

*Fonaments de Computadors*

# Pràctica llenguatge màquina i assemblador: **temperatures en coma fixa**

Professors responsables: Pere Millán i Santiago Romaní

Departament d'Enginyeria Informàtica i Matemàtiques (DEIM)

*Universitat Rovira i Virgili (URV)*

(Revisió: 21, 11/abril/2021)

## Taula de continguts

0. Objectius i Resum	4
1. Conceptes previs	6
1.1. Codificació en coma fixa	6
1.1.1. Codificació de números reals (amb decimals) en coma fixa	6
1.1.2. Rang de representació en coma fixa	10
1.1.3. Codificació en coma fixa 1:17:14 (Q14) a la pràctica d'ARM	12
1.2. Operacions aritmètiques en coma fixa	14
1.2.1. Suma en coma fixa	14
1.2.2. Resta en coma fixa	15
1.2.3. Producte (multiplicació) en coma fixa	15
1.2.4. Divisió en coma fixa	16
2. Fase 1: Conversió de temperatures en coma fixa	18
2.1. Relació (fórmula) entre graus Celsius i Fahrenheit	18
2.2. Estructura de directoris i fitxers	19
2.3. Treball a realitzar	20
3. Fase 2: Accés a taules i multicamps de temperatures	25
3.1. Introducció	25
3.2. Estructura de directoris i fitxers	25
3.3. Treball a realitzar	26
4. Fase 3: Rutines aritmètiques en coma fixa	35
4.1. Estructura de fitxers i directoris	35
4.2. Rutina de suma en coma fixa	36
4.3. Rutina de resta en coma fixa	37
4.4. Rutina de producte (multiplicació) en coma fixa	38
4.5. Rutina de divisió en coma fixa	39
4.6. Treball a realitzar	41

5. Organització, lliurament i avaluació	45
5.1. Treball en equip	45
5.2. Lliurament i Terminis	45
5.3. Valoració del treball (avaluació)	46
5.4. Explicació de la pràctica en vídeo	47
6. Sobre la programació en assemblador GAS	48
6.1. Criteris d'interfície entre llenguatge C i assemblador GAS	48
6.2. Desplaçament de valors de 64 bits (amb operacions de 32 bits)	50
6.3. Variables locals a memòria en assemblador GAS	54

## 0. Objectius i Resum

Aquesta pràctica està dissenyada per cobrir els següents conceptes i metodologies:

- Codificació i operació de números reals en coma fixa.
- Accés a vectors, matrius i multicamps<sup>1</sup> (estructures).
- Programació en llenguatge assemblador.
- Traducció de codi en alt nivell a llenguatge assemblador.
- Desenvolupament i aplicació de jocs de proves.
- Ús d'eines per a la creació, execució i depuració de codi ARM.

La temàtica de la pràctica d'ARM d'aquest curs és la gestió de valors de temperatura (en graus Celsius i Fahrenheit), codificats com a números reals en coma fixa, emmagatzemats en variables tipus taula i multicamp (registre/*struct*), i com operar (suma, resta, producte i divisió) usant aquesta codificació. La pràctica està organitzada en 3 fases incrementals (el codi d'una fase, es basa en el codi de la fase anterior) i tractarem cada fase en una sessió de laboratori.

Es proporcionen codis en C (que caldrà traduir a assemblador) i jocs de proves per verificar que les routines desenvolupades funcionen correctament. Caldrà utilitzar les eines i tècniques que s'han presentat als laboratoris anteriors.

Abans de començar a escriure codi, cal tenir un mínim coneixement del tipus d'informació amb què es treballarà. A l'apartat “1.1. Codificació en coma fixa” es fa un repàs de la codificació de números reals en coma fixa. L'apartat “1.2. Operacions aritmètiques en coma fixa” indica com realitzar les 4 operacions aritmètiques bàsiques i com detectar resultats fora de rang (*overflow*).

La primera fase de la pràctica es descriu a l'apartat “2. Fase 1: Conversió de temperatures en coma fixa” i consisteix en traduir a assemblador dues routines en C de conversió de temperatures de graus Centígrads a graus Fahrenheit i a l'inrevés.

A la segona fase de la pràctica, descrita a l'apartat “3. Fase 2: Accés a taules i multicamps de temperatures” es treballa temperatures de diferents llocs de la Terra i dels 12 mesos de l'any, emmagatzemades en matrius (amb algunes temperatures en graus Celsius i altres en graus Fahrenheit), i sobre les què caldrà

---

<sup>1</sup> Utilitzarem el terme “**multicamp**” per referir-nos a tipus, variables i valors que estan formats per varis camps de dades, cada camp pot ser de tipus diferents, i l'accés en alt nivell es realitza pel nom del camp. En llenguatge C, els multicamps es declaren amb la paraula reservada **struct**.

localitzar temperatures màximes i mínimes, així com calcular temperatures mitjanes, desant el resultat en una variable multicamp.

La tercera i darrera fase de la pràctica es descriu a l'apartat “4. Fase 3: Rutines aritmètiques en coma fixa” i consisteix en traduir a llenguatge assemblador el codi proporcionat en llenguatge C de les 4 rutines que implementen la suma, resta, producte i divisió de números codificats en coma fixa, més la detecció d'*overflow*. Posteriorment cal adaptar el codi de les fases anteriors per utilitzar les rutines de coma fixa desenvolupades. Les rutines aritmètiques han de superar el joc de proves proporcionat (que es pot ampliar per provar altres casos).

L'apartat “5. Organització, lliurament i avaluació” conté la resta de detalls “acadèmics” sobre la pràctica.

Finalment, a l'apartat “6. Sobre la programació en assemblador GAS” es recullen algunes informacions i tècniques que poden ser d'utilitat per implementar la pràctica.

Els fitxers necessaris per desenvolupar la pràctica es troben al fitxer [PracticaARM.zip](#), que cal descomprimir a `/c/[siglesGrauURV]/FonCompus/` i estan organitzats en els següents directoris:

```
/c/.../FonCompus/PracticaARM/
    +-- 0_CelsiusFahrenheit_C/
    +-- 1_CelsiusFahrenheit_GAS/
    +-- 2_GeoTemp/
    +-- 3_LibQ14/
```

Els 2 primers directoris `CelsiusFahrenheit` s'utilitzen a la fase 1, `2_GeoTemp/` és per a la fase 2 i `3_LibQ14/` per a la fase 3.

# 1. Conceptes previs

Abans d'explicar els detalls de la pràctica, és important conèixer i entendre com es codificarà la informació en coma fixa i com es poden realitzar les 4 operacions aritmètiques bàsiques.

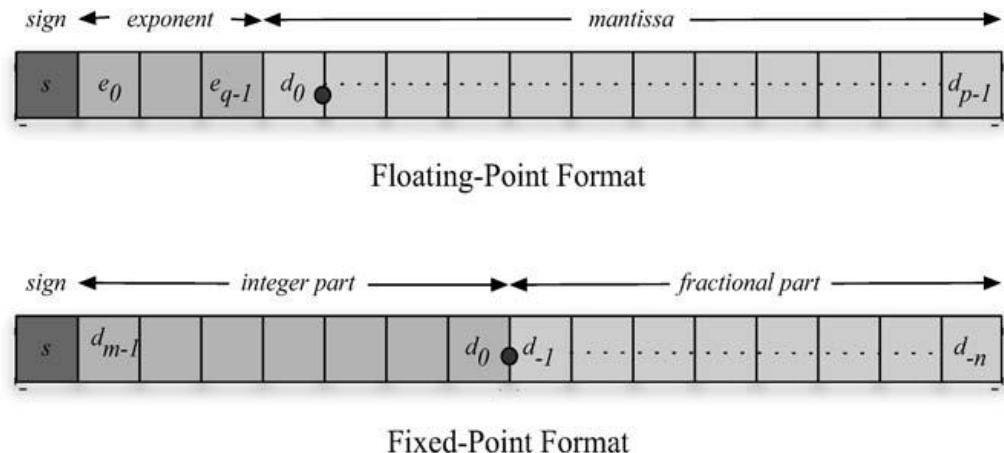
## 1.1. Codificació en coma fixa

En aquest apartat es fa un repàs de la codificació de números reals en coma fixa, com aplicar-la, rang de representació, possibles errors, ...

### 1.1.1. Codificació de números reals (amb decimals) en coma fixa

Al Tema 1 de Fonaments de Computadors hem vist com els ordinadors codifiquen diversos tipus d'informació (números, caràcters, símbols, etc.). En el cas dels números reals (els que tenen decimals), cal dedicar una certa quantitat de bits per codificar la part fraccionària (els decimals). Per codificar números reals es poden utilitzar principalment 2 formats:

- **Coma fixa:** per a la part fraccionària es reserva un cert número de bits **f** als bits baixos de la codificació, que corresponen a “pesos” de potències negatives de 2 (tal com els bits de la part entera corresponen a “pesos” de potències positives de 2).
- **Coma flotant:** s'utilitza una codificació similar a la “notació científica” ( $X, xx \cdot 10^{exp}$ ) amb un camp “exponent” per tenir un rang de representació més ampli (exponent negatiu per a valors molt petits i exponent positiu per a valors molt grans), i s'emmagatzemen només els díigits “més significatius” (mantissa).



De les diverses [notacions de Coma Fixa](#) que existeixen, nosaltres utilitzarem dues:

- **Qf**, sent **f** el número de bits de la part fraccionaria.
- **s:i:m**, sent **s** el bit de signe, **i** el número de bits de la part entera i **f** el número de bits de la part fraccionaria.

En aquesta pràctica utilitzarem **valors de 32 bits amb f=14 bits (Q14)** per als decimals. Això implica una codificació en format **Coma Fixa 1:17:14** (1 bit per al **signe**, 17 bits per a la part **entera** i 14 bits per a la part **fraccionària**).

Per interpretar quin valor representa un número real codificat en coma fixa amb **f** bits per a la part fraccionària, cal aplicar el “pes” de les potències de 2 (tal com es fa amb els números naturals i enters en complement a 2), però els **f** bits de la part fraccionària correspon a potències negatives.

En general, agafant la representació en binari **Codif** que codifiqui un valor **Real** utilitzant **f** bits per a la part fraccionaria (Qf), tenim la relació:

$$\text{Real} = \text{Codif} / 2^f$$

**Exemple 1.1.1:**

**Pregunta:** considerant una codificació en coma fixa  $0:5:3$  (Q3), quin valor real representa  $0b01001011$ ?

**Resposta 1:** en format  $0:5:3$  (Q3), tenim 5 bits de part entera i 3 bits de part fraccionària. Per tant, el “punt decimal” estaria ubicat com:  $0b01001.011$ . Els “pesos” de cadascun dels bits anteriors és:  $2^4 | 2^3 | 2^2 | 2^1 | 2^0 | 2^{-1} | 2^{-2} | 2^{-3}$ . Agafant només les potències de 2 dels bits a 1, tenim:  $2^3 + 2^0 + 2^{-2} + 2^{-3} = 8 + 1 + (1/4) + (1/8) = 8 + 1 + 0,25 + 0,125 = \mathbf{9,375}$ .

**Resposta 2:** la codificació  $0b01001011$ , si l’interpretem amb 3 bits fraccionaris (Q3), podem calcular el seu valor amb la fórmula anterior. Considerant valors enters,  $\text{Codif} = 0b01001011 = 75$ ; tenim  $2^f = 2^3 = 8$ . Per tant:  $\text{Real} = 75/8 = \mathbf{9,375}$ .

A partir de la fórmula anterior, podem obtenir un valor (enter) **Codif** que representa un valor **Real** (amb decimals) codificat en coma fixa amb **f** bits per a la part fraccionaria (**Qf**):

$$\text{Codif} = \text{Real} * 2^f$$

**Exemple 1.1.2:**

**Pregunta:** volem codificar el valor 9,375 en format de coma fixa amb  $f = 3$  bits (Q3). Quin valor binari obtindríem?

**Resposta:** per codificar el valor  $\text{Real} = 9,375$ , amb  $f = 3$  bits (Q3), apliquem la fórmula anterior i tenim:  $\text{Codif} = \text{Real} * 2^f = 9,375 * 2^3 = 9,375 * 8 = 75 = 0b01001011$ .

Si volem poder codificar com a mínim un cert número de díigits decimals (en base 10), quin valor de **f** hem d'utilitzar? Per a 1 dígit decimal, hem de poder representar com a mínim 10 valors diferents ( $f \geq 4$ ); per a 2 díigits decimals, calen 100 o més valors diferents ( $f \geq 7$ ), etc. A la següent taula es recullen els valors mínims de **f** segons el número de díigits decimals necessaris:

Decimals:	1	2	3	4	5	6	7	8	9	10
<b>f</b> mínim:	4	7	10	14	17	20	24	27	30	34

A la pràctica utilitzem  $f = 14$  (Q14) i, d'acord amb la taula anterior, podrem representar **4 díigits decimals** (de forma aproximada) i els 17 bits que queden per a la part entera, permetran representar 5 díigits en base 10.

