Aritmética en coma flotante

Índice

1. El juego de instrucciones de coma flotante

PROBLEMA 1 El siguiente código escrito en lenguaje de alto nivel corresponde a una función que calcula el perímetro de una circunferencia a partir de su radio:

```
float perimetro (float radio)
{
    return (2.0 * 3.1415 * radio);
}
```

Implemente esta función en código ensamblador del MIPS R2000. Suponga que los dos valores constantes se encuentran ubicados en el segmento de datos.

SOLUCIÓN Como las dos constantes de la función (2,0 y 3,1415) están ubicadas en la memoria, la directiva que permite codificar los valores en el estándar IEEE 754 de simple precisión es `.float`. Por otra parte, la variable `radio` se pasará a la subrutina por medio del registro `$f12` y el resultado calculado se deijará en el registro `$f0`. El código en ensamblador será el sigüiente:

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

            .text 0x400000
            ...
perimetro:  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 las constantes
            mul.s $f0, $f0, $f12    # cálculo del perímetro
            jr $ra
```

■

PROBLEMA 2 Modifique el código en ensamblador del problema 1 para que considere todos los valores involucrados codificados en doble precisión.

SOLUCIÓN En este caso deberíamos de tener en cuenta que las variables serán de tipo `double` y, por tanto, ocuparán 64 bits. Las instrucciones de multiplicación también se tendrán que modificar para que operen con valores de este tipo. En el caso de las lecturas de memoria, el MIPS R2000 sólo dispone de instrucciones para leer palabras de 32 bits, por lo que la carga de un valor expresado en doble precisión necesita dos instrucciones `lwc1`.

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

            .text 0x400000
            ...
perimetre: la $t0, 0x10000000
            lwc1 $f4, 0($t0)        # lee parte baja de 2.0
            lwc1 $f5, 4($t0)        # lee parte alta de 2.0
            lwc1 $f6, 8($t0)        # lee parte baja de 3.1415
            lwc1 $f7, 12($t0)       # lee parte alta de 3.1415
            mul.d $f0, $f4, $f6     # producto de las dos constantes
            mul.d $f0, $f0, $f12    # cálculo del perímetro
            jr $ra
```

■

PROBLEMA 3 Determine lo que hace el código siguiente escrito en lenguaje ensamblador 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
```

SOLUCIÓN La primera instrucción del programa copia el contenido del registro de coma flotante `$f0` en el registro entero `$t0`. A continuación el código extrae, por medio de tres máscaras de bits, los tres campos que conforman el estándar de codificación de números de coma flotante IEEE 754 de simple precisión. Así, en `$t1` deja el bit de signo, en `$t2` los ocho bits del exponente y, finalmente, en `$t3` los 23 bits de la mantisa. Después, mediante desplazamientos lógicos hacia la derecha, alinea los bits obtenidos del signo y del exponente en la parte baja del registro donde se han copiado. A la mantisa, que ya está en la parte más baja del registro, le añade el bit implícito.

■

PROBLEMA 4 Algunos ensambladores del MIPS admiten las pseudoinstrucciones `li.s` y `li.d` que permiten cargar una constante en un registro de coma flotante. La traducción de estas pseudoinstrucciones es una serie de instrucciones que forman la representación de la constante en un registro entero y hacen la transferencia de esta representación al banco de coma flotante, como muestra el ejemplo siguiente:

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

Escriba una traducción posible de las pseudoinstrucciones siguientes en secuencias de instrucciones 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`

SOLUCIÓN

1. Traducción de `li.s $f0, +0.0`

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

2. Traducción de `li.d $f0, -0.0`

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

3. Traducción de `li.d $f0, -2.75`

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

4. Traducción de `li.s $f0, +Inf`

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

■

PROBLEMA 5 Considere el siguiente código escrito en lenguaje de alto nivel:

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

Implemente este código en lenguaje ensamblador del MIPS R2000. Suponga que la variable `f` se encuentra inicialmente en el registro `$f18`.

SOLUCIÓN El código en lenguaje ensamblador que proponemos para implementar el código anterior es el siguiente:

