir a enter i després s'ha de copiar en el registre enter `$v0`. El codi en assemblador que fa totes aquestes operacions és el següent:

```
mtc1 $t0, $f4       # Copia $t0 en $f4
mtc1 $t1, $f6       # Copia $t1 en $f6
cvt.s.w $f4, $f4    # Conversió d'enter a real
cvt.s.w $f6, $f6    # Conversió d'enter a real
add.s $f4, $f4, $f6 # Suma de dos valors reals
cvt.w.s $f4, $f4    # Conversió de real a enter
mfc1 $v0, $f4       # Copia el resultat en $v0
```

■

PROBLEMA 10 Volem dissenyar una biblioteca de funcions de càlcul que combinen operands enters i de coma flotant. Utilitzeu els registres d'acord amb els criteris habituals de pas de paràmetres i retorn de valors. La llista de funcions que componen aquesta biblioteca és la següent:

1. `add_ws`. Calcula la suma d'un enter (*int*) i d'un real de simple precisió (*float*). El resultat ha de ser un valor real de simple precisió.
2. `mul_wsd`. Calcula el producte d'un enter (*int*), d'un real de simple precisió (*float*) i d'un real de doble precisió (*double*). El resultat ha de ser un valor real de doble precisió .

SOLUCIÓ Les funcions que volem dissenyar inclouran, necessàriament, instruccions de conversió de tipus, concretament d'enter a real. Aquesta conversió és imprescindible perquè les operacions de suma i multiplicació es fan sobre nombres expressats en coma flotant.

1. Funció `add_ws`. S'ha de fer una conversió del valor enter en un real de simple precisió abans de dur a terme la suma:

```
add_ws: mtc1 $a0, $f16        # Copia paràmetre real en $f16
        cvt.s.w $f0, $f16     # Conversió d'enter a  real
        add.s $f0, $f0, $f12  # Suma de valors reals
        jr $ra
```

2. Funció `mul_wsd`. En aquest cas s'han de fer dues conversions: la primera, de real en simple precisió a doble precisió; la segona, d'enter a real de doble precisió:

```
mul_wsd:    cvt.d.s $f0, $f12      # Conversió simple a doble
            mul.d $f0, $f0, $f14   # Multiplicació inicial
            mtc1 $a0, $f16         # Copia enter en $f16
            cvt.d.w $f16, $f16     # Conversió enter a doble
            mul.d $f0, $f0, $f16   # Multiplicació final
            jr $ra
```

■

PROBLEMA 11 Escriviu una subrutina `m_arit` que calcule la mitjana aritmètica dels elements d'un vector. En el registre `$a0` rep l'adreça del vector i en `$a1` la dimensió; el resultat calculat es torna en `$f0`. Us podeu basar en el codi d'alt nivell següent:

```
float m_arit (float *A[], int n)
{
    float aux;
    int i;

    aux = 0.0;
    for (i=0; i<n; i++)
    {
        aux = aux + A[i];
    }
    return (aux / (float) n);
}
```

Cal remarcar que l'última divisió de la funció fa una conversió prèvia de la variable `n` en valor real.

SOLUCIÓ La subrutina implementa un bucle de lectura dels elements del vector. Els elements són llegits amb la instrucció `lwc1`. Per a fer la divisió entre el nombre d'elements caldrà fer una conversió de tipus (enter a float).

```
m_arit: or $t0, $zero, $a0      # copia de $a0 en $t0
        or $t1, $zero, $a1      # copia de $a1 en $t1
        mtc1 $zero, $f0         # posa 0.0 en $f0

bucle:  lwc1 $f4, 0($t0)        # llig element A[i]
        add.s $f0, $f0, $f4     # incrementa $f0 amb A[i]
        addiu $t0, $t0, 4       # actualitza adreça a A[i+1]
        addi $t1, $t1, -1       # decrementa comptador
        bne $t1, $zero, bucle   # salta si queden elements

        mtc1 $a1, $f4           # copia $a1 (dimensió) en $f4
        cvt.s.w $f4, $f4        # converteix la dimensió en real
        div.s $f0, $f0, $f4     # divideix entre la dimensió
        jr $ra
```

■

PROBLEMA 12 Considereu el codi següent escrit en llenguatge d'alt nivell que calcula la mitjana harmònica dels valors continguts en un vector de reals de simple precisió:

```
float harmonica (int n, float *v[])
{
    float aux;
    int i;

    aux = 0.0;
    for (i=0; i<n; i++)
    {
        aux = aux + (1.0/v[i]);
    }
    return ( (float) n / aux);
}
```

Cal remarcar que l'última divisió de la funció fa una conversió prèvia de la variable `n` en valor real. Es demana implementar aquesta funció en codi assemblador del MIPS R2000.