La **codificació de números reals negatius** funciona de la mateixa forma que en el cas del Complement a 2 (el bit de més pes s'aplica amb valor negatiu). De fet, la codificació en Complement a 2 (Ca2) es podria considerar un “cas particular” de codificació en Coma fixa (amb  $f = 0$  bits, sense decimals, Q0).

### Exemple 1.1.3:

**Pregunta:** considerant una codificació en coma fixa 1:4:3 (Q3), quin valor real representa 0b11001011?

**Resposta 1:** en format 1:4:3, tenim 1 bit de signe, 4 bits de part entera i 3 bits de part fraccionària. Per tant, el “punt decimal” estaria ubicat com: 0b11001.**011**.

Els “pesos” de cadascun dels bits anteriors és:  $-(2^4) | 2^3 | 2^2 | 2^1 | 2^0 | 2^{-1} | 2^{-2} | 2^{-3}$ . Agafant només les potències de 2 dels bits a 1, tenim:  $-(2^4) + 2^3 + 2^0 + 2^{-2} + 2^{-3} = -16 + 8 + 1 + (1/4) + (1/8) = -16 + 8 + 1 + 0,25 + 0,125 = -6,625$ .

**Resposta 2:** considerant valors enters (Ca2), Codif = 0b11001011 = -53; tenim  $2^f = 2^3 = 8$ . Per tant: Real =  $-53/8 = -6,625$ .

### 1.1.2. Rang de representació en coma fixa

El rang de representació indica els valors màxims i mínims que es poden codificar (en funció del número de bits utilitzat: per exemple, amb 8 bits es poden codificar molts menys valors que amb 32 bits). De forma similar al que passa amb la codificació dels valors màxims/mínims en Complement a 2, tenim:

Valor	En binari <sup>2</sup>	$1:i:f$	<b>1:17:14 (Q14)</b>
mínim positiu (no 0)	0b0000...001	$1/2^f$	$1/2^{14} = 1/16384 = 0,00006103515625$
màxim positiu	0b0111...111	$2^i - 1/2^f$	131071,99993896484375
mínim negatiu (no 0)	0b1111...111	$-1/2^f$	$-1/2^{14} = -1/16384 = -0,00006103515625$
màxim negatiu	0b1000...000	$-2^i$	-131072,0000

Tal com passa també amb altres codificacions numèriques amb un número de bits concret (i limitat), tindrem errors d'**overflow (resultat massa gran)**, si el número a representar necessita més bits dels disponibles. En el nostre cas, això passarà quan calgui codificar un real amb **valor < -131072,0** o **valor > 131071,99993896484375**.

Amb la codificació de números reals, podem tenir un altre tipus d'error: **underflow (resultat massa petit)**, situació que es dona quan el valor a codificar és molt petit (proper a 0) i queda representat com a 0,0 (tot i que és lleugerament superior a 0). El valor més petit codifiable amb **f** bits per a la part fraccionària és  $\pm 1/2^f$  i qualsevol valor més petit queda representat com a 0. A la pràctica NO cal detectar l'*underflow*, però es pot donar amb productes i divisions, si el resultat és 0 i cap dels operands és 0 (producte) o el numerador no és 0 (divisió).

Un tercer tipus d'error que podem trobar en la codificació de números reals, són els **errors d'inexactitud**: un valor real que en base 10 té un número limitat de decimals<sup>3</sup>, no es pot codificar exactament en base 2, i llavors tenim un valor “aproximat” en comptes del valor “exacte”. En tot cas, l’error (diferència entre el valor real i el valor codificat) sempre serà inferior a  $1/2^f$ . Amb operacions en varis passos, aquest error es pot anar incrementant en cada nova operació i, per verificar el resultat final, pot ser necessari utilitzar un marge d’error ( $\varepsilon$ ) més gran que  $1/2^f$ .

---

<sup>2</sup> El símbol ... indica que els bits que falten tenen el mateix valor que els 3 bits anteriors i 2 posteriors als tres punts.

<sup>3</sup> També conegut com a “número racional”.

**Exemple 1.2.1:**

**Pregunta:** quins decimals entre 0,1 i 0,9 es poden representar de forma exacta amb  $f = 10$  bits per a la part fraccionària (Q10)?

**Resposta:** amb  $f = 10$  bits (Q10), s'hauria de poder representar sense problemes fins a 3 decimals (en base 10). Veiem si és així a la següent taula:

Valor real	Codificació (0:0:10)	Valor codificat	Valor exacte?
0,1	0b0001100110	0,099609375	✗
0,2	0b0011001100	0,19921875	✗
0,3	0b0100110011	0,2998046875	✗
0,4	0b0110011001	0,3994140625	✗
0,5	0b1000000000	0,5	✓
0,6	0b1001100110	0,599609375	✗
0,7	0b1011001100	0,69921875	✗
0,8	0b1100110011	0,7998046875	✗
0,9	0b1110011001	0,8994140625	✗

Només es pot codificar de forma exacta 1 dels 9 valors: 0,5 ( $1/2 = 2^{-1}$ ).

### 1.1.3. Codificació en coma fixa 1:17:14 (Q14) a la pràctica d'ARM

Per utilitzar la codificació en coma fixa 1:17:14 (Q14) a la pràctica d'ARM (en llenguatge C) es disposa del fitxer `include/Q14.h`. En primer lloc es defineix el tipus **Q14** (equivalent a `int`). Per tant, estem utilitzant codificacions de 32 bits, amb bit de signe:

```
Q14.h
14  /* Declaració de tipus Coma Fixa 1:17:14
15      | 1 bit de signe (bit 31), =0 -> positiu, =1 -> negatiu
16      | 17 bits de part entera (bits 30..14), aprox. 5 díigits en base 10
17      | 14 bits de part fraccionària (bits 13..0), aprox. 4 díigits en base 10
18  */
19  typedef int Q14;
20
```

En llenguatge assemblador, els valors en Coma Fixa 1:17:14 (Q14) caldrà tractar-los com es fa amb altres valors numèrics de 32 bits amb signe.

Per facilitar la codificació de valors en coma fixa en llenguatge C, es disposa de la “macro” **MAKE\_Q14** que converteix un valor `real` (amb decimals) a format Coma Fixa 1:17:14 (Q14). La codificació es calcula multiplicant el valor `real` per  $2^{14}$  (el valor  $2^{14}$ , 16384, s’aconsegueix desplaçant 14 bits a l’esquerra el valor 1); a continuació s’arrodoneix el valor (afegint  $\pm 0.5$ , segons si és positiu o negatiu) i finalment “truncant” el resultat, mitjançant un “cast” a enter (`int`):

```
Q14.h
22  /* Macro per convertir valors reals en valors Coma Fixa 1:17:14
23      | Exemples:
24      |     MAKE_Q14(3.1416) --> 0x0000C910
25      |     MAKE_Q14(-5.125) --> 0xFFFFEB800
26  */
27  #define MAKE_Q14(real)  (int)((real)*(1<<14)+(real<0?-0.5:0.5))
28
```

Per usar **MAKE\_Q14** (en C) cal indicar entre parèntesi el valor real que es vol codificar, com es fa a la fase 3 al fitxer `tests/test_Q14.c` quan es defineixen els casos del joc de proves de les operacions aritmètiques:

```
test_Q14.c
32  {add, MAKE_Q14(1.125),                                /* 3: 1.125 + -2.5 = -1.375
33      | MAKE_Q14(-2.5),
34      | MAKE_Q14(-1.375), 0},
```

Per facilitar l'accés als 3 camps de bits de la codificació en Coma Fixa 1:17:14 (signe, part entera, part fraccionària), al fitxer `include/Q14.h` es defineixen 3 “màscares” de 32 bits, que contenen bits a 1 a les posicions que corresponen als bits de cadascun dels 3 camps (i la resta de bits a 0):

```
Q14.h
30  /* MÀSCARES per als camps de bits de valors 1:17:14 */
31  #define MASK_SIGN  0x80000000 /* bit 31:    signe */
32  #define MASK_INT   0x7FFC000  /* bits 30..14: part entera */
33  #define MASK_FRAC  0x00003FFF /* bits 13..0:  part fraccionària */
34
```

També es disposa de la definició de les màscares en format GAS, al fitxer `include/Q14.i`:

```
Q14.i
10  @; MÀSCARES per als camps de bits de valors 1:17:14
11  MASK_SIGN = 0x80000000          @; bit 31:    signe
12  MASK_INT  = 0x7FFC000          @; bits 30..14: part entera
13  MASK_FRAC = 0x00003FFF        @; bits 13..0:  part fraccionària
14
```

## 1.2. Operacions aritmètiques en coma fixa

En aquest apartat es comenta com es poden calcular les operacions bàsiques aritmètiques (suma, resta, producte, divisió) amb operands i resultat codificats en coma fixa, així com la detecció de resultats fora del rang de representació (*overflow*).

Per entendre bé com funcionen les operacions en Coma Fixa (molt semblants, a les operacions en Ca2) i el motiu d'alguns “ajustos de bits” (al producte i la divisió) cal recordar la relació entre el valor real i el valor codificat:

$$\text{Real} = \text{Codif} / 2^f$$

$$\text{Codif} = \text{Real} * 2^f$$

Els resultats de les operacions aritmètiques, perquè “respectin” la codificació en coma fixa Qf, han d'incloure el “factor”  $2^f$  i, si no l'inclouen, caldrà afegir-ho a l'hora de calcular el resultat de l'operació (com passa amb el producte i la divisió).

### 1.2.1. Suma en coma fixa

Per sumar 2 números reals codificats en coma fixa, apliquem la següent fórmula:

$$\begin{aligned}\text{Codif}_{\text{result}} &= \text{Real}_{\text{result}} \times 2^f = (\text{Real}_1 + \text{Real}_2) \times 2^f = \\ &= \left( \frac{\text{Codif}_1}{2^f} + \frac{\text{Codif}_2}{2^f} \right) \times 2^f = \left( \frac{\text{Codif}_1 + \text{Codif}_2}{2^f} \right) \times 2^f = \\ &= \text{Codif}_1 + \text{Codif}_2\end{aligned}$$

Per tant, per obtenir la codificació en coma fixa de la suma de dos valors reals només cal sumar les codificacions en coma fixa d'aquests valors, com si fossin valors enters en Ca2.

Per detectar resultats fora de rang (*overflow*) en el resultat de la suma, es fa exactament igual que en Ca2: quan sumem 2 valors del mateix signe, el signe del resultat ha de ser el mateix que el dels operands. Si els números sumats són de signe diferent, mai es produirà *overflow*, perquè el número positiu es compensa amb el número negatiu, o sigui que el resultat no podrà sortir-se de rang.

### 1.2.2. Resta en coma fixa

Per restar 2 números reals codificats en coma fixa, apliquem la següent fórmula:

$$\begin{aligned}\text{Codif}_{\text{result}} &= \text{Real}_{\text{result}} \times 2^f = (\text{Real}_1 - \text{Real}_2) \times 2^f = \\ &= \left( \frac{\text{Codif}_1}{2^f} - \frac{\text{Codif}_2}{2^f} \right) \times 2^f = \left( \frac{\text{Codif}_1 - \text{Codif}_2}{2^f} \right) \times 2^f = \\ &= \text{Codif}_1 - \text{Codif}_2\end{aligned}$$

Per tant, per obtenir la codificació en coma fixa de la resta de dos valors reals només cal restar les codificacions en coma fixa d'aquests valors, com si fossin valors enters en Ca2.

Per detectar resultats fora de rang (*overflow*) en el resultat de la resta, es fa exactament igual que en Ca2: quan restem 2 valors de signe diferent, el signe del resultat ha de ser el mateix que el del primer operand. Si els números restats són del mateix signe, llavors la resta canviarà el signe del segon operand, de manera que mai es podrà produir *overflow* per la mateixa raó indicada en el cas de la suma de dos números de signe diferent.

### 1.2.3. Producte (multiplicació) en coma fixa

Per multiplicar 2 números reals codificats en coma fixa, apliquem la següent fórmula:

$$\begin{aligned}\text{Codif}_{\text{result}} &= \text{Real}_{\text{result}} \times 2^f = (\text{Real}_1 \times \text{Real}_2) \times 2^f = \\ &= \left( \frac{\text{Codif}_1}{2^f} \times \frac{\text{Codif}_2}{2^f} \right) \times 2^f = \frac{\text{Codif}_1 \times \text{Codif}_2}{2^f \times 2^f} \times 2^f = \\ &= \text{Codif}_1 \times \text{Codif}_2 / 2^f\end{aligned}$$

Per tant, per obtenir la codificació en coma fixa de la multiplicació de dos valors reals cal multiplicar les codificacions en coma fixa d'aquests valors, però després cal dividir entre  $2^f$  per obtenir el resultat codificat correctament.