```
        li.s $f4, 1707.0        # $f4 contiene el valor 1707.0
        li.s $f6, 0.0001        # $f6 contiene el valor 0.0001
bucle:  div.s $f18, $f18,$f4    # división y asignación de f
        c.lt.s $f18, $f6        # comparación del while
        bc1t bucle              # volver al bucle si f<0.0001
```

Las dos primeras pseudoinstrucciones cargan los valores constantes 1707,0 y 0,0001 en dos registros temporales del coprocesador matemático. Note que la traducción de estas dos pseudoinstrucciones a instrucciones máquina la hace el programa ensamblador; esta traducción implica, también, la traducción de las constantes del programa a las cadenas de bits correspondientes codificadas según las normas del estándar IEEE 754.

A continuación el bucle se limita a hacer la división y la asignación de la variable `f` y evaluar la condición. Esta condición de "menor que" se comprueba con la instrucción `c.lt.s` (las letras `lt` indican *lower than*). Si la condición se cumple, el indicador de condición `FPc` del registro de estado del procesador (bit número 23) se pone a 1 y, en caso contrario, a 0. La última instrucción, `bc1t`, saltarán a la primera instrucción del bucle cuando la comprobación ponga el indicador `FPc` a 1.

■

PROBLEMA 6 El convenio de programación en lenguaje ensamblador del MIPS R2000 propone el uso de los registros de coma flotante siguiente:

Nombre	Uso
`$f0`	Retorno de función (parte real)
`$f2`	Retorno de función (parte imaginaria)
`$f4..$f10`	Registros temporales
`$f12,$f14`	Paso de parámetros
`$f16,$f18`	Registros temporales
`$f20..$f30`	Registros a preservar

donde cada registro de estos 16 se puede tratar como un registro de 64 bits donde se almacema un número en formato IEEE 754 de doble precisión (DP) o como un registro de 32 bits para números en simple precisión (SP). Vea que el convenio sólo referencia los registros con índices pares (`$f0`, `$f2`, ..., `$f30`); por tanto los registros impares (`$f1`, `$f3`, ..., `$f31`) sólo se utilizan para contener la parte alta de valores en doble precisión.

Note que la distribución de los registros previene el cálculo de variable compleja, y por esto las funciones pueden devolver en `$f2` la parte imaginaria del resultado. Las funciones de variable real sólo devolverán sus valores en `$f0`. En consecuencia, las funciones tomarán como parámetros los valores contenidos en `$f12` y `$f14` y depositarán el resultado calculado en los registros `$f0` (parte real) y `$f2` (parte imaginaria, si ha lugar). Finalmente, consideremos seis registros temporales: `$f4`, `$f6`, `$f8`, `$f10`, `$f16` y `$f18`.

Siguiendo este convenio, construya una biblioteca de funciones de coma flotante que implementen los cálculos siguientes:

1. `float F2C_s(float a)`. Hace la conversión de una temperatura t_F, expresada en grados Fahrenheit, a t_C, expresada en grados Celsius, según la fórmula:

$$t_C = \frac{5{,}0}{9{,}0} \times (t_F - 32{,}0)$$

2. `float sqr_s(float a)`. Calcula el cuadrado de dos valores reales en simple precisión.

3. `double add_sd(float a, double b)`. Calcula la suma de dos valores reales de tipo distinto: el parámetro `a` (ubicado en `$f12`) es de simple precisión y el parámetro b (en el registro `$f14`) es de doble precisión. El resultado de la función es un valor real de doble precisión.

4. `addcx_s`. Calcula la suma de dos valores complejos de simple precisión. Como es un caso complicado, no haga mucho caso del convenio: considere que el primer sumando es `$f12` (parte real) y `$f13` (parte imaginaria) y que el segundo sumando está en `$f14` y `$f15`.

5. `float max_s (float a, float b)`. Calcula el máximo de dos valores reales en simple precisión.

6. `double max_d (double a, double b)`. Calcula el máximo de dos valores reales expresados en doble precisión.

SOLUCIÓN Para cada función hay que diseñar el código en ensamblador correspondiente. Note que, si no se dice nada que lo contradiga, habrá que respetar el convenio de uso de los registros.

1. Función `sqr_s`: el código calcula primero el cociente $5{,}0/9{,}0$ empleandp valores reales (no enteros), después calcula el resto que hay entre paréntesis y, finalmente, hace la multiplicación:

```
F2C_s:  li.s $f4, 5.0           # Constante 5.0 en $f4
        li.s $f6, 9.0           # Constante 9.0 en $f6
        div.s $f4, $f4, $f6     # Cálculo de la fracción 5.0/9.0
        li.s $f6, 30.0          # Constante 32.0 en $f4
        sub.s $f0, $f12, $f6    # Cálculo tF - 32.0
        mul.s $f0, $f0, $f4     # Multiplicación final
        jr $ra
```

2. Función sqr_s: en este caso el código es muy senzillo, ya que habrá bastante con multiplicar el parámetro de entrada (registro $f12) por sí mismo y retornar el valor calculado en el registro $f0:

```
sqr_s:  mul.s $f0, $f12, $f12  # Elevamos al cuadrado
        jr $ra
```

3. Función add_sd: como el resultado ha de ser un valor en doble precisión, hay que hacer una conversión previa del parámetro de simple precisión:

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

4. Función addcx_s: en este caso tendremos de sumar las partes reales entre sí y las imaginarias entre sí, y dejar el resultado en $f0 y $f2, respectivamente:

```
addcx_s:  add.s $f0, $f12, $f14   # Cálculo parte real
          add.s $f2, $f13, $f15   # Cálculo parte imaginaria
          jr $ra
```

5. Función max_s: se ha de hacer un comparación entre los dos parámetros de la función; la instrucción que utilizaremos es c.le.s (establece si el primer registro es menor que el segundo). Si la condición se cumple entonces el indicador de condición FPc del registro del estado del coprocesador matemático se pone a 1 y, consiguientemente, la instrucción bc1t provocara un salto:

```
max_s:      c.le.s $f12, $f14  # Comparación menor que
            bc1t f14_gana      # Si salta $f14 es mayor
f12_gana:   mov.s $f0, $f12    # Retorna el valor de $f12
            jr $ra
f14_gana:   mov.s $f0, $f14    # Retorna el valor de $f14
            jr $ra
```

6. Función max_d: este caso es similar al anterior, excepto que los valores vienen expresados en doble precisión y, por tanto, la instrucción de comparación ahora serà c.le.d. Además, note que los valores manejados en el código son de 64 bits (suffix .d en la instrucción de movimiento de datos):

```
max_d:      c.le.d $f12, $f14  # Comparación menor que
            bc1t f14_gana      # Si salta $f14 es mayor
f12_gana:   mov.d $f0, $f12    # Retorna el valor de $f12
            jr $ra
f14_gana:   mov.d $f0, $f14    # Retorna el valor de $f14
            jr $ra
```

■

PROBLEMA 7 Escriba una subrutina es_NaN que, dado un número real en el registro $f12, devuelva en $v0 un 1 si es un NaN o un 0 en caso contrario.

SOLUCIÓN La subrutina analiza el exponente para averigüar si vale 255 (todo a unos). En el caso que tenga este valor y, además, la mantisa sea diferente de cero, entonces será un NaN.