SOLUCIÓ La variable `n` es passarà a la funció per mitjà del registre `$a0`, l'adreça de començament del vector en memòria per mitjà del registre `$a1`, i el resultat calculat es deixarà al registre `$f0`. El codi en assemblador serà el següent:

```
harmonica:  li.s $f4, 0.0           # aux en $f4 a zero
            li.s $f6, 1.0           # $f6 conté un 1.0
            mfc0 $t0, $a0           # i en $t0 igual a n

bucle:      lwc1 $f8, 0($a1)        # lectura de v[i] en $f8
            div.s $f8, $f6, $f8     # càlcul de 1.0/v[i]
            add.s $f4, $f4, $f8     # increment d'aux
            addi $a1, $a1, 4        # increment de l'adreça
            addi $t0, $t0, -1       # decrement de i
            bne $t0, $zero, bucle   # salt si diferent de zero
            mtc1 $a0, $f6           # mou n al coprocessador 1
            cvt.s.w $f6, $f6        # converteix n a float
            mul.s $f12, $f6, $f4    # producte final
            jr $ra
```

Com a detall a remarcar, la pseudoinstrucció `li.s` s'empra per tal de carregar valors constants en registres del coprocessador CP1. Per exemple, `li.s $f6, 1.0` carrega el valor 1.0 en el registre `$f6` del coprocessador matemàtic tot codificant-lo d'acord amb l'estàndard IEEE 754 de simple precisió.

■

PROBLEMA 13 Considereu els fragments de programa següents: el fragment *A* correspon a un programa en assemblador del MIPS R2000; els fragments *B* i *C* estan escrits en un llenguatge d'alt nivell en què el tipus és un enter de 32 bits amb signe.

Fragment *A*:

```
        .data
i:      .word ...
x:      .float ...
        ...
        .text
        mtc1 $zero,$f0
        lwc1 $f2,x
        c.ge.f $f0,$f2
        bc1f else
if:     li $t0,1
        sw $t0,i
        j fi
else:   li $t0,-1
        sw $t0,i
fi:
```

Fragment *B*:

```
int j;
float y;
...
if(j<=0)
    y = 1.0;
else
    y = -1.0;
```

Fragment *C*:

```
int k;
float z;
...
k = (long) z + 8;
```

1. Escriviu el programa d'alt nivell equivalent al fragment *A*.
2. Escriviu el programa en assemblador equivalent al fragment *B*.
3. Escriviu el programa en assemblador equivalent al fragment *C*.

SOLUCIÓ

1. Fragment A en alt nivell:

```
int i;
float x;
...
if(x>=0)
    i = 1;
else
    i = -1;
```

2. Fragment B en assemblador:

```
            .data
      j:    .word ...
      y:    .float ...
            ...
            .text
            lw $t0,j
            bgtz $t0,else
      if:   li $t0,1
            mtc1 $t0,$f0
            cvt.s.w $f0,$f0
            swc1 $f0,y
            j fi
      else: li $t0,-1
            mtc1 $t0,$f0
            cvt.s.w $f0,$f0
            swc1 $f0,y
      fi:
```

3. Fragment C en assemblador:

```
            .data
      k:    .word ...
      z:    .float ...
            ...
            .text
            lwc1 $f0,z
            cvt.w.s $f0,$f0
            mfc1 $t0,$f0
            addi $t0,$t0,8
            sw $t0,k
```

■

PROBLEMA 14 Codifiqueu en assemblador del MIPS R2000 la funció `calcul` que es mostra a continuació. La funció rep com a paràmetres quatre punters a memòria codificats en els registres `$a0`, `$a1`, `$a2`, `$a3`. La funció torna un valor de tipus `double` (valor en coma flotant de doble precisió) en el parell de registres `$f0`, `$f1`.

```
double calcul(int *a, int *b, float *c, float *d) {
  if (a<b)
    return (double)(c*d);
  else
    return (double)((float)b+c);
}
```

SOLUCIÓ El codi de la funció en assemblador es mostra a continuació.

```
calcul:  lw $t0, 0($a0)
         lw $t1, 0($a1)
         blt $t0, $t1, menor
         mtc1 $t1, $f2
         cvt.s.w $f2, $f2
         lwc1 $f0, 0($a2)
         add.s $f0, $f0, $f2
         cvt.d.s $f0, $f0
         jr $ra
menor:   lwc1 $f0, 0($a2)
         lwc1 $f2, 0($a3)
         mul.s $f0, $f2, $f0
         cvt.d.s $f0, $f0
         jr $ra
```

En primer lloc es carrega en els registres `$t0` i `$t1` els valors de les variables `a` i `b`, respectivament. Per això, fem servir la instrucció `lw` ja que els paràmetres passats a la funció són punters a les variables. A continuació comparem les dues variables (`a<b`) amb la pseudo-instrucció `blt` (una alternativa és utilitzar les instruccions `slt` i `bne`).