També cal tenir en compte que, al realitzar el producte de les dues codificacions ( $\text{Codif}_1 \times \text{Codif}_2$ ), el resultat pot superar fàcilment el número de bits de la codificació original, o sigui que serà necessari utilitzar el doble de bits per allotjar el resultat d'aquest producte. La divisió entre  $2^f$  que es realitza després permetrà reduir el número de bits del resultat. Per exemple, a la pràctica treballem

amb valors de 32 bits, però el producte caldrà enregistrar-lo en 64 bits. Després de la divisió entre  $2^f$  (desplaçament aritmètic de f bits a la dreta) hauríem d'obtenir un resultat final que càpiga en 32 bits una altra vegada.

Això no obstant, és possible que el resultat final superi el límit dels 32 bits, o sigui, que surti del rang de la codificació en coma fixa (*overflow*). Per comprovar que no s'hagi produït *overflow*, caldrà detectar si els bits a l'esquerra del bit de signe tenen el mateix valor que el propi bit de signe. Per exemple, treballant amb 32+32 bits, si el bit de signe (bit 31) és 0 (positiu), llavors tots els bits 32-63 han de ser zero; de forma equivalent, si el bit de signe és 1 (negatiu), llavors tots els bits 32-63 també han de ser 1. En cas que hi hagi algun bit amb valor diferent de l'esperat, significarà que s'ha produït *overflow*.

#### 1.2.4. Divisió en coma fixa

Per dividir 2 números reals codificats en coma fixa, apliquem la següent fórmula:

$$\begin{aligned} \text{Codif}_{\text{result}} &= \text{Real}_{\text{result}} \times 2^f = \frac{\text{Real}_1}{\text{Real}_2} \times 2^f = \\ &= \frac{\frac{\text{Codif}_1}{2^f}}{\frac{\text{Codif}_2}{2^f}} \times 2^f = \frac{\text{Codif}_1}{\text{Codif}_2} \times \frac{2^f}{2^f} \times 2^f = \\ &= \text{Codif}_1 / \text{Codif}_2 \times 2^f \end{aligned}$$

Per tant, per obtenir la codificació en coma fixa de la divisió de dos valors reals cal dividir les codificacions en coma fixa d'aquests valors, però després cal multiplicar per  $2^f$  per obtenir el resultat codificat correctament.

Per fer les operacions anteriors conservant el màxim de precisió en el resultat final, és fa necessari aplicar la multiplicació del factor  $\times 2^f$  a  $\text{Codif}_1$  abans de fer la divisió:

$$\text{Codif}_{\text{result}} = (\text{Codif}_1 \times 2^f) / \text{Codif}_2$$

Per poder representar  $(\text{Codif}_1 \times 2^f)$  caldrà utilitzar f bits més que el número de bits original de la codificació en coma fixa. Després, la divisió també s'haurà d'adaptar al número de bits estès del dividend. Per exemple, es poden trobar implementacions de la divisió on el dividend utilitza el doble de bits que el divisor, amb un resultat (quotient) que també necessita el doble de bits.

Això no obstant, si el quocient utilitzés més bits de la codificació original en coma fixa, caldria considerar-ho com una situació d'*overflow*, que es pot detectar com en el cas de la multiplicació: si algun bit a l'esquerra del bit de signe del quocient té un valor diferent al del propi bit de signe, llavors hi ha *overflow*.

Un càlcul alternatiu que no requereix la divisió amb excés de bits és multiplicar el primer operand per l'invers del segon operand multiplicat per  $2^f$ . La fórmula quedaria com:

$$\text{Codif}_{\text{result}} = \text{Codif}_1 \times (2^f / \text{Codif}_2)$$

En aquesta segona fórmula, la divisió  $(2^f / \text{Codif}_2)$  es pot efectuar amb el mateix número de bits al dividend que al divisor.

Per detectar si el resultat final està fora de rang (*overflow*) es fa exactament igual que en el cas de la multiplicació, doncs la darrera operació de la fórmula és el producte, que pot generar valors que necessitin el doble de bits. De nou, caldrà detectar si algun bit a l'esquerra del bit de signe té un valor diferent al propi bit de signe.

Un cas particular de la divisió en coma fixa és quan volem **dividir un número real entre un número enter**. En aquest cas la fórmula queda com:

$$\begin{aligned} \text{Codif}_{\text{result}} &= \text{Real}_{\text{result}} \times 2^f = (\text{Real}_1 / \text{Enter}_2) \times 2^f = \\ &= (\text{Real}_1 \times 2^f) / \text{Enter}_2 = \text{Codif}_1 / \text{Enter}_2 \end{aligned}$$

És a dir, si volem dividir un número codificat en coma fixa entre un valor enter, es pot dividir directament el valor codificat del dividend entre el valor enter del divisor.

## 2. Fase 1: Conversió de temperatures en coma fixa

En aquest apartat es detallen les tasques a realitzar a la primera fase de la pràctica d'ARM. Es treballarà amb conversió de temperatures entre graus Celsius i graus Fahrenheit codificades en coma fixa. En primer lloc es repassa la fórmula que relaciona les dues unitats de temperatura. Després es comenta l'estrucció de fitxers i directoris ja disponibles amb el codi en C. Finalment es detalla el treball a realitzar: traducció a assemblador de les dues funcions de conversió de temperatures, programa principal i verificació amb el joc de proves proporcionat.

## 2.1. Relació (fórmula) entre graus Celsius i Fahrenheit

Per convertir temperatures entre graus Celsius ( $TempC$ ) i graus Fahrenheit ( $TempF$ ), apliquem la següent fórmula:

$$\frac{TempC}{5} = \frac{TempF - 32}{9}$$

Aillant  $\text{TempC}$  i  $\text{TempF}$  a la fórmula anterior, obtenim les dues fórmules de conversió entre graus Celsius i graus Fahrenheit:

$$TempC = (TempF - 32) \times \frac{5}{9}$$

$$TempF = TempC \times \frac{9}{5} + 32$$

La implementació de les dues fòrmules de conversió es farà en dues funcions declarades a `include/CelsiusFarenheit.h`:

```
CelsiusFahrenheit.h
13 #include "Q14.h"
14
15 /* Celsius2Fahrenheit(): converteix una temperatura en graus Celsius a la
16    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
17    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
18    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
19    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
19    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
18    output = (input * 9/5) + 32.0;
19 */
20 extern Q14 Celsius2Fahrenheit(Q14 input);  I
21
22 /* Fahrenheit2Celsius(): converteix una temperatura en graus Fahrenheit a la
23    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
24    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
24    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
25    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
25    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
25    output = (input - 32.0) * 5/9;
26 */
27 extern Q14 Fahrenheit2Celsius(Q14 input);
28
```

Totes dues rutines de conversió reben com a entrada la temperatura a convertir en format Q14 (coma fixa 1:17:14) i retornen com a resultat la temperatura en l'altra escala (codificada també en format Q14).

## 2.2. Estructura de directoris i fitxers

Els projectes `0_CelsiusFahrenheit_C` i `1_CelsiusFahrenheit_GAS` són els que utilitzarem per a la fase 1 de la pràctica. El primer conté codi en C amb tota la funcionalitat implementada, i el segon és l'estructura de directoris i fitxers que cal completar amb codi equivalent del 1r projecte però en assemblador GAS. L'estructura de directoris i fitxers d'aquests 2 projectes és la següent:

```
/c/.../FonCompus/PracticaARM/
    --- 0_CelsiusFahrenheit_C/
        |           +- build/
        |           +- include/
        |               |           +- CelsiusFahrenheit.h
        |               |           +- Q14.h
        |           +- p_lib/
        |               |           +- startup.o
        |           +- source/
        |               |           +- CelsiusFahrenheit.c
        |               |           +- demo.c
        |           +- tests/
        |               |           +- test_CelsiusFahrenheit.c
        |           +- 0_CelsiusFahrenheit_C.pnproj
        |           +- Makefile

    --- 1_CelsiusFahrenheit_GAS/
        +- build/
        +- include/
            |           +- CelsiusFahrenheit.h
            |           +- Q14.h
            |           +- Q14.i
        +- p_lib/
            |           +- startup.o
        +- source/
            |           +- CelsiusFahrenheit.s
            |           +- demo.s
        +- tests/
            |           +- test_CelsiusFahrenheit.c
        +- 1_CelsiusFahrenheit_GAS.pnproj
        +- Makefile
```

### 2.3. Treball a realitzar

A la primera fase de la pràctica cal traduir a llenguatge assemblador el codi en C que implementa les rutines de conversió de temperatures, així com un programa “demo” que utilitza les rutines. El codi en C de les rutines es troba al fitxer `source/CelsiusFahrenheit.c` del projecte `0_CelsiusFahrenheit_C`:

Al codi de les rutines s'utilitza `MAKE_Q14` per codificar els valors numèrics constants, les sumes i restes es fan directament entre valors de tipus `Q14`, els productes es fan amb resultat de 64 bits (tipus `long long`) i la divisió entre  $2^{14}$  s'aconsegueix desplaçant 14 bits a la dreta (línies 32 i 58).

El codi en C que crida les rutines es troba al fitxer `source/demo.c` del projecte `0_CelsiusFahrenheit_C`. Els paràmetres i els resultats de les crides a les dues funcions estan definits com a variables globals. El `main()` només fa una crida a cada rutina:

```
demo.c
19  /* global variables as inputs and output */
20  Q14 temp1C = MAKE_Q14(35.21);           // sample Celsius temperature
21  Q14 temp2F = MAKE_Q14(-23.75);         // sample Fahrenheit temperature
22
23  Q14 temp1F;    // expected conversion of temp1C:  95.378  °F
24  Q14 temp2C;    // expected conversion of temp2F: -30.97222  °C
25
26  int main(void)
27  {
28      temp1F = Celsius2Fahrenheit(temp1C);
29      temp2C = Fahrenheit2Celsius(temp2F);
30
31  /* TESTING POINT: check the results
32      (gdb) p temp1F
33      (gdb) p temp2C
34  */
35
36  /* BREAKPOINT */
37  return(0);
38 }
```

El programa `demo.c` és només un exemple de com cridar a les funcions de conversió de temperatures i emmagatzemar el resultat. Per aplicar un joc de proves complet es disposa del fitxer `tests/test_CelsiusFahrenheit.c` del projecte `0_CelsiusFahrenheit_C`. L'estructura del programa i casos de test segueix la mateixa mecànica utilitzada en sessions de laboratori anteriors:

```
test_CelsiusFahrenheit.c
16  typedef struct {
17      char scale;          /* letter of input scale ('C' or 'F') */
18      Q14 input;           /* input temperature value for conversion */
19      Q14 xresult;         /* Expected result of the operation */
20  } test_ops_struct;
21
22  /* the list of test case values */
23  test_ops_struct test_case[] =
24      /* Tests Celsius -> Fahrenheit */
25  {{'C', MAKE_Q14(0.0)},                  /* 0: 0.0 °C = 32.0 °F */
26      | MAKE_Q14(32.0)}},
```

Per a la traducció a assemblador, el fitxer `source/CelsiusFahrenheit.s` del projecte `1_CelsiusFahrenheit_GAS` conté la declaració de les dues rutines, però sense les instruccions que l'implementen (que és la tasca a realitzar a la primera fase):

```
CelsiusFahrenheit.s
1 @;-----
2 @; CelsiusFahrenheit.s: rutines de conversió de temperatura en
3 @;                                     format Q14 (Coma Fixa 1:17:14).
4 @;-----
5 @; santiago.romani@urv.cat
6 @; pere.millan@urv.cat
7 @; (Març 2021)
8 @;-----
9 @; Programador/a 1: xxx.xxx@estudiants.urv.cat
10 @; Programador/a 2: yyy.yyy@estudiants.urv.cat
11 @;-----*/
12
13 .include "include/Q14.i"
14
15
16 .text
17     .align 2
18     .arm
19
20
21 @; Celsius2Fahrenheit(): converteix una temperatura en graus Celsius a la
22 @;                                     temperatura equivalent en graus Fahrenheit, utilitzant
23 @;                                     valors codificats en Coma Fixa 1:17:14.
24 @; Entrada:
25 @;     input    -> R0
26 @; Sortida:
27 @;     R0      -> output = (input * 9/5) + 32.0;
28 .global Celsius2Fahrenheit
29 Celsius2Fahrenheit:
30     push {lr}
31
32
33     pop {pc}
34
35
36 @; Fahrenheit2Celsius(): converteix una temperatura en graus Fahrenheit a la
37 @;                                     temperatura equivalent en graus Celsius, utilitzant
38 @;                                     valors codificats en Coma Fixa 1:17:14.
39 @; Entrada:
40 @;     input    -> R0
41 @; Sortida:
42 @;     R0      -> output = (input - 32.0) * 5/9;
43 .global Fahrenheit2Celsius
44 Fahrenheit2Celsius:
45     push {lr}
46
47
48     pop {pc}
49
50
```

Les rutines només contenen les instruccions `push` i `pop` que permeten retornar sense fer res. A més d'incloure les instruccions que implementen la funcionalitat requerida, cal guardar a la pila (`push`) i recuperar al final (`pop`) els registres de llenguatge màquina utilitzats per cada rutina i que no hagin de quedar modificats.