```
es_NaN: mfc1 $t0, $f12          # copia $f12 en $t0
        li $t1, 0x7F800000      # máscara para el exponente
        and $t2, $t0, $t1       # exponente en $t2
        bne $t1, $t2, noes      # si exp<>255 no es NaN
        li $t1, 0x007FFFFF      # máscara para la mantisa
        and $t2, $t0, $t1       # mantisa en $t2
        beq $t2, $zero, noes    # si mantisa==0 no es NaN
sies:   li $v0, 1
        j fin
noes:   li $v0, 0
fin:    jr $ra
```

■

PROBLEMA 8 En la siguiente sección de datos se definen un número real y uno entero. Se pide escribir el código correspondiente que sume estos dos números y deje el resultado de la suma en la dirección definida con la etiqueta `resultado`.

```
            .data
entero:     .word 2
real:       .float 3.5
resultado:  .float 0.0
            .text
            ....
            .end
```

Para implementar el programa se deben utilizar sólo registros temporales del banco de registros de enteros (`$t0`, ...,`$t9`) y del banco de registros de reales (`$f4`,...,`$f10`).

SOLUCIÓN Para poder sumar los dos números, ambos deben codificarse con el estándar IEEE754, por lo tanto han de estar en el banco de registros de reales.

```
.text
lwc1 $f6, entero   # lectura de la variable "entero"
cvt.s.w $f6,$f6    # conversión de la variable "entero" a coma flotante
lwc1 $f4, real     # lectura de la variable "real"
add.s $f4,$f4,$f6  # suma en coma flotante
swc1 $f4,resultado # escritura del resultado
.end
```

Note que el código anterior ha leído la variable entera de la memoria y la ha depositado en el banco de registros de coma flotante para su tratamiento. Una alternativa a `lwc1 $f6, entero`, donde el valor de la variable pasa por el banco de registros de enteros, sería la siguiente:

```
lw $t0, entero     # lee la variable "entero"
mtc1 $t0, $f6      # copia el valor en un registro de coma flotante
```

■

PROBLEMA 9 Escribe un segmento de programa que calcule la suma de los enteros ubicados en los registros `$t0` y `$t1` utilizando instrucciones de coma flotante y deje el resultado en el registro `$v0`.

SOLUCIÓN Esta es una situación que se da algunas veces cuando programamos en un lenguaje de alto nivel: hay dos enteros que queremos sumar, pero la suma se hace sobre los valores reales equivalentes a los enteros. Finalmente, el resultado de esta suma se vuelve a convertir en un valor entero. Un ejemplo sencillo en alto nivel que ilustra esta situación (uso del citado *casting*) es el siguiente:

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

Desde el punto de vista del procesador, esta operación implica un movimiento de los valores enteros (32 bits) contenidos en los registros `$t0` y `$t1` a los registros `$f4` y `$f6` (por ejemplo) del coprocesador matemático (los valores reales que empleamos son de simple precisión). Después de este movimiento de datos, se ha de hacer una conversión de tipos (de entero a real), ya que la instrucción `mtc1` se limita a copiar, tal cual, el valor del registro fuente en el destino. Una vez hecha la suma, el resultado real se ha de convertir a enteros y después se ha de copiar en el registro entero `$v0`. El código en ensamblador que hace todas estas operaciones es el siguiente:

```
mtc1 $t0, $f4       # Copia $t0 en $f4
mtc1 $t1, $f6       # Copia $t1 en $f6
cvt.s.w $f4, $f4    # Conversión de entero a real
cvt.s.w $f6, $f6    # Conversión de entero a real
add.s $f4, $f4, $f6 # Suma de dos valores reales
cvt.w.s $f4, $f4    # Conversión de real a entero
mfc1 $v0, $f4       # Copia el resultado en $v0
```

■

PROBLEMA 10 Ha de diseñar una biblioteca de funciones de cálculo que combinen operandos enteros y de coma flotante. Utilice los registros de acuerdo con los criterios habituales de paso de parámetros y retorno de valores. La lista de funciones que componen esta biblioteca es la siguiente:

1. `add_ws`. Calcula la suma de un entero (*int*) y un real de simple precisión (*float*). El resultado ha de ser un valor real de simple precisión.
2. `mul_wsd`. Calcula el producto de un entero (*int*), un real de simple precisión (*float*) y un real de doble precisión (*double*). El resultado ha de ser un valor real de doble precisión.

SOLUCIÓN Las funciones que hay que diseñar incluirán, necesariamente, instrucciones de conversión de tipo, concretamente de entero a real. Esta conversión es imprescindible porque las operaciones de suma y multiplicación se hacen sobre números expresados en coma flotante.

1. Función `add_ws`. Se ha de hacer una conversión del valor entero en un real de simple precisión antes de llevar a término la suma:

```
add_ws: mtc1 $a0, $f16        # Copia parámetro real en $f16
        cvt.s.w $f0, $f16     # Conversión de entero a real
        add.s $f0, $f0, $f12  # Suma de valors reales
        jr $ra
```

2. Función `mul_wsd`. En este caso se han de hacer dos conversiones: la primera, de real en simple precisión a doble precisión; la segunda, de entero a real de doble precisión:

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

■

PROBLEMA 11 Escriba una subrutina `m_arit` que calcule la media aritmética de los elementos de un vector. En el registro `$a0` recibe la dirección del vector y en `$a1` la dimensión; el resultado calculado se retorna en `$f0`. Puede basarse en el siguiente código de alto nivel:

```
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);
}
```

Nótese que la última división de la función hace una conversión previa de la variable `n` en valor real.

SOLUCIÓN La subrutina implementa un bucle de lectura de los elementos del vector. Los elementos son leídos con la instrucción `lwc1`. Para hacer la división entre el número de elementos habrá que hacer una convesión de tipo (entero 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         # pone 0.0 en $f0

bucle:  lwc1 $f4, 0($t0)        # lee elemento A[i]
        add.s $f0, $f0, $f4     # incrementa $f0 con A[i]
        addiu $t0, $t0, 4       # actualiza dirección a A[i+1]
        addi $t1, $t1, -1       # decrementa contador
        bne $t1, $zero, bucle   # salta si quedan elementos

        mtc1 $a1, $f4           # copia $a1 (dimensión) en $f4
        cvt.s.w $f4, $f4        # convierte la dimensión en real
        div.s $f0, $f0, $f4     # divide entre la dimensión
        jr $ra
```

■

PROBLEMA 12 Considere el código siguiente, escrito en lenguaje de alto nivel, que calcula la media armónica de los valores contenidos en un vector de reales de simple precisión:

```
float armonica (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);
}
```

Hay que recalcar que la última división de la función hace una conversión previa de la variable `n` a valor real. Implemente esta función en ensamblador del MIPS R2000.