Per últim, en cadascuna de les dues possibles alternatives de la comparació es fan les operacions en coma flotant pertinents (multiplicació amb la instrucció `mul.s` o suma amb la instrucció `add.s`). Ara bé, en cadascun dels casos cal fer conversions amb la instrucció `cvt` i transferències entre registres d'enters y de coma flotant (instrucció `mtc1`) o transferències entre memòria y els registres de coma flotant (instrucció `lwc1`).

■

PROBLEMA 15 Escriviu una subrutina en assemblador del MIPS R2000 per a calcular el factorial d'un nombre enter $x!$, l'expressió matemàtica del qual és: $x! = x \cdot (x-1) \cdot (x-2) \cdot ... \cdot 2 \cdot 1$. El perfil d'aquesta subrutina serà:

```
double Factorial(int x)
```

El valor del nombre enter se li passa a la funció per mitjà de `$a0`, tornant el resultat com un nombre (`double`) en el registre `$f12`. Per a implementar el programa s'han d'utilitzar només registres temporals del banc de registres d'enters (`$t0`, ..., `$t9`) i del banc de registres de reals (`$f4`, ... ,`$f10`).

SOLUCIÓ

```
            .text
Factorial: mtc1    $a0,$f0        # Passem el número en $a0 a $f0
           cvt.d.w $f0,$f0        # convertim a double
           abs     $f0,$f0        # ho passem a positiu per si de cas
           li.d    $f12,1.0       # mult = 1.0
           li.d    $f4,-1.0
           li.d    $f2,1.0
bucle:     mul.d   $f12,$f12,$f0  # mult = mult * $f0
           add.d   $f0,$f0,$f4    # $f0 = $f0 - 1.0
           c.eq.d  $f0, $f2       # és $f0 <> 1.0?
           bclf    bucle          # salta a bucle
           jr      $ra            # resultat en $f12
```

■

PROBLEMA 16 Esteu creant una biblioteca de funcions aritmètiques de coma flotant per a un processador MIPS. En particular, heu d'implementar una función de transformació de coordenades. Siga $\bar{x} = (x_0, x_1, x_2)$ un vector de *3* coordenades (enters) i siga R una matriu de transformació de coordenades (double), de dimensió *3x3*. El resultat de la transformació és altre vector $\bar{y}$ que s'obté mitjançant el producte $\bar{y} = R \cdot \bar{x}$:

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \end{pmatrix} = \begin{bmatrix} r_{00} & r_{01} & r_{02} \\ r_{10} & r_{11} & r_{12} \\ r_{20} & r_{21} & r_{22} \end{bmatrix} \cdot \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} \quad (1)$$

La versió d'alt nivell d'aquesta funció és:

```
void transf_coord(int *x, double *R, double *y )
```

on:

- *x: adreça d'inici del vector x. Es passarà en $a0.
- *R: adreça d'inici de la matriz R. Es passarà en $a1.
- *y: adreça d'inici del vector y de resultat. Es passarà en $a2.

Tot seguit es mostra una versió del codi corresponent a la subrutina *transf_coord*. Tingau en compte que la matriu R està emmagatzemada en la memòria per fileres i que els seus components són valors en doble precisió, mentre que els components del vector x són enters. En aquesta solució s'obté cada component del vector resultat y mitjançant el producte escalar de la filera corresponent de R pel vector de entrada x.

```
transf_coord: li $t3,3             # índex i=3
bucle:        jal prod_escalar     # producte escalar $a0 = dir x $a1= dir fila R[i]
              swc1 $f12, 0($a2)    # resultat $f12/13
              swc1 $f13, 4($a2)
              addi $a1, $a1, 24
              addi $a2, $a2, 8
              addi $t3, $t3, -1    # i = i-1
              bne $t3, $0, bucle   # bucle 'i'
              jr $ra
```

Escriviu el codi corresponent a la subrutina *prod_escalar*.

SOLUCIÓ

```
prod_escalar:  li $t2,3               # Índex j=3
               mov $t0,$a0            # Copiem $a0 i $a1
               mov $t1,$a1
               mtc1 $f12, $0          # Inicializa suma ($f12/13) = 0
               mtc1 $f13, $0
bucle_j:       lwc1 $f0, 0($t0)       # $f0/1 = x[j]
               cvt.d.w $f0, $f0       # $f0/1 = (double)x[j]
               lwc1 $f2, 0($t1)
               lwc1 $f3, 4($t1)       # $f2/3 = R[i][j]
               mult.d $f0, $f0, $f2   # $f0/1 = R[i][j] * x[j]
               add.d  $f12, $f12, $f0 # suma = suma + R[i][j] * x[j]
               addi $t0,$t0,4
               addi $t1,$t1,8
               addi $t2, $t2, -1      # j = j-1
               bne $t2, $0, bucle_j   # bucle_j
               jr $ra
```