La inclusió del fitxer `include/Q14.i` permet utilitzar en llenguatge assemblador les màscars definides, però no hi ha definida cap “macro” `MAKE_Q14`, de forma que la codificació de valors `Q14` en assemblador caldrà realitzar-la “manualment” (amb calculadora), segons s’indica a l’apartat “1.1.1. Codificació de números reals (amb decimals) en coma fixa”. Pel que fa al desplaçament de valors de 64 bits, caldrà seguir les indicacions de l’apartat “6.2. Desplaçament de valors de 64 bits (amb operacions de 32 bits)”.

Als fitxers que han de contenir el codi desenvolupat per estudiants, cal identificar qui ha fet la programació d’aquell fitxer, indicant els corresponents correus d’estudiant URV.

També cal traduir a assemblador, el fitxer `source/demo.s` del projecte `1_CelsiusFahrenheit_GAS`, que ha de fer una crida a cada funció de conversió. Els paràmetres d’entrada i els resultats de sortida, estan definits com a variables globals, just abans de la funció `main`:

```
demo.s | 1 @; -----
2 @; Description: a program to check the temperature-scale conversion
3 @; functions implemented in "CelsiusFahrenheit.c".
4 @; IMPORTANT NOTE: there is a much confident testing set implemented in
5 @; "tests/test_CelsiusFahrenheit.c"; the aim of "demo.c" is
6 @; to show how would it be a usual main() code invoking the
7 @; mentioned functions.
8 @; -----
9 @; Author: Santiago Romani (DEIM, URV)
10 @; Date: March/2021
11 @; -----
12 @; Programmer 1: xxx.xxx@estudiants.urv.cat
13 @; Programmer 2: yyy.yyy@estudiants.urv.cat
14 @; ----- */
15
16 .data
17     .align 2
18     temp1C: .word 0x0008CD71      @; temp1C = 35.21 °C
19     temp2F: .word 0xFFFFA1000      @; temp2F = -23.75 °F
20
21 .bss
22     .align 2
23     temp1F: .space 4            @; expected conversion: 95.378 °F
24     temp2C: .space 4            @; expected conversion: -30.97222 °C
25
```

```
27 .text
28     .align 2
29     .arm
30     .global main
31 main:
32     push {lr}
33
34     @; temp1F = Celsius2Fahrenheit(temp1C);
35
36     @; temp2C = Fahrenheit2Celsius(temp2F);
37
38 @; TESTING POINT: check the results
39 @; (gdb) p temp1F
40 @; (gdb) p temp2C
41 @; BREAKPOINT
42     mov r0, #0           @; return(0)
43
44     pop {pc}
45
46 .end
47
```

En aquest cas, els valors que s'han de passar com a paràmetres d'entrada de les funcions, ja estan inicialitzats a les variables globals corresponents. La comprovació del resultat caldrà fer-la “manualment” (amb ajuda de calculadora).

### 3. Fase 2: Accés a taules i multicamps de temperatures

En aquest apartat es detallen les tasques a realitzar a la 2a fase de la pràctica d'ARM. S'utilitzaran les rutines de conversió de temperatures de la fase 1 per realitzar càlculs de temperatures mitjanes, màximes, mínimes, treballant sobre taules (matrius) de temperatures de diverses ciutats del món (en °C o °F) durant els 12 mesos de l'any. Les rutines de la fase 1 s'utilitzaran per convertir els valors de temperatura a la mateixa escala (°C).

A l'apartat 3.2 es presenta l'estructura de fitxers i directoris disponibles amb el codi en C, així com el directori on cal posar el fitxer `CelsiusFahrenheit.o` amb les rutines de la fase 1 que s'usaran a la fase 2.

A l'apartat 3.3 es detalla el treball a realitzar: creació del fitxer `.s` amb la traducció de les 2 rutines de consulta de dades de temperatura (per ciutat i per mes de l'any), que caldrà verificar amb el joc de proves proporcionat. També s'haurà de crear i configurar el fitxer `Makefile` del projecte de la fase 2.

#### 3.1. Introducció

Aquesta fase 2 té com a objectiu practicar l'accés en assemblador a tipus de dades “taula” (matrius) i “multicamps” (`struct` en C), a més de continuar posant en pràctica la traducció a assemblador d'alguns dels codis en C que es proporcionen.

L'accés a matrius i a variables multicamp es tracta als darrers apartats de la secció 2.3 de Teoria, que convé repassar/refrescar per saber com implementar en assemblador el codi demanat. En essència, l'accés a un camp d'una variable multicamp consisteix en calcular el desplaçament del camp respecte a l'adreça de memòria inicial de la variable, i llavors accedir a 1, 2 o 4 bytes, considerats com a naturals o enters (si cal convertir el valor numèric de 8 o 16 bits a 32 bits).

#### 3.2. Estructura de directoris i fitxers

El projecte `2_GeoTemp` és el que utilitzarem per a la fase 2 de la pràctica. Els codis en C es proporcionen al directori `source/` on també caldrà afegir el fitxer `avgmaxmintemp.s` amb el codi assemblador traduït de la fase 2. El fitxer `Makefile` també s'ha d'afegir i configurar amb les dependències i regles de la fase 2. El fitxer `CelsiusFahrenheit.o` s'ha de copiar des del directori `build/`

de la fase 1, evidentment quan el codi de les rutines estigui acabat i verificat. L'estructura de directoris i fitxers és la següent:

```
/c/.../FonCompus/PracticaARM/
    +-+ 2_GeoTemp/
        +-+ build/
        +-+ include/
            |           +-+ avgmaxmintemp.h
            |           +-+ avgmaxmintemp.i
            |           +-+ CelsiusFahrenheit.h
            |           +-+ data.h
            |           +-+ divmod.h
            |           +-+ geotemp.h
            |           +-+ Q14.h
        +-+ p_lib/
            |           +-+ CelsiusFahrenheit.o
            |           +-+ libfoncompus.a
            |           +-+ startup.o
        +-+ source/
            |           +-+ avgmaxmintemp.c
            |           +-+ avgmaxmintemp.s
            |           +-+ data.c
            |           +-+ geotemp.c
        +-+ tests/
            |           +-+ test_geotemp.c
    +-+ 2_GeoTemp.pnproj
    +-+ Makefile
```

### 3.3. Treball a realitzar

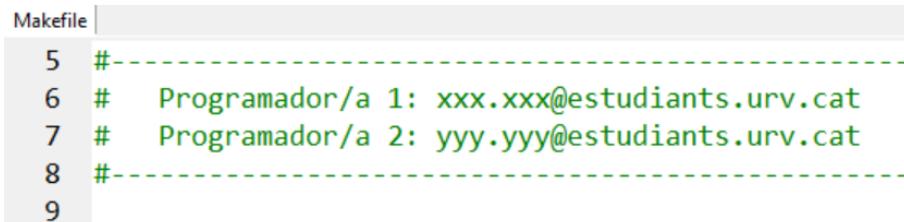
A la segona fase de la pràctica cal traduir a llenguatge assemblador el codi en C que implementa les rutines de càlcul de temperatures mitjana, màxima i mínima d'una ciutat (per a tots els mesos de l'any) o d'un mes (per a totes les ciutats disponibles).

Abans, però, com que en alguns punts del codi caldrà convertir entre unitats de temperatura (Celsius i Fahrenheit), cal incorporar al directori de la fase 2 `2_GeoTemp/p_lib/` el codi objecte de les rutines contingut al fitxer de la fase 1 `1_CelsiusFahrenheit_GAS/build/CelsiusFahrenheit.o`.

Aquest fitxer NO s'ha d'incorporar al directori `build/` de la fase 2, doncs s'esborraria quan es faci “`make clean`”. El seu lloc “natural” és al directori `p_lib/`, doncs és el directori creat per contenir fitxers objecte o llibreries (fitxers

.a) externes al projecte, és a dir que s'han compilat o assemblat en altres projectes.

Per poder compilar, assemblar, enllaçar, executar i depurar amb la comanda make, cal crear i configurar el fitxer **Makefile** (que no es proporciona). És imprescindible indicar els correus dels membres de l'equip que desenvolupa la pràctica, quelcom semblant a:



```

Makefile
5  #-
6  # Programador/a 1: xxx.xxx@estudiants.urv.cat
7  # Programador/a 2: yyy.yyy@estudiants.urv.cat
8  #-
9

```

El fitxer **Makefile** ha de tenir una estructura com la d'altres projectes que ja hem vist anteriorment (i a les sessions de laboratori). El que cal incloure a les diferents seccions és:

- A la secció “`make commands`” cal afegir les regles/dependències per generar els corresponents fitxers objectes “`.o`” (al directori `build/`) a partir dels fitxers del directori `source/`. A les dependències de fitxers de codi font, cal indicar, a més dels fitxers `.s` i `.c`, els fitxers `.h` i `.i` que corresponguin del directori `include/`. També cal afegir les regles/dependències per generar els fitxers executables `geotemp.elf` i `test_geotemp.elf`.
- A la secció “`clean commands`” del fitxer **Makefile** cal afegir les comandes d'esborrat de tots els fitxers intermedis del directori `build/` i tots els fitxers executables `.elf`.
- A “`run commands`” cal executar `geotemp.elf` dins del depurador Insight.
- I a “`debug commands`” cal executar `test_geotemp.elf` dins de l'Insight.

En cas de dubtes sobre la sintaxi de les dependències/regles del fitxer **Makefile**, pot ser útil revisar altres fitxers **Makefile** de sessions de laboratori anteriors.

Després de definir el fitxer **Makefile** ja es disposarà d'un entorn de creació, execució i depuració dels mòduls de la fase 2 i es pot començar amb la programació del codi font.

Les rutines que cal traduir de C a assemblador GAS estan declarades al fitxer `include/avgmaxmintemp.h`:

```
avgmaxmintemp.h
29 /* avgmaxmin_city(): calcula la temperatura mitja, màxima i mínima d'una
30 | ciutat d'una taula de temperatures, amb una fila per ciutat i
31 | una columna per mes, expressades en graus Celsius en format
32 | Q14.
33 | Paràmetres:
34 |     ttemp[][12] -> taula de temperatures, amb 12 columnes i nrows files
35 |     nrows        -> número de files de la taula
36 |     id_city      -> índex de la fila (ciutat) a processar
37 |     *mmres       -> adreça de l'estructura t_maxmin que retornarà els
38 |                       resultats de temperatures màximes i mínimes
39 |   Resultat:   temperatura mitja, expressada en graus Celsius, en format Q14.
40 */
41 Q14 avgmaxmin_city(Q14 ttemp[][12], unsigned short nrows, unsigned short id_city, t_maxmin *mmres);
42

44 /* avgmaxmin_month(): calcula la temperatura mitja, màxima i mínima d'un mes
45 | d'una taula de temperatures, amb una fila per ciutat i una
46 | columna per mes, expressades en graus Celsius en format Q14.
47 | Paràmetres:
48 |     ttemp[][12] -> taula de temperatures, amb 12 columnes i nrows files
49 |     nrows        -> número de files de la taula
50 |     id_month     -> índex de la columna (mes) a processar
51 |     *mmres       -> adreça de l'estructura t_maxmin que retornarà els
52 |                       resultats de temperatures màximes i mínimes
53 |   Resultat:   temperatura mitja, expressada en graus Celsius, en format Q14.
54 */
55 Q14 avgmaxmin_month(Q14 ttemp[][12], unsigned short nrows, unsigned short id_month, t_maxmin *mmres);
56
```

Ambdues rutines treballen sobre les dades de temperatura de diverses ciutats al llarg de 12 mesos (una temperatura per mes). Les temperatures estan emmagatzemades en una matriu de 12 columnes (una columna per cada mes de l'any) i tantes files com ciutats estiguin registrades (una fila per cada ciutat). El primer paràmetre de les rutines és la matriu amb les temperatures (pas per referència) i el segon paràmetre indica quantes ciutats (files) conté la matriu. El 4t paràmetre de les rutines és una variable multicamp `t_maxmin`, on s'haurà d'emmagatzemar (pas per referència) les temperatures màximes i mínimes (en °C i °F) així com les posicions (índex de fila o de columna) on es troben:

```
avgmaxmintemp.h
17     /* declaració del tipus t_maxmin, per retornar resultats */
18     typedef struct
19 {
20         Q14 tmin_C;           // temperatura mínima, en graus Celsius
21         Q14 tmax_C;           // temperatura màxima, en graus Celsius
22         Q14 tmin_F;           // temperatura mínima, en graus Fahrenheit
23         Q14 tmax_F;           // temperatura màxima, en graus Fahrenheit
24         unsigned short id_min; // índex temperatura mínima
25         unsigned short id_max; // índex temperatura màxima
26     } t_maxmin;
27 }
```

Ambdues rutines són funcions que retornen la temperatura mitjana de la ciutat o del mes indicats. La rutina `avgmaxmin_city()` rep com a 3r paràmetre l'índex de la ciutat (fila) que es vol analitzar, mentre que la segona rutina `avgmaxmin_month()` rep com a 3r paràmetre l'índex del mes (columna: 0-11) que es vol analitzar.