SOLUCIÓN La variable `n` se pasará a la función por medio del registro `$a0`, la dirección de comienzo del vector en memoria por medio del registro `$a1`, y el resultado calculado se dejará en el registro `$f0`. El código en ensamblador será el siguiente:

```
armonica:   li.s $f4, 0.0           # aux en $f4 a zero
            li.s $f6, 1.0           # $f6 contiene 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     # cálculo de 1.0/v[i]
            add.s $f4, $f4, $f8     # incremento de aux
            addi $a1, $a1, 4        # incremento de la dirección
            addi $t0, $t0, -1       # decremento de i
            bne $t0, $zero, bucle   # salta si diferente de zero
            mtc1 $a0, $f6           # copia n en el coprocesador 1
            cvt.s.w $f6, $f6        # convierte n a float
            mul.s $f12, $f6, $f4    # producto final
            jr $ra
```

Como detalle a recalcar, la pseudoinstrucción `li.s` se usa para cargar valores constantes en registros del coprocesador CP1. Por ejemplo, `li.s $f6, 1.0` carga el valor 1,0 en el registro `$f6` del coprocesador matemático codificándolo de acuerdo con el estándar IEEE 754 de simple precisión.

■

PROBLEMA 13 Considere los siguientes fragmentos de programa: el fragmento *A* corresponde a un programa en ensamblador del MIPS R2000; los fragmentos *B* y *C* están escritos en un lenguaje de alto nivel cuyo tipo `int` es un entero de 32 bits con signo.

Fragmento *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:
```

Fragmento *B*:

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

Fragmento *C*:

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

1. Escriba el programa de alto nivel equivalente al fragmento *A*.
2. Escriba el programa en ensamblador equivalente al fragmento *B*.
3. Escriba el programa en ensamblador equivalente al fragmento *C*.

SOLUCIÓN

1. Fragmento A en alto nivel:

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

2. Fragmento B en ensamblador:

```
            .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. Fragmento C en ensamblador:

```
            .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 Codifique en ensamblador del MIPS R2000 la función `calculo` que se muestra a continuación. La función recibe como parámetros cuatro **punteros** a memoria codificados en los registros `$a0`, `$a1`, `$a2`, `$a3`. La función devuelve un valor de tipo `double` (valor en coma flotante de doble precisión) en el par de registros `$f0`, `$f1`.

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

SOLUCIÓN El código de la función en ensamblador se muestra a continuación.

```
calculo: 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 lugar se cargan en los registros `$t0` y `$t1` los valores de las variables `a` y `b`, respectivamente. Para ello utilizamos la instrucción `lw` ya que los parámetros pasados a la función son punteros a las variables. A continuación comparamos las dos variables (`a<b`) con la pseudo-instrucción `blt` (una alternativa es utilizar las instrucciones `slt` y `bne`).

Por último, en cada una de las dos posibles alternativas de la comparación se realizan las operaciones en coma flotante pertinentes (multiplicación con la instrucción `mul.s` o suma con la instrucción `add.s`). Ahora bien, en cada uno de los casos hay que realizar conversiones con la instrucción `cvt` y transferencias entre registros de enteros y de coma flotante (instrucción `mtc1`) o transferencias entre memoria y registros de coma flotante (instrucción `lwc1`).

■

PROBLEMA 15 Realizar una función en ensamblador del MIPS R2000 para calcular el factorial de un número entero $x!$, cuya expresión matemática es: $x! = x \cdot (x-1) \cdot (x-2) \cdot ... \cdot 2 \cdot 1$. El perfil de la función en alto nivel es:

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

El valor del número entero se le pasa a la función mediante `$a0`, devolviendo el resultado como un número real (`double`) en el registro `$f12`. Para implementar el programa se deben utilizar sólo registros temporales del banco de registros de enteros (`$t0`,...,`$t9`) y del banco de registros de reales (`$f4`,...,`$f10`).

SOLUCIÓN

```
            .text
Factorial: mtc1    $a0,$f0        # Pasamos el número en $a0 a $f0
           cvt.d.w $f0,$f0        # convertimos a double
           abs     $f0,$f0        # lo pasamos a positivo por si acaso
           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       # es $f0 <> 1.0?
           bclf    bucle          # salta a bucle
           jr      $ra            # resultado en $f12
```

■

PROBLEMA 16 Estamos realizando una biblioteca de funciones aritméticas de coma flotante para un procesador MIPS versión IV. En particular, se desea implementar una función de transformación de coordenadas. Sea $\bar{x} = (x_0, x_1, x_2)$ un vector de *3* coordenadas (enteros) y sea R una matriz de transformación de coordenadas (double), de dimensión *3x3*. El resultado de la transformación es otro vector $\bar{y}$ que se obtiene mediante el producto $\bar{y} = R \cdot \bar{x}$:

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

La versión de alto nivel de esta función es:

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

donde:

- *x: dirección de inicio del vector x. Se pasará en $a0.
- *R: dirección de inicio de la matriz R. Se pasará en $a1.
- *y: dirección de inicio del vector y de resultado. Se pasará en $a2.