■

---

2 Emulació d'instruccions de coma flotant

PROBLEMA 17 Escriviu el codi en llenguatge d'assemblador del processador MIPS R2000 que emule, amb instruccions enteres, les instruccions de coma flotant següents:

1. `abs.s $f0, $f4`
2. `abs.d $f0, $f4`
3. `neg.s $f0, $f4`

SOLUCIÓ

1. `abs.s $f0, $f4`. Es tracta d'obtenir en `$f0` el valor absolut del valor de coma flotant de simple precisió contingut en `$f4`. L'efecte net consisteix a fer positiu el valor de `$f4`, això és, posar el bit de signe a zero. Com que hem de fer servir instruccions enteres, podem aplicar una màscara de 32 bits consistent en 31 bits a un (els de menor pes) i el bit més significatiu (el de major pes) a zero:

   ```
   mfc1 $t0, $f4         # Copia $f4 en $t0
   li $t1, 0x7FFFFFFF    # Màscara de bits en $t1
   and $t0, $t0, $t1     # Bit més significatiu a zero
   mtc1 $t0, $f0         # Copia $t0 en $f0
   ```

2. `abs.d $f0, $f4`. Aquest cas és similar a l'anterior, però amb la diferència que els valors són de 64 bits (doble precisió). El valor real original està contingut als registres `$f4` (part menys significativa) i `$f5` (part més significativa). Així doncs, el bit de signe és el més significatiu, no del registre `$f4`, sinó del registre `$f5`, que conté la part més alta de valor real (el bit de signe, onze bits de l'exponent, i una part de la mantissa). Per tant, podem escriure:

```
mfc1 $t0, $f5          # Copia $f5 en $t0
li $t1, 0x7FFFFFFF     # Màscara de bits en $t1
and $t0, $t0, $t1      # Bit més significatiu a zero
mtc1 $t0, $f1          # Bit de signe a zero
mov.s $f0, $f4         # Part baixa sense modificar
```

3. `neg.s $f0, $f4`. Aquesta instrucció canvia el signe del valor contingut en el registre `$f4` i torna el resultat en `$f0`. La inversió del bit de signe es pot dur a terme mitjançant una operació lògica OR exclusiva:

```
mfc1 $t0, $f4          # Copia $f4 en $t0
li $t1, 0x80000000     # Màscara de bits en $t1
xor $t0, $t0, $t1      # Inversió del bit de signe
mtc1 $t0, $f0          # Copia de $t0 en $f0
```

■

PROBLEMA 18 Escriviu en assemblador del MIPS R2000 les funcions d'un paràmetre, indicat en el registre `$f12`, que us demanem a continuació. El resultat calculat es deixa en el registre `$f0`. No cal tractar els possibles desbordaments dels exponents.

1. `mul2_s`: multiplicació per dos en simple precisió.
2. `mul2_d`: multiplicació per dos en doble precisió.
3. `div2_s`: divisió per dos en simple precisió.
4. `div4_s`: divisió per quatre en simple precisió.

SOLUCIÓ En les solucions que proposem tot seguit, el procediment a seguir és molt fàcil: es tracta de manipular adequadament els bits que representen l'exponent del valor real. Així, com que la base implícita de l'exponent en l'estàndard IEEE 754 és 2, multiplicar per dos equival a sumar un un a l'exponent, mentre que la divisió equival a restar-li un un. Vegem cada cas per separat.

1. `mul2_s`. En aquest cas hem de sumar un un a l'exponent. Les posicions que ocupen els 8 bits de l'exponent són les de major pes (sense considerar el bit de signe). En particular, si hem de sumar un un a l'exponent, la posició que ocupa el bit de menor pes de l'exponent serà la que fa 23 (comptant des de zero). El codi que demanen és el següent:

```
mul2_s: mfc1 $t0, $f12      # Copia $f12 en $t0
        li $t1,0x00800000   # Màscara: un un en la 23a posició
        add $t0, $t0, $t1   # Suma un un a l'exponent
        mtc1 $t0, $f0       # Copia $t0 en $f0
        jr $ra
```

2. `mul2_d`. Es tracta d'un cas similar a l'anterior, però ara s'ha de tenir en compte que l'exponent, situat en els 32 bits de major pes del valor real (registre `$f13`), té 11 bits, el bit menys significatiu del qual ocupa la posició 20na, tot comptant des de zero:

```
mul2_d: mfc1 $t0, $f13       # Copia part més significativa
        li $t1, 0x00100000   # Màscara: un un en la 20a posició
        add $t0, $t0, $t1    # Suma un un a l'exponent
        mtc1 $t0, $f1        # Copia $t0 en $f1 (modificat)
        mov.s $f0, $f12      # Copia $f12 en $f0 (no modificat)
        jr $ra
```