Per facilitar l'accés als camps de `t_maxmin` des d'assemblador, el fitxer `include/avgmaxmintemp.i` inclou definicions dels desplaçaments fins a l'inici dels diferents camps dels valors multicamp:

```
avgmaxmintemp.i
10     @; Estructura 't_maxmin' (veure fitxer 'avgmaxmintemp.h')
11 MM_TMINC = 0      @; Offset del camp temperatura mínima, en graus Celsius
12 MM_TMAXC = 4      @; Offset del camp temperatura màxima, en graus Celsius
13 MM_TMINF = 8      @; Offset del camp temperatura mínima, en graus Fahrenheit
14 MM_TMAXF = 12     @; Offset del camp temperatura màxima, en graus Fahrenheit
15 MM_IDMIN = 16     @; Offset del camp índex temperatura mínima
16 MM_IDMAX = 18     @; Offset del camp índex temperatura màxima
17 MM_SIZE  = 20     @; Número de bytes que ocupa l'estructura 't_maxmin'
18
```

Com a exemples de taules de temperatura mitjana mensual, el fitxer `source/data.c` defineix dues matrius amb la informació més recent de diferents ciutats del món (font: [Wikipedia](#)). Les ciutats s'han separat per hemisferi, ja que les estacions de l'any estan invertides (quan a un hemisferi és estiu, a l'altre és hivern), però això només és un exemple, es podrien crear noves taules amb agrupaments de ciutats per diferents criteris (altitud, vora mar o interior, per continent, país o regió, etc.).

La definició de les dades de `source/data.c` està recollida al fitxer `include/data.h`:

```
data.h
17 #define NUMCITIESHNORD 18
18 #define NUMCITIESHSUD 19
19
20 extern Q14 tempHNord_2020[NUMCITIESHNORD][12];
21 extern Q14 tempHSud_2020[NUMCITIESHSUD][12];
22
23 extern t_cityinfo info_HNord[NUMCITIESHNORD];
24 extern t_cityinfo info_HSud[NUMCITIESHSUD];
25
```

Les dades completes sobre temperatures mensuals de ciutats als hemisferis nord i sud estan declarades al fitxer `source/data.c`:

```
data.c |
18  Q14 tempHNord_2020[NUMCITIESHNORD][12] = {
19    {MAKE_Q14(13.4), MAKE_Q14(13.9), MAKE_Q14(15.7), MAKE_Q14(18.5),           // Alexandria
20    MAKE_Q14(21.2), MAKE_Q14(24.3), MAKE_Q14(25.9), MAKE_Q14(26.3),
21    MAKE_Q14(25.1), MAKE_Q14(22.0), MAKE_Q14(18.7), MAKE_Q14(14.9)},
22    {MAKE_Q14(2.2), MAKE_Q14(3.5), MAKE_Q14(5.8), MAKE_Q14(7.5),
23    MAKE_Q14(11.5), MAKE_Q14(15.4), MAKE_Q14(18.8), MAKE_Q14(18.5),
24    MAKE_Q14(14.9), MAKE_Q14(10.3), MAKE_Q14(5.7), MAKE_Q14(3.0)},           // Andorra la Vella
76  t_cityinfo info_HNord[NUMCITIESHNORD] = {
77    {"Alexandria", 'C'},               // Egypt
78    {"Andorra la Vella", 'C'},        // Andorra
79    {"Beijing", 'C'},                // China
80    {"Calgary", 'C'},                // Canada
81    {"Dikson", 'C'},                 // Russia
82    {"Djibouti", 'C'},               // Djibouti
83    {"George Town", 'F'},            // Cayman Islands
84    {"Kansas City", 'F'},             // USA
98  Q14 tempHSud_2020[NUMCITIESHSUD][12] = {
99    {MAKE_Q14(23.2), MAKE_Q14(23.3), MAKE_Q14(20.9), MAKE_Q14(17.6),           // Adelaide
100   MAKE_Q14(13.5), MAKE_Q14(12.1), MAKE_Q14(11.4), MAKE_Q14(12.4),
101   MAKE_Q14(14.4), MAKE_Q14(16.7), MAKE_Q14(19.7), MAKE_Q14(21.3)},
102   {MAKE_Q14(15.0), MAKE_Q14(14.8), MAKE_Q14(11.9), MAKE_Q14(7.9),           // Bariloche
103   MAKE_Q14(4.9), MAKE_Q14(2.9), MAKE_Q14(2.1), MAKE_Q14(3.0),
104   MAKE_Q14(5.1), MAKE_Q14(8.0), MAKE_Q14(10.8), MAKE_Q14(13.5)},
159 t_cityinfo info_HSud[NUMCITIESHSUD] = {
160   {"Adelaide", 'C'},              // Australia
161   {"Bariloche", 'C'},              // Argentina
162   {"Canberra", 'C'},              // Australia
163   {"Dili", 'C'},                  // East Timor
164   {"Fianarantsoa", 'C'},          // Madagascar
165   {"Ghanzi", 'C'},                // Botswana
166   {"Hanga Roa", 'C'},              // Chile
```

El multicamp `t_cityinfo` permet emmagatzemar el nom de la ciutat i les unitats de temperatura que s'utilitzen, i està declarada a `include/geotemp.h`:

```
geotemp.h |
12  /* declaració del tipus t_cityinfo */
13  typedef struct
14  {
15    char *name;                  // nom de la ciutat
16    char scale;                  // escala utilitzada ('C': Celsius, 'F': Fahrenheit)
17  } t_cityinfo;
18
```

El fitxer `source/geotemp.c` conté un possible programa principal que calcula resultats per a dues ciutats i dos mesos de cada hemisferi. Prèviament, es

crida a la funció `normalitzar_temperatures()`, que converteix les temperatures en Fahrenheit a Celsius, per tal de fer tots els càlculs en °C:

```
geotemp.c
50 int main(void)
51 {
52     Q14 avgres[4];
53     t_maxmin maxminres[4];
54
55     normalitzar_temperatures(info_HNord, tempHNord_2020, NUMCITIESHNORD);
56     normalitzar_temperatures(info_HSud, tempHSud_2020, NUMCITIESHSUD);
57
58         // càlculs sobre George Town (Cayman Island)
59     avgres[0] = avgmaxmin_city(tempHNord_2020, NUMCITIESHNORD, 6, &maxminres[0]);
60         // càlculs sobre el mes d'agost (hemisferi nord)
61     avgres[1] = avgmaxmin_month(tempHNord_2020, NUMCITIESHNORD, 7, &maxminres[1]);
62         // càlculs sobre Wellington (New Zealand)
63     avgres[2] = avgmaxmin_city(tempHSud_2020, NUMCITIESHSUD, 18, &maxminres[2]);
64         // càlculs sobre el mes de desembre (hemisferi sud)
65     avgres[3] = avgmaxmin_month(tempHSud_2020, NUMCITIESHSUD, 11, &maxminres[3]);
66 }
```

El fitxer `tests/test_geotemp.c` conté un possible joc de proves de les rutines a traduir a assemblador:

```
test_geotemp.c
13 #define NUM_TEST_ROWS 3
14
15 Q14 test_data[NUM_TEST_ROWS][12] = {
16     {MAKE_Q14(13.4), MAKE_Q14(13.4), MAKE_Q14(20.0), MAKE_Q14(13.4),      // several replicated min and
17     MAKE_Q14(20.0), MAKE_Q14(20.0), MAKE_Q14(25.9), MAKE_Q14(25.9),
18     MAKE_Q14(20.0), MAKE_Q14(20.0), MAKE_Q14(20.0), MAKE_Q14(25.9)},
19     {MAKE_Q14(-2.2), MAKE_Q14(-3.5), MAKE_Q14(-5.8), MAKE_Q14(-7.5),      // all negatives
20     MAKE_Q14(-11.5), MAKE_Q14(-15.4), MAKE_Q14(-18.8), MAKE_Q14(-18.5),
21     MAKE_Q14(-14.9), MAKE_Q14(-10.3), MAKE_Q14(-5.7), MAKE_Q14(-3.0)},
22     {MAKE_Q14(0.1), MAKE_Q14(0.3), MAKE_Q14(0.7), MAKE_Q14(0.8),          // all values around 0° Celsius
23     MAKE_Q14(0.8), MAKE_Q14(-0.9), MAKE_Q14(-0.7), MAKE_Q14(0.5),
24     MAKE_Q14(0.0), MAKE_Q14(0.7), MAKE_Q14(0.5), MAKE_Q14(-0.9)}};
25 };
26
27 /* type definition of the structured record that holds the test case values */
28 typedef struct {
29     unsigned char op; /* type of operation ('C': by city, 'M': by month) */
30     unsigned short id; /* index to be checked (city or month) */
31     Q14 xavg; /* expected average */
32     t_maxmin xmm; /* expected max-min results */
33 } test_struct;
34
35 /* the list of test case values */
36 test_struct test_case[] =
```

La tasca principal de la fase 2 és traduir de C a llenguatge assemblador les rutines `avgmaxmin_city()` i `avgmaxmin_month()`. El codi font s'haurà d'introduir al fitxer `source/avgmaxmintemp.s`, que s'haurà de crear expressament. Tot i ser dues rutines diferents, l'algorisme de càcul de les temperatures mitjanes, màxima i mínima és equivalent. En el cas d'una ciutat, cal recórrer una fila de la matriu, mentre que en el cas d'un mes de l'any cal recórrer una columna completa. Per tant, bastant codi de la traducció de les rutines serà pràcticament idèntic i es podrà

reaprofitar (o adaptar de forma senzilla). És més, es pot dissenyar una rutina auxiliar que pugui processar les dades tant per files com per columnes, si se li passa per paràmetre el desplaçament que s'ha de fer en memòria per a passar d'una posició de la matriu a la següent posició a tractar.

El codi en llenguatge C de les routines a traduir es troba al fitxer `source/avgmaxmintemp.c`:

```
avgmaxmintemp.c
17 /* avgmaxmin_city(): calcula la temperatura mitjana, màxima i mínima d'una
18 | ciutat d'una taula de temperatures, amb una fila per ciutat i
19 | una columna per mes, expressades en graus Celsius en format
20 | Q14.
21 | Paràmetres:
22 |     ttemp[][12] -> taula de temperatures, amb 12 columnes i nrows files
23 |     nrows      -> número de files de la taula
24 |     id_city    -> índex de la fila (ciutat) a processar
25 |     *mmres     -> adreça de l'estructura t_maxmin que retornarà els
26 |                   resultats de temperatures màximes i mínimes
27 |   Resultat:  temperatura mitjana, expressada en graus Celsius, en format Q14.
28 */
29 Q14 avgmaxmin_city(Q14 ttemp[][12], unsigned short nrows, unsigned short id_city, t_maxmin *mmres)
30 {
31     Q14 avg, max, min;
32     unsigned short idmin = 0, idmax = 0; // índexos de temp. mínima i màx.
33     unsigned short i;
34     Q14 tvar;                         // variable temporal de temperatura
35
36     avg = ttemp[id_city][0];          // inicialitza valors amb primera columna
37     max = avg;                      // (mes de gener)
38     min = avg;
39
40     for (i = 1; i < 12; i++)        // per a la resta de mesos
41     {
42         tvar = ttemp[id_city][i];      // obtenir temperatura del següent mes
43         avg += tvar;
44         if (tvar > max)             // actualitzar valors resultat
45         {
46             max = tvar;
47             idmax = i;
48         }
49         if (tvar < min)
50         {
51             min = tvar;
52             idmin = i;
53         }
54     avg /= 12;                      // ajustar valor mitjà
55
56     mmres->tmin_C = min;           // transferir mínim-màxim en Celsius
57     mmres->tmax_C = max;           // transferir mínim-màxim en Fahrenheit
58     mmres->tmin_F = Celsius2Fahrenheit(min);
59     mmres->tmax_F = Celsius2Fahrenheit(max);
60                                         // transferir índexos mínim i màxim
61     mmres->id_min = idmin;
62     mmres->id_max = idmax;
63
64     return(avg);
65 }
66
67 }
```

```

avgmaxmintemp.c
70  /* avgmaxmin_month(): calcula la temperatura mitjana, màxima i mínima d'un mes
71   d'una taula de temperatures, amb una fila per ciutat i una
72   columna per mes, expressades en graus Celsius en format Q14.
73   Paràmetres:
74     ttemp[][][12] -> taula de temperatures, amb 12 columnes i nrows files
75     nrows        -> número de files de la taula (mínim 1 fila)
76     id_month     -> índex de la columna (mes) a processar
77     *mmres       -> adreça de l'estructura t_maxmin que retornarà els
78                  resultats de temperatures màximes i mínimes
79   Resultat:  temperatura mitjana, expressada en graus Celsius, en format Q14.
80 */
81 Q14 avgmaxmin_month(Q14 ttemp[][][12], unsigned short nrows, unsigned short id_month, t_maxmin *mmres)
82 {
83     Q14 avg, max, min;
84     unsigned short idmin = 0, idmax = 0;      // índexos de temp. mínima i màx.
85     unsigned short i;
86     Q14 tvar;                                // variable temporal de temperatura
87     unsigned int mod;                         // variable per invocar div_mod()
88     unsigned char avgNeg;                     // booleà per indicar si mitjana negativa
89
90     avg = ttemp[0][id_month];                 // inicialitza valors amb primera fila
91     max = avg;                               // (primera ciutat de la taula)
92     min = avg;
93     i = 1;                                   // posicionar índex a la segona fila
94     while (i < nrows)                      // per a la resta de ciutats
95     {
96         tvar = ttemp[i][id_month];           // obtenir temperatura següent ciutat
97         avg += tvar;
98         if (tvar > max)                   // actualitzar valors resultat
99         {
100             max = tvar;
101             idmax = i;
102         }
103         if (tvar < min)
104         {
105             min = tvar;
106             idmin = i;
107         }
108     }
109     avgNeg = (avg < 0);                   // memoritza si valor acumulat negatiu
110     tvar = (avgNeg ? -avg : avg);          // tvar conté valor absolut de avg
111     div_mod(tvar, nrows, (unsigned int *)&avg, &mod); // calcular valor mitjà sobre avg
112     if (avgNeg) avg = -avg;                // canviar signe de valor mitjà
113
114     mmres->tmin_C = min;                 // transferir mínim-màxim en Celsius
115     mmres->tmax_C = max;                 // transferir mínim-màxim en Fahrenheit
116     mmres->tmin_F = Celsius2Fahrenheit(min);
117     mmres->tmax_F = Celsius2Fahrenheit(max);
118     mmres->id_min = idmin;               // transferir índexos mínim i màxim
119     mmres->id_max = idmax;
120
121     return(avg);
122 }
123
124
125

```

Per fer les divisions (al càcul de la temperatura mitjana), el codi en C usa l'operador de divisió (línia 54) i també crida la rutina `div_mod()` (línia 111). En llenguatge assemblador, però, caldrà usar `div_mod()` en els 2 casos.