A continuación se muestra una versión del código correspondiente a la subrutina *transf_coord*. Tenga en cuenta que la matriz R está almacenada en memoria por filas y que sus componentes son valores en doble precisión, mientras que los componentes del vector x son enteros. En esta solución se obtiene cada componente del vector resultado y mediante el producto escalar de la fila correspondiente de R por el vector de entrada x.

```
transf_coord: li $t3,3            # índice i=3
bucle:        jal prod_escalar    # producto escalar $a0 = dir x $a1= dir fila R[i]
              swc1 $f12, 0($a2)   # resultado $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
```

Escriba el código correspondiente a la subrutina *prod_escalar*.

SOLUCIÓN

```
prod_escalar: li $t2,3          # Índice j=3
              mov $t0,$a0       # se copian $a0 y $a1
              mov $t1,$a1
              mtc1 $f12, $0      # Inicializa suma ($f12/13) = 0
              mtc1 $f13, $0
bulce_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. Emulación de instrucciones de coma flotante

PROBLEMA 17 Escriba el código en lenguaje ensamblador del procesador MIPS R2000 que emule, con instrucciones enteras, las siguientes intrucciones de coma flotante:

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

SOLUCIÓN

1. `abs.s $f0, $f4`. Se trata de obtener en `$f0` el valor absoluto del valor de coma flotante de simple precisión contenido en `$f4`. El efecto neto consiste en hacer positivo el valor de `$f4`, es decir, poner el bit de signo a cero. Como hay que utilizar instrucciones enteras, podemos aplicar una máscara de 32 bits consistente en 31 bits a uno (los de menor peso)y el bit más significativo (el de mayor peso) a cero:

```
mfc1 $t0, $f4         # Copia $f4 en $t0
li $t1, 0x7FFFFFFF    # Máscara de bits en $t1
and $t0, $t0, $t1     # Bit más significativo a cero
mtc1 $t0, $f0         # Copia $t0 en $f0
```

2. `abs.d $f0, $f4`. Este caso es similar al anterior, pero amb la diferència que els valors són de 64 bits (doble precisión). El valor real original está contenido en los registros `$f4` (parte menos significativa) y `$f5` (parte más significativa). Así pues, el bit de signo es el más significativo, no del registro `$f4`, sino del registro `$f5`, que contiene la parte más alta de valor real (el bit de signo, once bits del exponente y una parte de la mantisa). Por tanto, podemos escribir:

```
mfc1 $t0, $f5          # Copia $f5 en $t0
li $t1, 0x7FFFFFFF     # Máscara de bits en $t1
and $t0, $t0, $t1      # Bit más significativo a cero
mtc1 $t0, $f1          # Bit de signo a cero
mov.s $f0, $f4         # Parte baja sin modificar
```

3. `neg.s $f0, $f4`. Esta instrucción cambia el signo del valor contenido en el registro `$f4` y retorna el resultado en `$f0`. La inversión del bit de signo se puede llevar a cabo mediante una operació lógica OR exclusiva:

```
mfc1 $t0, $f4          # Copia $f4 en $t0
li $t1, 0x80000000     # Máscara de bits en $t1
xor $t0, $t0, $t1      # Inversión del bit de signo
mtc1 $t0, $f0          # Copia de $t0 en $f0
```

■

PROBLEMA 18 Escriba en ensamblador del MIPS R2000 las funciones de un parámetro indicado en el el registro `$f12`, que se piden a continuación. El resultado calculado se deja en el registro `$f0`. No hay que tratar los posibles desbordamientos de los exponentes.

1. `mul2_s`: multiplicación por dos en simple precisión.
2. `mul2_d`: multiplicación por dos en doble precisión.
3. `div2_s`: división por dos en simple precisión.
4. `div4_s`: división por cuatro en simple precisión.

SOLUCIÓN En las soluciones que proponemos el procedimiento a seguir es muy fácil: se trata de manipular adecuadametne los bits que representan el exponente del valor real. Así, como la base implítica del exponente en el estándar es 2, multiplicar por dos equivale a sumar un uno al exponente, mientras que la división equivale a restarle un uno. Veamos cada caso por separado.

1. `mul2_s`. En este caso hemos de sumar un uno al exponente. Las posiciones que ocupan los 8 bits del exponente son las de mayor peso (sin considerar el bit de signo). En particular, si hemos de sumar un uno al exponente, la posición que ocupa el bit de menor peso del exponente será la 23a (contando desde cero). El código que se pide es el siguiente:

```
mul2_s: mfc1 $t0, $f12      # Copia $f12 en $t0
        li $t1,0x00800000   # Máscara: un uno en la 23a posició
        add $t0, $t0, $t1   # Suma un uno al exponente
        mtc1 $t0, $f0       # Copia $t0 en $f0
        jr $ra
```

2. `mul2_d`. Se trata de un caso similar al anterior, pero ahora se ha de tener en cuenta que el exponente, situado en los 32 bits de mayor peso del valor real (registro `$f13`), tiene 11 bits, cuyo bit menos significativo ocupa la posición 20 (contando desde cero):

```
mul2_d: mfc1 $t0, $f13       # Copia parte más significativa
        li $t1, 0x00100000   # Máscara: un uno en la 20a posición
        add $t0, $t0, $t1    # Suma un uno al exponente
        mtc1 $t0, $f1        # Copia $t0 en $f1 (modificado)
        mov.s $f0, $f12      # Copia $f12 en $f0 (no modificado)
        jr $ra
```

3. `div2_s`. A diferencia de los casos anteriores, en este se trata de hacer una resta. El código que resulta es similar al utilizado en la implementación de la función `mul2_s`, ya que únicamente cambia la instrucción `add` por la `sub`:

```
div2_s: mfc1 $t0, $f12      # Copia $f12 en $t0
        li $t1,0x00800000   # Máscara: un uno en la 23a posición
        sub $t0, $t0, $t1   # Resta un uno a el exponente
        mtc1 $t0, $f0       # Copia $t0 en $f0
        jr $ra
```

4. `div4_s`. En este caso se trata de restar dos unidades al exponente, es decir, habremos de restar un uno pero ahora en la posición 24:

```
div4_s: mfc1 $t0, $f12      # Copia de $f12 en $t0
        li $t1,0x01000000   # Máscara: un uno en la 24a posición
        sub $t0, $t0, $t1   # Resta un dos al exponente
        mtc1 $t0, $f0       # Copia $t0 en $f0
        jr $ra
```

■

PROBLEMA 19 Escriba en ensamblador del MIPS R2000 una subrutina `mul2_s` que, dado un número real en el registro `$f12`, emule una multiplicación por 2, y deje el resultado en el registro `$f0`. La subrutina ha de tener en cuenta los casos especiales (cero, infinito y NaN).

SOLUCIÓN La siguiente subrutina trata los casos especiales por medio del análisis del valor del exponente (casos en que vale 0 y 255).

```
mul2_s: mfc1 $t0, $f12          # copia $f12 en $t0
        li $t1, 0x7FFFFFFF      # máscara para E y M
        and $t1, $t0, $t1       # extrae E y M en $t1
        beq $t1, $cero, eixir   # si E=M=0 es +0.0 o bien -0.0

        li $t1, 0x7F800000      # máscara para E
        and $t2, $t0, $t1       # extrae E en $t2
        beq $t1, $t2, eixir     # si exp==255 es NaN, +Infty o -Infty

        li $t1, 0x00800000      # máscara: un uno en la 23a posició
        add $t0, $t0, $t1       # suma un uno al exponente
        li $t1, 0x7F800000      # máscara para E
        and $t2, $t0, $t1       # extrae E en $t2
        bne $t2, $t1, eixir     # si exp<>255 es un número "normal"
                                # si exp==255 ha desbordado y hay que generar un infinito
        li $t1, 0xFF800000      # máscara con M a cero
        and $t0, $t0, $t1       # pone M del resultado a cero

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

3. Medida de prestaciones

PROBLEMA 20 El segmento de programa siguiente tarda exactamente 7 ms en ejecutarse en un computador basado 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
```

Calcule en MFLOPS la productividad alcanzada por el programa.

SOLUCIÓN Como conocemos el tiempo de ejecución del código, sólo hay que calcular el número N_{CF} de operaciones en coma flotante que se hacen. En particular, fuera del bucle podemos contabilizar tres: la segunda instrucción del programa (`add.s`) y las dos últimas instrucciones (`div.s` y `abs.s`). El bucle, que se ejecuta un total de 25000 veces (valor inicial del registro temporal `$t1`), contiene un total de dos instrucciones de coma flotante (`mul.s` y `sub.s`).

$$N_{CF} = 3 \text{ operaciones fuera del bucle} + 25000 \text{ iteraciones} \times 2 \text{ operaciones/iteración} = 50003 \text{ operaciones}$$

Por lo tanto, la productividad alcanzada por el programa será:

$$\text{Productividad} = \frac{N_{CF}}{\text{tiempo}} = \frac{50003 \text{ operaciones}}{7 \cdot 10^{-3} \text{ segundos}} = 7{,}14 \cdot 10^6 \text{ operaciones/segundo} = 7{,}14 \text{ MFLOPS}$$

■

PROBLEMA 21 Dispone de un operador combinacional que puede realizar dos operaciones de coma flotante: suma y multiplicación. Las operaciones se seleccionan mediante una entrada de control y tienen retardos distintos. La productividad máxima de este operador depende de la operación seleccionada; en concreto, para la *suma* es de 1 MFLOPS y para la *multiplicación* de 2 MFLOPS.

1. ¿Cuál es el tiempo necesario para hacer una suma? ¿Y una multiplicación?
2. ¿Cuál es el tiempo mínimo necesario para calcular el producto escalar de dos vectores de 2000 elementos con este operador? Suponga que el producto escalar de vectores de n elementos requiere hacer n productos y n sumas.

SOLUCIÓN En este problema tendremos de hacer la conversión de MFLOPS a tiempo de operación, algo similar a la conversión entre frecuencia y período.

1. Veamos primero el tiempo característico para cada operación a partir de las productividades máximas expresadas en MFLOPS. Como 1 MFLOP $= 10^6$ FLOPS (operaciones de coma flotante por segundo), el tiempo T_{suma} necesario para hacer una suma es:

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

   Por su parte, aplicando el mismo razonamiento, el tiempo T_{mult} necesario para hacer una multiplicación es:

$$T_{\text{mult}} = \frac{1}{2 \cdot 10^6 \text{ FLOPS}} = 0{,}5 \ \mu\text{s}$$

Note que el circuito aritmético tarda más en hacer una suma que una multiplicación (justamente el doble), hecho que también se desprende directamente si comparamos los MFLOPS alcanzados por el circuito en operaciones de suma y en operaciones de multiplicación.

2. El tiempo T_{op} de operación total para el producto escalar será el necesario para hacer las 2000 sumas y las 2000 multiplicaciones:

$$T_{\text{op}} = 2000 \times 1 \ \mu s + 2000 \times 0{,}5 \ \mu s = 3 \text{ ms}$$

■

PROBLEMA 22 Dispone de un procesador MIPS R2000 con coprocesador matemático que funciona con una frecuencia de reloj de 50 MHz. Este procesador calcula el producto escalar de dos vectores `X` ty `Y` de 10000 elementos de reales de simple precisión mediante el bucle siguiente:

```
        li $t0, 0            # Desplazamiento inicial
        li $t1, 40000        # Desplazamiento final
        mtc1 $zero, $f0      # Carga un cero
bucle:  lwc1 $f4, X($t0)     # Lectura de X[i]
        lwc1 $f6,Y($t0)      # Lectura de Y[i]
        mul.s $f8, $f4, $f6  # Cálculo X[i]*Y[i]
        add.s $f0, $f0, $f8  # Actualización de suma
        addi $t0, $t0, 4     # Incremento desplazamiento
        bne $t0, $t1, bucle  # Salta si faltan elementos
```

Considere que el coste temporal de ejecución (en ciclos de reloj) de las instrucciones en el procesador es el siguiente:

Tipo de instrucción	Ciclos
Aritmética entera	1
Bifurcación	1
Lectura de memoria	2
Suma en coma flotante	4
Producto en coma flotante	7

1. Determine el tiempo de ejecución de este bucle.
2. Calcule la productividad del procesador, expresada en MFLOPS, ejecutando operaciones de coma flotante.
3. Suponga ahora que no hay coprocesador disponible, y las instrucciones de coma flotante se emulan mediante instrucciones de enteros. Una instrucción `mul.s` se emula con 60 instrucciones de enteros, y una instrucción `add.s` se emula con 30 instrucciones. Cuál sería el tiempo de ejecución de este bucle y la productividad en MFLOPS resultante?

SOLUCIÓN Antes que nada, calcularemos el tiempo de ciclo del procesador. Como la frecuencia de funcionamiento es de 50 MHz, entonces el tiempo de ciclo se puede obtener calculando la inversa: $1/(50 \text{ MHz}) = 1/(50 \cdot 10^6)$ s $=$ 20 ns.

1. Para saber cuánto tiempo tarda en ejecutarse el bucle hay que calcular el tiempo de una iteración. En particular, examinando el código, cada iteración pide $2 + 2 + 7 + 4 + 1 + 1 = 17$ ciclos de reloj. Así pues, el tiempo de ejecución será:

$$10000 \text{ iteraciones} \times 17 \text{ ciclos/iteración} \times 20 \cdot 10^{-9} \text{ s/ciclo} = 3{,}4 \text{ ms}$$

2. Para calcular la productividad del bucle expresada en MFLOPS hemos de tener en cuenta que, de las 17 instrucciones que se ejecutan en cada iteración del bucle, sólo dos son instrucciones de coma flotante (`mul.s` y `add.s`). Esto es, por cada 3,4 milisegundos que tarda cada iteración, se ejecuten dos instrucciones de coma flotante. Por lo tanto, la productividad conseguida en las 10000 iteraciones del bucle se puede expresar como:

$$\frac{2 \times 10000}{3{,}4 \cdot 10^{-3} \times 10^{6}} = 5{,}88 \text{ MFLOPS}$$

3. La ausencia de coprocesador matemático hace que las operaciones de coma flotante se lleven a cabo ejecutando instrucciones de aritmética entera. En el caso que consideramos, hay dos instrucciones dentro del bucle que serán traducidas (emuladas) por un conjunto concreto de instrucciones enteras, cada una de ellas tarda, según la tabla adjunta, un ciclo en ejecutarse. En particular, la ejecución de una operación `mul.s` emulada necesita 60 ciclos, porque será sustituida por 60 instrucciones de un ciclo de duración, mientras que la ejecución de la instrucción `add.s` necesitará 30 ciclos. En consecuencia, cada iteración exigirá un total de $2 + 2 + 60 + 30 + 1 + 1 = 96$ ciclos. El cálculo del tiempo de ejecución se hará de la siguiente manera:

$$10000 \times 96 \times 20 \cdot 10^{-9} = 19{,}2 \text{ ms}$$

Note que, con la emulación, el tiempo de ejecución ha pasado de 3,4 a 19,2 milisegundos, incremento de tiempo bastante notable. Por su parte, la productividad resultante, expresada en MFLOPS, será:

$$\frac{20000}{20 \cdot 10^{-3} \times 10^{6}} = 1{,}04 \text{ MFLOPS}$$

De forma parecida, la productividad del bucle ha caído de los 5,88 MFLOPS alcanzados con el coprocesador matemático a los 1,04 MFLOPS obtenidos con la emulación.

■

PROBLEMA 23 Se dispone de un procesador MIPS que funciona a una frecuencia de reloj de 1 GHz. Este procesador calcula la suma de un valor entero con un vector de 256 elementos reales en precisión simple ($\vec{Z}$ = a + $\vec{Y}$) mediante el programa siguiente:

```
        li $t0, 0         # desplazamiento inicial.
        li $t1, 1024      # desplazamiento final, 256 elementos float ocupan 1024 bytes.
        lwc1 $f0, a($t0)  # Se almacena el valor de "a" en $f0.
        cvt.s.w $f0, $f0  # Se convierte a formato real de precision simple el entero "a".
bucle:
        lwc1 $f1, Y($t0)        # lectura de Y[i].
        add.s $f2, $f0, $f1     # cálculo de "a + Y[i]".
        swc1 $f2, Z($t0)        # escritura de Z[i].
        addi $t0, $t0, 4        # incremento del desplazamiento.
        bne $t0, $t1, bucle     # salta si quedan elementos.
```

Considerando que el coste temporal de ejecución (en ciclos de reloj) de las instrucciones en el procesador es el siguiente:

Tipo de instrucción	Ciclos
Aritmética entera	1
Bifurcación	1
Lectura de memoria	3
Escritura en memoria	3
Suma en coma flotante	5

1. Determine el tiempo de ejecución en segundos del BUCLE ($\vec{Z}$ = a + $\vec{Y}$) empleado en el programa.
2. Calcule la productividad del bucle (sólo el BUCLE) expresada en millones de operaciones en coma flotante por segundo (MFLOPS).

SOLUCIÓN

1. En cada iteración del bucle se ejecutan: 1 instrucción de lectura en memoria, 1 instrucción de suma en coma flotante, 1 instrucción de escritura en memoria, 1 instrucción entera y 1 instrucción de salto. Por lo tanto, el tiempo de ejecución de cada iteración de este bucle será de: $3 + 5 + 3 + 1 + 1 = 13$ ciclos de reloj. El bucle consta de 256 iteraciones (una por cada elemento del vector), por lo que el tiempo total de ejecución del bucle será de: 256 * 13 = 3328 ciclos. Teniendo en cuenta que la frecuencia de reloj de 1 GHz corresponde con un periodo o ciclo de 1 ns, el tiempo en segundos de ejecución del bucle es:

$$\text{Tiempo de ejecución} = 3328 \text{ ciclos} \times 10^{-9} \text{ segundos/ciclo} = 3,33 \ \mu\text{s}$$

2. En cada iteración del bucle se realiza una operación en coma flotante (add.s), por lo que en su ejecución se realizan 256 operaciones en coma flotante. Teniendo en cuenta el tiempo total de ejecución de este bucle, la productividad del bucle será igual a:

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

■

---

4. Operadores de coma flotante

PROBLEMA 24 La figura 1 muestra la estructura de un operador que hace la conversión de entero con signo a coma flotante de simple precisión.

El tiempo de retardo y la función de los componentes son:

- **Complemento a 2** (3 ns): Operador que hace el complemento a dos de un entero de 32 bits.
- **MUX** (1 ns): Multiplexor de dos entradas de 32 bits.
- **Codificador prioritario** (2 ns): Codificador de 32 entradas. Si una o más entradas valen 1, el circuito actúa com un detector de ceros por la izquierda devolviendo un valor Z comprendido entre 0 y 31 y la salida na^* queda a nivel alto. Si todas las entradas valen 0, el valor codificado es $Z = 0$ y la salida na^* queda a nivel bajo.
- **Barrel shifter** (2 ns): Desplazador variable hacia la izquierda para números de 32 bits. Gobernado por Z, pude hacer desplazamientos de entre 0 y 31 posiciones.
- **Circuito de redondeo** (3 ns): Elimina los ocho bits menos significativos de la mantisa de 32 bits que recibe a la entrada, aplicando el redondeo hacia la mantisa de 24 bits más próxima (hacia el par), y elimina el bit más significativo para dejarlo implícito. La salida T indica a nivel alto que el redondeo ha producido un transporte.
- **Restador modificado** (3 ns): Si $na^* = 1$, calcula el exponente representado en exceso 127 haciendo la operación $158 - Z + T$. Si $na^* = 0$ calcula $-Z + T$.

1. Calcule los valores que aparecen en los puntos de observación A, B, C, D, E i F de la figura 1 para los valores de la entrada 1023 (*0x3FF*), -2 (*0xFFFFFFFE*), 0 y $2^{31} - 1$ (*0x7FFFFFFF*). Dé los valores B y E en decimal y el resto en hexadecimal. Puede rellenar una tabla como esta:

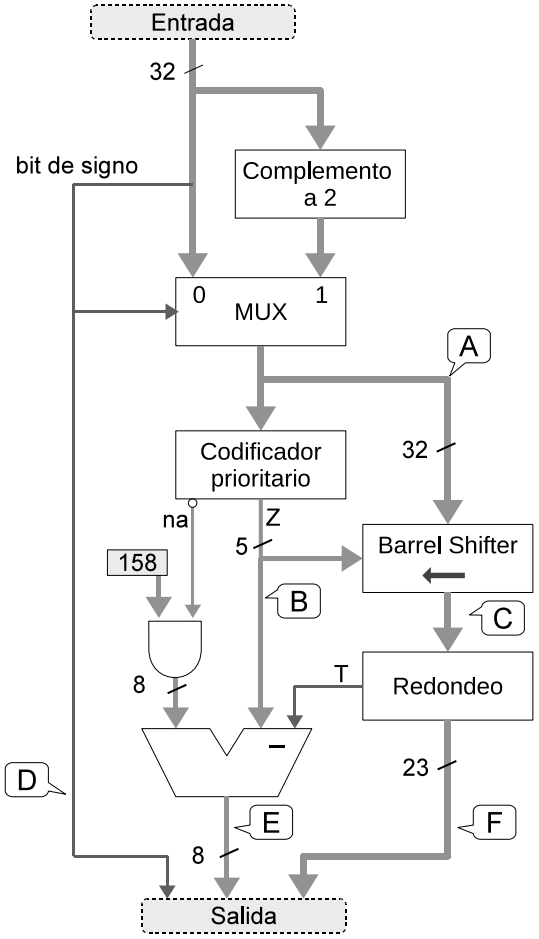

Figura 1: Estructura del conversor de *word* a *float*

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

2. Razone sobre qué modificaciones habría que hacer para adaptar este operador para que genere resultados en doble precisión (exponente de 11 bits en exceso 1023 y mantisa de 52 bits): El cambio afectaria al multiplexor, al circuito de desplazamiento o al codificador prioritario? I al restador? Haría falta un circuito de redondeo?

3. Calcule en MOPS la máxima productividad del operador de la figura 1.

SOLUCIÓN

1. Tabla

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. El operador equivalente en doble precisión mantendría todos los componentes del de la figura 1 excepto que:

   - No haría falta circuito de redondeo, porque el espacio de la mantisa en doble precisión es más grande que los 32 bits de la entrada.

- El restador tendría que operar con 11 bits y hacer la operación $1055 - Z$

3. El camino crítico del operador es aquél que pasa por el circuito de complemento a 2, el multiplexor, el codificador prioritario, el *barrel shifter*, el circuito de redondeo y el restador final. Sumando los retardos, tenemos un retardo total de 12 ns y una productividad de 83,3 MOPS.

■