3. `div2_s`. A diferència dels casos anteriors, en aquest es tracta de fer una resta. El codi resultat és molt similar a l'utilitzat en la implementació de la funció `mul2_s`, ja que únicament canvia la instrucció `add` per la `sub`:

```
div2_s: mfc1 $t0, $f12      # Copia $f12 en $t0
        li $t1,0x00800000   # Màscara: un un en la 23a posició
        sub $t0, $t0, $t1   # Resta un un a l'exponent
        mtc1 $t0, $f0       # Copia $t0 en $f0
        jr $ra
```

4. div4_s. En aquest cas es tracta de restar dues unitats a l'exponent, això és, haurem de restar un un però ara en la posició 24a:

```
div4_s: mfc1 $t0, $f12      # Copia de $f12 en $t0
        li $t1,0x01000000   # Màscara: un un en la 24a posició
        sub $t0, $t0, $t1   # Resta un dos a l'exponent
        mtc1 $t0, $f0       # Copia $t0 en $f0
        jr $ra
```

■

PROBLEMA 19 Escriviu en assemblador del MIPS R2000 una subrutina mul2_s que, donat un nombre real en el registre $f12, emule una multiplicació per 2, i deixe el resultat en el registre $f0. La subrutina ha de tenir en compte els casos especials (zero, infinit i NaN).

SOLUCIÓ La subrutina següent tracta els casos especials mitjançant l'anàlisi del valor de l'exponent (casos en què val 0 i 255).

```
mul2_s: mfc1 $t0, $f12          # copia $f12 en $t0
        li $t1, 0x7FFFFFFF      # màscara per a E i M
        and $t1, $t0, $t1       # extrau E i M en $t1
        beq $t1, $zero, eixir   # si E=M=0 es +0.0 o bé -0.0

        li $t1, 0x7F800000      # màscara per a E
        and $t2, $t0, $t1       # extrau E en $t2
        beq $t1, $t2, eixir     # si exp==255 és NaN, +Infty o -Infty

        li $t1, 0x00800000      # màscara: un un en la 23a posició
        add $t0, $t0, $t1       # suma un un a l'exponent
        li $t1, 0x7F800000      # màscara per a E
        and $t2, $t0, $t1       # extrau E en $t2
        bne $t2, $t1, eixir     # si exp<>255 és un nombre "normal"
                                # si exp==255 ha desbordat i cal generar un infinit
        li $t1, 0xFF800000      # màscara amb M a zero
        and $t0, $t0, $t1       # posa M del resultat a zero

eixir:  mtc1 $t0, $f0
        jr $ra
```

3 Mesura de prestacions

PROBLEMA 20 El segment de programa següent tarda exactament 7 ms a executar-se en un computador basat en el MIPS R2000:

```
        li $t0, 0
        li $t1, 25000
        add.s $f16, $f15, $f15
        mtc1 $zero, $f0
gris:   lwc1 $f4, 0($t0)
        mul.s $f8, $f6, $f6
        sub.s $f0, $f8, $f6
```

```
addi $t0, $t0, 4
bne $t0, $t1, gris
div.s $f0, $f0, $f6
abs.s $f0, $f0
```

Calculeu en MFLOPS la productivitat assolida pel programa.

SOLUCIÓ Com que coneixem el temps d'execució del codi, només cal calcular el nombre N_{CF} d'operacions en coma flotant que s'hi fan. En particular, fora del bucle podem comptabilitzar-ne tres: la segona instrucció del programa (`add.s`) i les dues últimes instruccions (`div.s` i `abs.s`). El bucle, que s'executa un total de 25000 vegades (valor inicial del registre temporal `$t1`), conté dues instruccions de coma flotant (`mul.s` i `sub.s`).

El nombre d'operacions en coma flotant és

$$N_{CF} = 3 \text{ operacions fora del bucle} + 25000 \text{ iteracions} \times 2 \text{ operacions/iteració} = 50003 \text{ operacions totals}$$

Per tant, la productivitat assolida pel programa serà:

$$\text{Productivitat} = \frac{N_{CF}}{\text{temps}} = \frac{50003 \text{ operacions}}{7 \cdot 10^{-3} \text{ segons}} = 7.14 \cdot 10^6 \text{ operacions/segon} = 7.14 \text{ MFLOPS}$$

■

PROBLEMA 21 Disposem d'un operador combinacional que pot realitzar dues operacions de coma flotant: suma i multiplicació. Les operacions se seleccionen mitjançant una entrada de control i tenen retards distints. La productivitat màxima d'aquest operador depén de l'operació seleccionada; en concret, per a la *suma* és d'1 MFLOPS i per a la *multiplicació* de 2 MFLOPS.

1. Quin és el temps necessari per a fer una suma? I una multiplicació?
2. Quin és el temps mínim necessari per a calcular el producte escalar de dos vectors de 2000 elements amb aquest operador? Suposeu que el producte escalar de vectors de n elements requereix fer n productes i n sumes.