A més, cal tenir en compte que `div_mod()` només permet divisions naturals: en cas que el dividend sigui negatiu, caldrà passar-lo a positiu i després el resultat (positiu) s'haurà de negar per obtenir el valor correcte (com es fa al codi en C a la segona divisió).

Com ja s'ha dit anteriorment, el codi de les rutines en assemblador s'ha d'introduir al fitxer **source/avgmaxmintemp.s**. A l'inici d'aquest fitxer cal indicar els correus electrònics dels components de l'equip, per exemple:

```
avgmaxmintemp.s
9 @;-----
10 @; Programador/a 1: xxx.xxx@estudiants.urv.cat
11 @; Programador/a 2: yyy.yyy@estudiants.urv.cat
12 @;-----
13
14 .include "include/avgmaxmintemp.i"
15
```

Com es pot veure, també cal incloure el fitxer **include/avgmaxmintemp.i** amb les declaracions per accedir al multicamp amb els resultats de les funcions.

Per cridar la rutina **div\_mod()**, cal passar com a 3r i 4t paràmetre les adreces de memòria de dues zones de 4 bytes cadascuna on la rutina desarà el quotient i residu que calculi. Per aconseguir-ho, es pot aplicar alguna de les tècniques descrites a l'apartat “6.3. Variables locals a memòria en assemblador GAS”.

Per verificar el correcte funcionament del codi assemblador de les rutines, es disposa del fitxer **tests/test\_geotemp.c** que aplica un joc de proves a les rutines. El joc de proves es pot ampliar amb nous casos. També es disposa del fitxer **source/geotemp.c** amb un possible programa principal que calcula resultats per a dues ciutats i dos mesos de cada hemisferi. Degut a possibles errors d'arrodoniment, els resultats poden ser lleugerament diferents als indicats. Per aquest motiu, el joc de proves verifica que la diferència entre el valor obtingut i el valor esperat no superin un determinat percentatge del valor obtingut (error relatiu).

## 4. Fase 3: Rutines aritmètiques en coma fixa

En aquesta secció es detalla una possible implementació en C de les quatre operacions aritmètiques (suma, resta, producte i divisió) amb números en Coma Fixa 1:17:14 (Q14). La tercera fase de la pràctica consisteix en desenvolupar rutines equivalents en assemblador GAS per a processadors ARM (caldrà crear el fitxer **Q14.s** amb el codi) i integrar-les dins d'una llibreria. Per provar les rutines, es proporciona un joc de proves amb diversos casos (que es poden ampliar).

Les 4 rutines aritmètiques reben per paràmetre els 2 valors en format Q14 a operar, així com l'adreça (pas per referència) d'un variable de mida 1 byte on la rutina indicarà si s'ha produït (1) o no (0) *overflow* en l'operació. Les 4 rutines són funcions que retornen el resultat de l'operació aritmètica.

En alguns casos es treballa amb valors de 64 bits (tipus **long long** en C), que en el llenguatge màquina que estem utilitzant caldrà implementar amb dos valors de 32 bits.

També serà necessari aplicar màscares per accedir a camps de bits concrets (per exemple, al signe, bit 31). Al campus virtual de Fonaments de Computadors es disposa del document [Com treballar amb màscares \(ARM i C\)](#), on s'explica amb més detall els fonaments de les màscares i les operacions més habituals.

### 4.1. Estructura de fitxers i directoris

El projecte **3.LibQ14** és el que utilitzarem per a la fase 3 de la pràctica. L'estructura de directoris i fitxers és la següent:

```
/c/.../FonCompus/PracticaARM/
    +-- 3.LibQ14/
        +-- build/
        +-- includes/
            |           +-- divmod.h
            |           +-- Q14.h
            |           +-- Q14.i
        +-- sources/
            |           +-- Q14.c
            |           +-- Q14.s
        +-- tests/
            |           +-- demo.c
            |           +-- libfoncompus.a
            |           +-- startup.o
            |           +-- test_Q14.c
    +-- 3.LibQ14.pnproj
    +-- Makefile
```

## 4.2. Rutina de suma en coma fixa

Tal com hem vist a l'apartat “1.2.1. Suma en coma fixa”, la suma de dos números codificats en coma fixa es fa sumant les seves codificacions. Per detectar si el resultat és massa gran (*overflow*), cal comprovar si sumant dos números del mateix signe, el resultat té signe diferent (si els dos números a sumar tenen signe diferent, mai es produeix *overflow*). Alternativament, quan es treballa en assemblador, es poden utilitzar instruccions de suma que actualitzen els *flags* (el *flag* que ens interessa és l'*overflow*) i comprovar si està actiu amb una instrucció de salt condicional segons el valor de l'*overflow*. La declaració en C de la funció és:

```
Q14.h
39  /* add_Q14(): calcula i retorna la suma dels 2 primers operands,
40    (num1 + num2) codificats en coma fixa 1:17:14.
41    El 3r paràmetre (per referència) retorna l'overflow:
42    0: no s'ha produït overflow, resultat correcte.
43    1: hi ha overflow (resultat massa gran) i el que es
44    retorna són els bits baixos del resultat.
45 */
46 extern Q14 add_Q14(Q14 num1, Q14 num2, unsigned char * overflow);
47
```

Una possible implementació en C de la funció `add_Q14()` és:

```
Q14.c
28 Q14 add_Q14(Q14 num1, Q14 num2, unsigned char * overflow)
29 {
30     Q14 suma;
31     unsigned char ov = 0; // inicialment assumeix que no hi ha overflow
32
33     suma = num1 + num2;
34
35     // Detecció overflow
36     if (((MASK_SIGN & num1) == (MASK_SIGN & num2))
37         && ((MASK_SIGN & num1) != (MASK_SIGN & suma)))
38         ov = 1;
39
40     *overflow = ov;
41     return(suma);
42 }
43
```

Es pot observar que la part que necessita més codi és la detecció de l'*overflow*, la qual aplica màscares per obtenir els signes dels operands i del resultat. En assemblador sí que es pot generar i consultar el *flag* d'*overflow* directament, evitant l'aplicació de màscares i les comparacions del codi en C.

### 4.3. Rutina de resta en coma fixa

Tal com hem vist a l'apartat “1.2.2. Resta en coma fixa”, la resta de dos números codificats en coma fixa es fa restant les seves codificacions. Per detectar si el resultat és massa gran (*overflow*), cal comprovar si restant dos números de signe diferent, el resultat té signe diferent al del primer operand (si els dos operands tenen el mateix signe, mai es produeix *overflow*). Alternativament, quan es treballa en assemblador, es poden utilitzar instruccions de resta que actualitzen els *flags* (el *flag* que ens interessa és l'*overflow*) i comprovar si està actiu amb una instrucció de salt condicional segons el valor de l'*overflow*. La declaració en C de la funció és:

```
Q14.h
49  /* sub_Q14(): calcula i retorna la diferència dels 2 primers operands,
50  (num1 - num2) codificats en coma fixa 1:17:14.
51  |-----|-----|-----|
52  |-----|-----|-----|
53  |-----|-----|-----|
54  |-----|-----|-----|
55  */
56  extern Q14 sub_Q14(Q14 num1, Q14 num2, unsigned char * overflow);
57
```

Una possible implementació en C de la funció `sub_Q14()` és:

```
Q14.c
52  Q14 sub_Q14(Q14 num1, Q14 num2, unsigned char * overflow)
53  {
54      Q14 resta;
55      unsigned char ov = 0; // inicialment assumeix que no hi ha overflow
56
57      resta = num1 - num2;
58
59      // Detecció overflow
60      if (((MASK_SIGN & num1) != (MASK_SIGN & num2))
61          && ((MASK_SIGN & num1) != (MASK_SIGN & resta)))
62          ov = 1;
63
64      *overflow = ov;
65      return(resta);
66  }
67
```

Iugal que amb la suma, es pot observar que la part que necessita més codi és la detecció de l'*overflow*. De nou, en assemblador podem generar i consultar el *flag* d'*overflow* directament.

## 4.4. Rutina de producte (multiplicació) en coma fixa

La declaració en C de la funció de multiplicació és:

```
Q14.h
59  /* mul_Q14(): calcula i retorna el producte dels 2 primers operands,
60  (num1 * num2) codificats en Coma fixa 1:17:14.
61  El 3r paràmetre (per referència) retorna l'overflow:
62  0: no s'ha produït overflow, resultat correcte.
63  1: hi ha overflow (resultat massa gran) i el que es
64  retorna són els bits baixos del resultat.
65  */
66  extern Q14 mul_Q14(Q14 num1, Q14 num2, unsigned char * overflow);
67
```

Una possible implementació en C de la funció `mul_Q14()` és:

```
Q14.c
76  Q14 mul_Q14(Q14 num1, Q14 num2, unsigned char * overflow)
77  {
78      Q14 producte;
79      unsigned char ov = 0; // inicialment assumeix que no hi ha overflow
80
81      long long prod64; // Resultat de la multiplicació (64 bits)
82
83      // Calcular resultat (64 bits), ajustant bits fracció
84      prod64 = (((long long) num1) * num2) >> 14;
85
86      // Detecció overflow
87      if (prod64 < 0)
88      {
89          // Si resultat negatiu, els bits 63..31 haurien de ser tots 1
90          if ((prod64 & MASK_SIGN_64) != MASK_SIGN_64)
91              ov = 1;
92      }
93      else
94      {
95          // Si resultat positiu, els bits 63..31 haurien de ser tots 0
96          if ((prod64 & MASK_SIGN_64) != 0)
97              ov = 1;
98
99      producte = (Q14) prod64; // Retornar només els 32 bits baixos
100
101     *overflow = ov;
102     return(producte);
103 }
```

Tal com hem vist a l'apartat “1.2.3. Producte (multiplicació) en coma fixa”, al resultat (de 64 bits, tipus `long long` en C) del producte de dos números codificats en coma fixa cal ajustar el factor  $2^f$  ( $2^{14}$  en el cas de la pràctica). Per

dividir entre  $2^{14}$ , com que es tracta d'una potència de 2, es pot aconseguir desplaçant els bits 14 posicions a la dreta (mantenint el bit de signe, desplaçament aritmètic). Malauradament, en assemblador, aquest desplaçament de valors de 64 bits no es pot fer directament, sinó que cal aplicar les tècniques ja indicades a l'apartat “6.2. Desplaçament de valors de 64 bits (amb operacions de 32 bits)”.

Pel que fa a la detecció d'*overflow* en un resultat de 64 bits, que esperem que es pugui codificar en només 32 bits, cal analitzar els bits 31 a 63 del resultat. En C s'utilitza la màscara `MASK_SIGN_64`, definida a la línia 16:

```
Q14.c
15     /* MÀSCARA per als 33 bits alts d'una multiplicació llarga */
16 #define MASK_SIGN_64    0xFFFFFFFF80000000 /* bits 63..31 */
17
```

El codi de comprovació de l'*overflow* (línies 86 a 96) és diferent segons si el resultat és positiu o negatiu. Si el resultat és positiu, tots els bits 31 a 63 han de ser 0, mentre que si el resultat és negatiu, tots els bits han de ser 1.

## 4.5. Rutina de divisió en coma fixa

La declaració en C de la funció de divisió és:

```
Q14.h
69  /* div_Q14(): calcula i retorna la divisió dels 2 primers operands,
70  (num1 / num2) codificats en Coma fixa 1:17:14.
71  El 3r paràmetre (per referència) retorna l'overflow:
72  0: no s'ha produït overflow, resultat correcte.
73  1: hi ha overflow (resultat massa gran) i el que es
74  retorna són els bits baixos del resultat.
75 */
76  extern Q14 div_Q14(Q14 num1, Q14 num2, unsigned char * overflow);
77
```

Una possible implementació en C de la funció `div_Q14()` és:

```

Q14.c
112 Q14 div_Q14(Q14 num1, Q14 num2, unsigned char * overflow)
113 {
114     Q14 divisio;
115     unsigned char ov = 0;      // inicialment assumeix que no hi ha overflow
116     unsigned int quo, res;    // Quocient i residu de div_mod()
117     unsigned char op2neg;     // El segon operand és negatiu?
118     unsigned int abs2;        // Valor absolut del segon operand
119
120     // En cas de divisió entre 0, resultat 0 i Overflow
121     if (num2 == 0)
122     {
123         divisio = 0;
124         ov = 1;
125     }
126     else
127     { // Calcular divisió amb valor invers: a/b = a * (1/b)
128
129         // Calcular valor absolut del segon operand
130         op2neg = (num2 < 0);
131         abs2 = (op2neg ? -num2 : num2);
132
133         // Calcular valor invers Q14: quo = 1.0/den (ajustant bits numerador)
134         div_mod( MAKE_Q14(1.0) << 14, abs2, &quo, &res);
135
136         // Calcular resultat num1 * 1.0/num2, fixant overflow
137         divisio = mul_Q14(num1, quo, &ov);
138
139         // Canviar signe del resultat
140         if (op2neg)
141             divisio = -divisio;
142     }
143
144     *overflow = ov;
145     return(divisio);
146 }
147

```

En primer lloc es comprova (línia 121) que no s'intenti dividir entre 0. Si fos aquest el cas, es retornarà 0 com a quocient i s'activa l'*overflow*, ja que el resultat (infinit) no es pot representar numèricament.

En cas que no es demani dividir entre 0, i tal com hem vist a l'apartat “1.2.4. Divisió en coma fixa”, podem calcular la divisió multiplicant per la inversa del divisor (denominador). Com que la rutina de divisió que hem de fer servir `div_mod()` no admet valors negatius, caldrà obtenir el valor absolut del divisor. Per tant, en cas que el divisor sigui negatiu, se'l canviarà de signe (línia 131) i la variable `op2neg` memoritzarà aquesta condició (línia 130). Llavors ja es podrà calcular la inversa del divisor, però incorporant el factor  $2^{14}$  en comptes de 1, (línia 134), i multiplicar el resultat pel dividend (línia 137). Finalment, si el divisor era negatiu, caldrà negar el resultat (línia 140-141).

## 4.6. Treball a realitzar

A la tercera i darrera fase de la pràctica cal traduir a llenguatge assemblador el codi en C que implementa les 4 routines d'operacions aritmètiques en format de Coma Fixa 1:17:14 (Q14), integrar-les a la llibreria **libQ14.a** i provar que funcionen correctament amb els jocs de proves proporcionats (i altres propis que es vulguin afegir). A continuació caldrà copiar els codis desenvolupats a les fases 1 i 2 i modificar-los per utilitzar les routines de llibreria quan sigui necessari realitzar algun càlcul en format Q14. També caldrà afegir les dependències de les fases 1 i 2 al **Makefile** de la fase 3, i aplicar els jocs de proves de les fases anteriors per verificar que usant les routines de llibreria continua funcionant.

El fitxer **3\_LibQ14/Makefile** ja incorpora les dependències per crear la llibreria **3\_LibQ14/LibQ14.a** a partir del fitxer de codi font **sources/Q14.c**. Abans, però, cal indicar els correus electrònics dels/de les integrants de l'equip a la capçalera del **Makefile**:

```
Makefile | 
5  #-----
6  #   Programador/a 1: xxx.xxx@estudiants.urv.cat
7  #   Programador/a 2: yyy.yyy@estudiants.urv.cat
8  #-----
9
```

A continuació cal canviar les dependències de creació de **LibQ14.a** per a què es generi a partir del fitxer **sources/Q14.s** (que caldrà crear). Tal com passa amb altres fitxers de codi font assemblador de fases anteriors, cal que cada rutina aritmètica tingui a l'inici els comentaris amb les especificacions corresponents (indicant també en quins registres es passa i retorna els valors/referències de cada funció). També cal incloure les declaracions del fitxer **includes/Q14.i** per poder usar màscares per accedir als camps de bits que formen els valors Q14:

```
Q14.i | 
10     @; MÀSCARES per als camps de bits de valors 1:17:14
11 MASK_SIGN = 0x80000000          @; bit 31:      signe
12 MASK_INT  = 0x7FFC000           @; bits 30..14: part entera
13 MASK_FRAC = 0x00003FFF         @; bits 13..0:  part fraccionària
14
```

Les declaracions i possible implementació en llenguatge C de les 4 routines aritmètiques en coma fixa Q14 estan disponibles als fitxers **includes/Q14.h** i **sources/Q14.c**, que ja s'han comentat als anteriors apartats “4.2. Rutina de suma en coma fixa” a “4.5. Rutina de divisió en coma fixa”.

La traducció de codi C a codi assemblador dins del fitxer [sources/Q14.s](#) no cal que sigui “literal” (com faria automàticament un compilador), sinó que podem aplicar un cert “enginy” (arrel de la paraula “enginyeria”). Per exemple:

- Les operacions de **suma i resta** són “trivials” (una única instrucció en assemblador), i per detectar l’*overflow* es pot consultar el corresponent *flag* del processador (que caldrà haver indicat que es guardi). Alternativament, es pot comprovar si 2 valors tenen o no el mateix signe aplicant una operació “O Exclusiva” entre els valors i comprovant el *flag* de negatiu: si està actiu, tenen signe diferent; i si no està actiu, tenen el mateix signe.  
Les operacions de producte i divisió en format Q14 sí que requereixen més instruccions en assemblador.
- En el cas del **producte**, el resultat de 64 bits de la multiplicació s’ha de desplaçar 14 bits a la dreta i es pot aconseguir tal com s’indica a l’apartat “6.2. Desplaçament de valors de 64 bits (amb operacions de 32 bits)”. Per a la detecció de l’*overflow*, el codi en C analitza 33 bits (31 a 63) del resultat, que són 33 bits en total i que en assemblador (de 32 bits) caldrà “descomposar” per analitzar el bit 31 per un costat i els bits 32 a 63 per un altre.
- En el cas de la **divisió**, en primer lloc es descarta el cas de divisió entre 0, retornant valors “per defecte”. Si no es vol dividir entre 0, a partir d’aquell punt cal treballar amb el valor absolut del 2n operand (divisor), doncs cal invocar la rutina `div_mod`, que treballa amb valors naturals. Per tant, pot ser necessari canviar el signe (negar) el divisor i “apuntar-se” (variable `op2neg` en el codi C) que el resultat obtingut caldrà tornar-lo a negar per tenir el resultat amb el signe correcte. L’operació de divisió es realitza en 2 passos: primer es calcula la inversa (línia 134 del codi en C) i a continuació es multiplica el resultat pel dividend (línia 137 del codi en C) utilitzant la rutina aritmètica anterior `mul_Q14`. Al primer pas cal assignar directament el valor resultat de “`MAKE_Q14(1.0 << 14)`” com a primer operand de `div_mod`. Per als paràmetres 3 i 4 (de sortida) de `div_mod` caldrà aplicar alguna de les tècniques que s’indiquen a l’apartat “6.3. Variables locals a memòria en assemblador GAS”.

El desenvolupament i prova d’aquestes 4 rutines aritmètiques es pot realitzar de forma “incremental”: desenvolupar la primera rutina i provar que funciona bé (fent les correccions que calgui, en cas contrari). Quan una rutina ja funcioni bé, passar a la següent (tenint en compte que no es pot implementar la divisió abans que el producte, doncs la divisió usa el producte per al 2n pas de càcul).

El fitxer `source/demo.c` conté un possible programa principal que utilitza les rutines de suma, producte i divisió en format Q14 per calcular l'àrea d'un trapezi:

```
demo.c
15  /* global variables as inputs and output, for computing the areas */
16  Q14 long_base = MAKE_Q14(45.12);           // B = 45.12 cm
17  Q14 short_base = MAKE_Q14(30.75);          // b = 30.75 cm
18  Q14 height = MAKE_Q14(29.99);             // h = 29.99 cm
19  Q14 trapezium_area;           // expected value: (B+b)*h/2 = 1137.67065 cm^2
20
21  int main(void)
22  {
23      unsigned char ov;                  /* possible overflow */
24      Q14 pr;                         /* partial result */
25
26      // trapezium_area = (long_base + short_base)*height / 2.0
27      pr = add_Q14(long_base, short_base, &ov);
28      if (!ov)
29      {
30          // proceed with remaining ops only if not overflow
31          pr = mul_Q14(pr, height, &ov);
32          if (!ov)
33          {
34              pr = div_Q14(pr, MAKE_Q14(2.0), &ov);
35              if (!ov)           // if everything went fine,
36                  trapezium_area = pr; // update output global variable
37          }
38      }
39  /* TESTING POINT: check the result (if no overflow)
```

El fitxer `tests/test_Q14.c` conté un possible Joc de Proves de les 4 rutines aritmètiques a traduir a assemblador. El primer valor de cada cas de prova indica l'operació que es vol provar (si es desenvolupa incrementalment, només caldria aplicar els casos de prova de les rutines que ja es tenen implementades):

```
test_Q14.c
11  enum Q14_ops {add, sub, mul, div};
12
13  /* type definition of the structured record that holds the test case values */
14  typedef struct {
15      enum Q14_ops op;    /* add / sub / mul / div */
16      Q14 num1;          /* 1st input parameter of the operation */
17      Q14 num2;          /* 2nd input parameter of the operation */
18      Q14 result;        /* Expected result of the operation */
19      unsigned char ovf; /* Expected overflow of the operation */
20  } test_ops_struct;
21
22  /* the list of test case values */
23  test_ops_struct test_case[] =
24      /* Tests operació SUMA (add) */
25  {{add, 0, 0, 0, 0},           /* 0: 0.0 + 0.0 = 0.0, overflow 0 */
```

Una vegada les 4 rutines aritmètiques estiguin implementades i provades, s'han d'aplicar als codis assemblador desenvolupats a les fases 1 i 2. Cal copiar els fitxers `1_CelsiusFahrenheit_GAS/source/CelsiusFahrenheit.s` i `2_GeoTemp/source/avgmaxmintemp.s` al directori `3_LibQ14/sources`. També caldrà copiar els jocs de proves de les rutines d'aquests 2 fitxers de les fases 1 i 2 a la carpeta `3_LibQ14/tests` per poder provar que el codi adaptat cridant a les rutines de llibreria funciona correctament. A més, serà necessari incorporar al fitxer `3_LibQ14/Makefile` les dependències i regles necessàries per generar els executables de prova dels fitxers incorporats de les fases 1 i 2.

## 5. Organització, lliurament i avaluació

Aquest apartat conté les diferents informacions sobre la formació d'equips de pràctiques, forma i terminis de lliurament, valoració/ponderació, com es realitzarà l'avaluació de les diferents fases de la pràctica, etc.

### 5.1. Treball en equip

La pràctica s'haurà de realitzar en **equips de 2 estudiants**. També es pot realitzar individualment, però l'estudiant tindrà més treball de desenvolupament. Cal indicar amb qui es forma equip a la tasca corresponent del Campus Virtual, segons el professor de laboratori que s'estigui apuntat: [Carlos Molina](#), [Simeó Reig](#) o [Carlos Soriano](#).

Es poden formar equips de laboratoris diferents, sempre que el professor del laboratori sigui el mateix (Carlos Molina: L1 i L3; Simeó Reig: L2, L4 i L5; Carlos Soriano: L6, L7 i L8).

### 5.2. Lliurament i Terminis

Cal que **cada integrant de l'equip** realitzi un **vídeo individual** de cada fase lliurada, de **no més de 2 minuts de durada per cada fase**, amb l'explicació i demostració del correcte funcionament dels codis més significatius, fent èmfasi en la part de la pràctica on aquell integrant hagi participat més. En el cas de vídeos de més de 2 minuts, el professor pot decidir revisar només fins al minut 2:00.

El lliurament es realitzarà penjant un únic fitxer comprimit (en format .zip, mida màxima 700 MiBytes) a la [tasca corresponent del campus virtual](#). El fitxer amb el lliurament ha de contenir:

- L'informe de la pràctica en PDF, amb els següents punts (els punts 3 a 6: repetits per cada fase lliurada):
  1. **Portada:** nom de l'assignatura, nom de la pràctica i integrants de l'equip.
  2. **Índex** del document, amb número de pàgina.
  3. **Especificacions:** declaracions de les routines amb comentaris explicatius (com als fitxers de capçalera/include).
  4. **Disseny:** decisions més rellevants preses en la solució proposada.
  5. **Implementació:** codi GAS de les routines, modificacions als Makefile.
  6. **Joc de proves “ampliat”:** llistat de proves addicionals que s'hagin realitzat per verificar el correcte funcionament dels codis desenvolupats (a banda dels jocs de proves ja facilitats, que no cal indicar però sí superar).