SOLUCIÓ En aquest problema haurem de fer la conversió de MFLOPS a temps d'operació, cosa similar a la conversió entre freqüència i període.

1. Vegem primer el temps característic per a cada operació a partir de les productivitats màximes expressades en MFLOPS. Com que 1 MFLOP $= 10^6$ FLOPS (operacions de coma flotant per segon), el temps T_{suma} necessari per a fer una suma és:

   $$T_{\text{suma}} = \frac{1}{1 \cdot 10^6 \text{ FLOPS}} = 1 \cdot 10^{-6} \text{ segons} = 1 \ \mu\text{s}$$

   Per la seua banda, aplicant el mateix raonament, el temps T_{mult} necessari per a fer una multiplicació és:

   $$T_{\text{mult}} = \frac{1}{2 \cdot 10^6 \text{ FLOPS}} = 0.5 \ \mu\text{s}$$

   Noteu que el circuit aritmètic tarda més a fer una suma que una multiplicació (justament el doble), fet que també es desprén directament si comparem els MFLOPS assolits pel circuit en operacions de suma i en operacions de multiplicació.

2. El temps T_{op} d'operació total a fer el producte escalar serà el necessari per a fer les 2000 sumes i les 2000 multiplicacions:

   $$T_{\text{op}} = 2000 \times 1 \ \mu s + 2000 \times 0.5 \ \mu s = 3 \text{ ms}$$

■

PROBLEMA 22 Disposem d'un processador MIPS R2000 amb coprocessador matemàtic que funciona amb una freqüència de rellotge de 50 MHz. Aquest processador calcula el producte escalar de dos vectors `X` i `Y` de 10000 elements de reals de simple precisió mitjançant el bucle següent:

```
        li $t0, 0            # Desplaçament inicial
        li $t1, 40000        # Desplaçament final
        mtc1 $zero, $f0      # carreguem zero
bucle:  lwc1 $f4, X($t0)     # Lectura de X[i]
        lwc1 $f6,Y($t0)      # Lectura de Y[i]
        mul.s $f8, $f4, $f6  # Càlcul X[i]*Y[i]
        add.s $f0, $f0, $f8  # Actualització suma
        addi $t0, $t0, 4     # Increment desplaçament
        bne $t0, $t1, bucle  # Salta si falten elements
```

Considereu que el cost temporal d'execució (en cicles de rellotge) de les instruccions en el processador és el següent:

Tipus d'instrucció	Cicles
Aritmètica entera	1
Bifurcació	1
Lectura de memòria	2
Suma en coma flotant	4
Producte en coma flotant	7

1. Determineu el temps d'execució d'aquest bucle.
2. Calculeu la productivitat del processador, expressada en MFLOPS, executant operacions de coma flotant.
3. Suposeu ara que no hi ha coprocessador disponible, i les instruccions de coma flotant s'emulen mitjançant instruccions d'enters. Una instrucció `mul.s` s'emula amb 60 instruccions d'enters, i una instrucció `add.s` s'emula amb 30 instruccions. Quin seria el temps d'execució d'aquest bucle i la productivitat en MFLOPS resultant?

SOLUCIÓ Abans que res, calcularem el temps de cicle del processador. Com que la freqüència de funcionament és de 50 MHz, aleshores el temps de cicle es pot obtenir calculant la inversa: $1/(50\text{ MHz}) = 1/(50 \cdot 10^6)\text{ s} = 20\text{ ns}$.

1. Per a saber quant de temps tarda a executar-se el bucle cal calcular el temps d'una iteració. En particular, examinant-ne el codi, cada iteració demana $2+2+7+4+1+1 = 17$ cicles de rellotge. Així doncs, el temps d'execució serà:

$$10000\text{ iteracions} \times 17\text{ cicles/iteració} \times 20 \cdot 10^{-9}\text{ s/cicle} = 3.4\text{ ms}$$

2. Per a calcular la productivitat del bucle expressada en MFLOPS hem de tenir en compte que, de les 17 instruccions que s'executen en cada iteració del bucle, només dues són instruccions de coma flotant (`mul.s` i `add.s`). Això és, per cada 3.4 mil·lisegons que tarda cada iteració, s'executen dues instruccions de coma flotant. Per tant, la productivitat aconseguida en les 10000 iteracions del bucle es pot expressar com:

$$\frac{2 \times 10000}{3.4 \cdot 10^{-3} \times 10^6} = 5.88\text{ MFLOPS}$$

3. L'absència de coprocessador matemàtic fa que les operacions de coma flotant es duguen a terme executant instruccions d'aritmètica entera. En el cas que considerem, hi ha dues instruccions dins el bucle que seran traduïdes (emulades) per un conjunt concret d'instruccions enteres, cadascuna de les quals tarda, segons la taula adjuntada, un cicle a executar-se; en particular, l'execució d'una operació `mul.s` emulada demana 60

cicles, perquè serà substituïda per 60 instruccions d'un cicle de durada, mentre que l'execució de la instrucció `add.s` demanarà 30 cicles. En conseqüència, cada iteració demanarà un total de $2+2+60+30+1+1 = 96$ cicles. El càlcul del temps d'execució es farà de la manera següent:

$$10000 \times 96 \times 20 \cdot 10^{-9} = 19.2 \text{ ms}$$

Noteu que, amb l'emulació, el temps d'execució ha passat, de 3.4 a 19.2 mil·lisegons, increment de temps prou notable. Al seu torn, la productivitat resultant expressada en MFLOPS serà:

$$\frac{20000}{20 \cdot 10^{-3} \times 10^{6}} = 1.04 \text{ MFLOPS}$$

De forma pareguda, la productivitat del bucle ha caigut, dels 5.88 MFLOPS assolits amb el coprocessador matemàtic, als 1.04 MFLOPS obtinguts amb l'emulació.

■

PROBLEMA 23 Disposeu d'un processador MIPS que funciona a una freqüència de rellotge d'1 GHz. Aquest processador calcula la suma d'un valor enter amb un vector de 256 elements reals en precisió simple ($\vec{Z}$ = a + $\vec{Y}$) mitjançant el programa següent:

```
        li $t0, 0         # desplaçament inicial
        li $t1, 1024      # desplaçament final, 256 elements float ocupen 1024 bytes
        lwc1 $f0, a($t0)  # emmagatzema el valor "a" en $f0
        cvt.s.w $f0, $f0  # convert a formato real de simple precisió l'enter "a"
bucle:
        lwc1 $f1, Y($t0)        # lectura de Y[i].
        add.s $f2, $f0, $f1     # càlcul de "a + Y[i]"
        swc1 $f2, Z($t0)        # escriptura de Z[i]
        addi $t0, $t0, 4        # increment del desplaçament
        bne $t0, $t1, bucle     # salta si queden elements
```

Considerant que el cost temporal d'execució (en cicles de rellotge) de les instruccions en el processador és el següent:

Tipus d'instrucció	Cicles
Aritmètica entera	1
Bifurcació	1
Lectura de memòria	3
Escriptura en memòria	3
Suma en coma flotant	5

1. Determineu el temps d'execució en segons del BUCLE ($\vec{Z}$ = a + $\vec{Y}$) utilitzat en el programa.
2. Calculeu la productividad del bucle expresada en milions d'operacions de coma flotant per segon (MFLOP).

SOLUCIÓ

1. A cada iteració del bucle s'executen: 1 instrucció de lectura en memòria, 1 instrucció de suma en coma flotant, 1 instrucció d'escriptura en memòria, 1 instrucció entera i 1 instrucción de salt. Per tant, el temps d'execució de cada iteració d'aquest bucle serà de: 3 + 5 + 3 + 1 + 1 = 13 cicles de rellotge. El bucle consta de 256 iteracions (una per cada element del vector), així que el temps total d'execució del bucle serà de: 256 * 13 = 3328 cicles. Tenint en compte que la freqüència de rellotge d'1 GHz correspon a un període o cicle d'1 ns, el temps en d'execución del bucle en segons és:

$$\text{Temps d'execució} = 3328 \text{ cicles} \times 10^{-9} \text{ segons/cicle} = 3,328 \ \mu\text{s}$$

2. A cada iteració del bucle es fa una operació en coma flotant (add.s); per tant en la seua execució s'hi fan 256 operacions en coma flotante. Tenint en compte el temps total d'execució d'aquest bucle, la productivitat del bucle serà:

$$\text{Productivitat} = \frac{256 \text{ operacions}}{3,33\ \mu\text{s}} = 76,9 \cdot 10^6 \text{ operacions/s} = 76,9 \text{ MFLOPS}$$

■

---

4 Operadors de coma flotant

PROBLEMA 24 A la figura 1 teniu l'estructura d'un operador que fa la conversió d'enter amb signe a coma flotant de simple precisió.

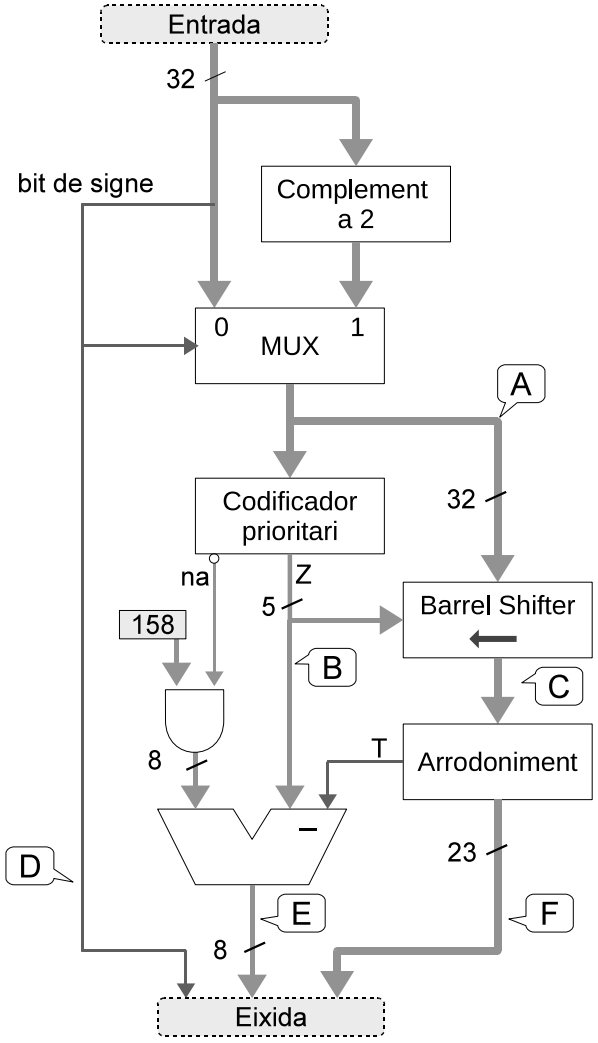

Figura 1: Estructura del conversor de *word* a *float*

El temps de retard i la funció dels components són:

- **Complement a 2** (3 ns): Operador que fa el complement a dos d'un enter de 32 bits.
- **MUX** (1 ns): Multiplexor de dues entrades de 32 bits.
- **Codificador prioritari** (2 ns): Codificador de 32 entrades. Si una o més entrades valen 1, el circuit actua com a detector de zeros per l'esquerra tornant un valor Z comprés entre 0 i 31 i l'eixida na^* queda a nivell alt. Si totes les entrades valen 0, el valor codificat és $Z = 0$ i l'eixida na^* queda a nivell baix.

- **Barrel shifter** (2 ns): Desplaçador variable cap a l'esquerra per a números de 32 bits. Governat per Z, pot fer desplaçaments d'entre 0 i 31 posicions.
- **Circuit d'arrodoniment** (3 ns): Elimina els vuit bits menys significatius de la mantissa de 32 bits que rep a l'entrada, aplicant l'arrodoniment cap a la mantissa de 24 bits més propera (esbiaixat cap al parell), i n'elimina el bit més significatiu per deixar-lo implícit. L'eixida T indica a nivell alt que l'arrodoniment ha produït un transport.
- **Restador modificat** (3 ns): Si $na^* = 1$, calcula l'exponent representat en excés 127 fent l'operació $158 - Z + T$. Si $na^* = 0$ calcula $-Z + T$.

1. Calculeu els valors que apareixen en els punts d'observació A, B, C, D, E i F de la figura 1 per als valors de l'entrada 1023 (*0x3FF*), -2 (*0xFFFFFFFE*), 0 i $2^{31} - 1$ (*0x7FFFFFFF*). Doneu els valors B i E en decimal i la resta en hexadecimal. Podeu omplir una taula com aquesta:

Entrada	*A (hex)*	*B (dec)*	*C (hex)*	*D (hex)*	*E (dec)*	*F (hex)*

2. Raoneu quines modificacions caldria fer per adaptar aquest operador perquè generara resultats en doble precisió (exponent d'11 bits en excés 1023 i mantissa de 52 bits): El canvi afectaria al multiplexor, al circuit de desplaçament o al codificador prioritari? I al restador? Caldria un circuit d'arrodoniment?
3. Calculeu en MOPS la màxima productivitat de l'operador de la figura 1.

SOLUCIÓ

1. Taula

Entrada	*A (hex)*	*B (dec)*	*C (hex)*	*D (hex)*	*E (dec)*	*F (hex)*
1023	0000 03FF	22	FFC0 0000	0	136	7FC 000
-2	0000 0002	30	8000 0000	1	128	000 000
0	0000 0000	0	0000 0000	0	0	000 000
$2^{31} - 1$	7FFF FFFF	1	FFFF FFFE	0	158	000 000

2. L'operador equivalent en doble precisió mantindria tots els components del de la figura 1 llevat que:
   - No hi caldria circuit d'arrodoniment, perquè l'espai de la mantissa en doble precisió és més gran que els 32 bits de l'entrada.
   - El restador hauria d'operar amb 11 bits i fer l'operació $1055 - Z$
3. El camí crític de l'operador es aquell que passa pel circuit de complement a 2, el multiplexor, el codificador prioritari, el *barrel shifter*, el circuit d'arrodoniment i el restador final. Sumant els retards, tenim un retard total de 12 ns i una productivitat de 83,3 MOPS.

■