- Un directori amb nom `PracticaARMequipAANN` (AANN és el codi de l'equip que ha desenvolupat la pràctica: CM06, SR12 o CS27, per exemple) en el qual s'inclouran tots els codis del projecte (amb la corresponent estructura de directoris i fitxers de les fases 1, 2 i 3, que permeti generar els executables amb `make`). Si no es lliura alguna fase, no s'ha d'incloure el directori d'aquella fase. Cal assegurar-se que s'ha indicat els correus electrònics d'estudiant URV a les capçaleres dels fitxers on s'ha afegit codi.
- Un directori amb nom `Videos` amb cada vídeo individual de cada component de l'equip de cada fase lliurada. El format dels vídeos (.mp4, .avi, etc.) és lliure. El nom dels vídeos serà `ExplicaPracticaARM_AANN_FaseN_CognomsNom`.

El termini màxim de lliurament de la pràctica en 1a convocatòria és el **dilluns 3/mai/2021 abans de les 23:55**. Els lliuraments posteriors a aquesta data es consideraran per a 2a convocatòria (tot i que l'entrevista es podria realitzar abans, segons disponibilitat del professor). El termini màxim de lliurament de la pràctica en 2a convocatòria és el **divendres 18/juny/2021 abans de les 23:55**.

### 5.3. Valoració del treball (avaluació)

Posteriorment al lliurament de la pràctica, es realitzarà una entrevista i prova del funcionament dels codis amb tots/es els/les integrants de l'equip, en un horari a convenir amb el professor de laboratori. Si es lliura la pràctica però sense vídeos o no es realitza l'entrevista, la nota serà de No Presentat. Si els codis no compilen o executen bé, la nota serà Suspens.

La nota de la pràctica és individual (els/les components d'un mateix equip, poden obtenir notes diferents). L'avaluació de la pràctica, si bé és conjunta pel que fa a la feina feta, és individual en funció de com es respongui a les diferents preguntes plantejades pel professor de laboratori. Per tant, els dos membres de l'equip ha de conèixer amb prou detall totes les parts de la pràctica, doncs el professor pot preguntar per parts que no hagi desenvolupat l'estudiant sinó el/la company/a.

Durant l'entrevista, el professor també pot demanar a l'estudiant que utilitzi les eines d'edició, compilació, depuració, etc. per comprovar la destresa de l'estudiant usant aquestes eines.

La ponderació/valoració de cada fase a la nota final de la pràctica és:

- Fase 1: 30%.
- Fase 2: 40%.
- Fase 3: 30%.

Cal obtenir una nota mínima de 4 en aquesta pràctica (sobre 10) per poder fer mitjana amb les altres 3 parts avaluables de l'assignatura.

## 5.4. Explicació de la pràctica en vídeo

Al campus virtual es disposa d'[informació, guies i programes per crear i editar els vídeos](#) explicatius.

És recomanable no esperar a finalitzar tota la pràctica per crear els vídeos. Pot ser millor preparar el vídeo corresponent de cada fase quan es finalitzi aquella fase, que es tindrà “més fresc” el treball realitzat.

## 6. Sobre la programació en assemblador GAS

Aquesta darrera secció conté informació i tècniques a tenir en compte quan es programa en assemblador GAS.

### 6.1. Criteris d'interfície entre llenguatge C i assemblador GAS

La creació de rutines en assemblador GAS que hagin de ser cridades des de llenguatge C (o a l'inrevés) requereix aplicar els criteris del llenguatge C pel que fa a pas de paràmetres, retorn de resultats, modificació de registres, etc.

A continuació es recull la informació més important:

- els 4 primers paràmetres d'una funció en C (començant per l'esquerra) es guarden respectivament en els registres r0, r1, r2 i r3,
- a partir del 5é paràmetre (inclòs), tots els paràmetres s'ubicaran a la pila,
- en el cas de paràmetres per valor, el registre o posició de pila corresponent contindrà el valor que es passa pel paràmetre,
- en el cas de paràmetres per referència, el registre o posició de pila corresponent contindrà l'adreça de memòria de la variable que es passa pel paràmetre,
- amb variables de tipus vector, taula o multicamp sempre s'utilitza el pas de paràmetres per referència,
- els valors de 8 o 16 bits emmagatzemats en registres, han d'emplenar els bits de més pes amb el valor corresponent: bits a 0 en el cas de valors naturals i bit de signe (0: positiu, 1: negatiu) en el cas de valors enters,
- el resultat de la funció es retornarà sempre pel registre r0,
- després de cridar una rutina en C, podrien quedar modificats els registres r0, r1, r2, r3 i r12.

A continuació es mostra un exemple de traducció de llenguatge C a llenguatge assemblador aplicant la convenció de pas de paràmetres i retorn de resultat que utilitza el compilador arm-none-eabi-gcc.

### Llenguatge C:

```
char my_function (int a, short b, unsigned char c, short *d);

int main(void)
{
    char res;
    short tmp = 0x4000;

    res = my_function(0x10000000, 0x2000, 0x30, &tmp);
}
```

### Llenguatge Assemblador:

```
mov r0, #0x10000000      @; 1r paràmetre a r0 per valor
mov r1, #0x2000          @; 2n paràmetre a r1 per valor
mov r2, #0x30            @; 3r paràmetre a r2 per valor
ldr r3, =tmp             @; 4t paràmetre a r3 per referència

bl my_function           @; resultat de la funció a r0

ldr r1, =res
strb r0, [r1]            @; escriure resultat a variable res
```

## 6.2. Desplaçament de valors de 64 bits (amb operacions de 32 bits)

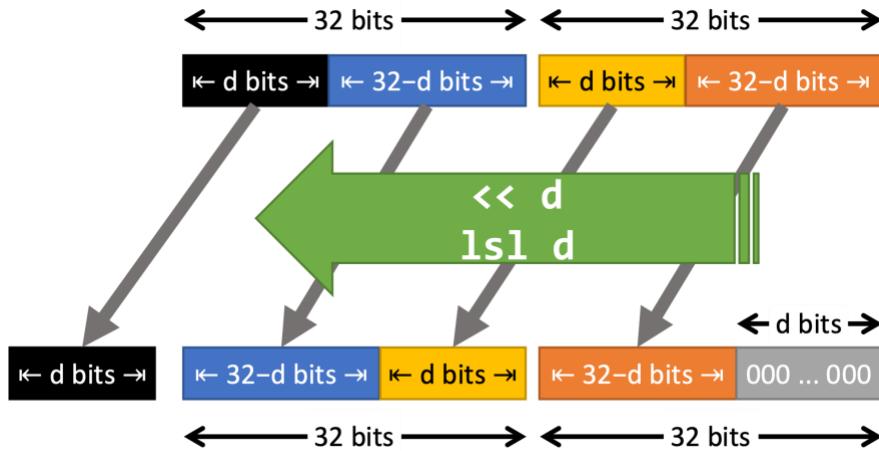
Per treballar amb valors en coma fixa, en alguns casos cal aplicar el factor  $2^f$ , multiplicant o dividint. Tal com vam veure al Tema 1 de FC, es pot multiplicar o dividir per potències de 2 mitjançant desplaçaments a la dreta (per multiplicar) o a l'esquerra (per dividir). En C utilitzem els operadors `<< i >>` i en assemblador els operadors `lsl`, `lsr` i `asr`. El problema arriba quan cal desplaçar valors de 64 bits. En C podem usar variables de 64 bits de tipus `long long` i desplaçar directament, però a Fonaments de Computadors estem usant un processador de 32 bits, i cal treballar amb la part alta i la part baixa de 32 bits. A continuació s'indica com aconseguir desplaçaments de 64 bits en processadors de 32 bits.

Suposarem que el valor de 64 bits es troba en 2 registres:  $R_{lo}$  (32 bits de menys pes) i  $R_{hi}$  (32 bits de més pes). El número de bits a desplaçar es trobarà en un altre registre:  $R_d$ . El resultat (de 64 bits) quedarà als mateixos registres d'entrada  $R_{lo}$  i  $R_{hi}$ .

Hi ha un “cas trivial” (desplaçar 32 bits) on directament un registre es carrega amb el valor de l'altre registre i l'altre registre queda amb tots els bits a 0 (valor 0) o tots els bits a 1 (valor -1), segons el tipus de desplaçament (lògic: emplena amb zeros; aritmètic a la dreta: manté el signe, 0 o 1). Al tercer cas (desplaçament aritmètic a la dreta, cal fer extensió de signe), a la primera instrucció s'analitza si el valor és positiu o negatiu (*flag* de signe) i, segons el cas, es carrega el valor -1 (si és negatiu, tots els bits alts a 1) o 0 (si és positiu, tots els bits alts a 0):

<code>val64 &lt;&lt; 32</code>	<code>nat64 &gt;&gt; 32</code>	<code>ent64 &gt;&gt; 32</code>
<code>mov R<sub>hi</sub>, R<sub>lo</sub></code> <code>mov R<sub>lo</sub>, #0</code>	<code>mov R<sub>lo</sub>, R<sub>hi</sub></code> <code>mov R<sub>hi</sub>, #0</code>	<code>movs R<sub>lo</sub>, R<sub>hi</sub></code> <code>movmi R<sub>hi</sub>, #-1</code> <code>movpl R<sub>hi</sub>, #0</code>

Per entendre el “cas general” (número de bits a desplaçar = 1..31) és millor plantejar-lo visualment. En primer lloc, el **desplaçament lògic a l’esquerra** dels bits continguts a  $R_{hi}:R_{lo}$  la quantitat de posicions  $d$  indicada a  $R_d$  ( $R_{hi}:R_{lo} \ll R_d$ , equivalent a multiplicar per  $2^d$ ):



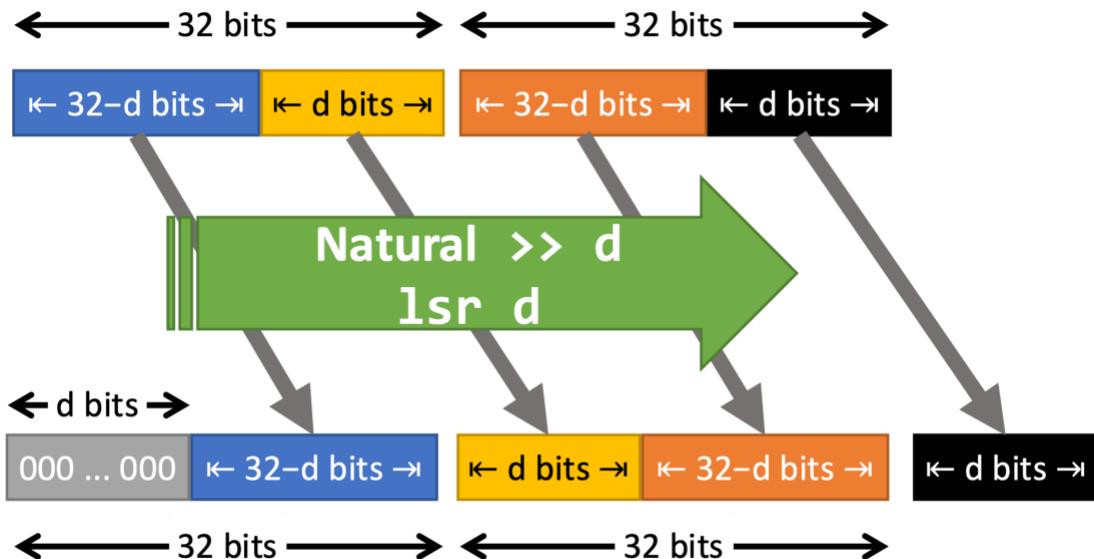
Un possible codi en assemblador GAS que permet desplaçar a l’esquerra  $R_d$  bits (1-31) continguts a  $R_{hi}:R_{lo}$  seria el següent (els subíndexs dels registres és per indicar la informació que contenen; en un cas real, cal utilitzar registres “reals” R0-R12):

```

rsb R32-d, Rd, #32          @; R32-d ← 32-d
mov Rout, Rhi, lsr R32-d    @; (Opcional: els d bits “que surten”)
mov Rhi, Rhi, lsl Rd        @; Desplaçar a l'esquerra part alta
mov Rtemp, Rlo, lsr R32-d    @; Agafar els d bits entre Rlo i Rhi
orr Rhi, Rtemp                @; Afegir d bits baixos a Rhi
mov Rlo, Rlo, lsl Rd        @; Desplaçar a l'esquerra part baixa

```

En segon lloc, el **desplaçament lògic a la dreta** dels bits continguts a  $R_{hi}:R_{lo}$  la quantitat de posicions  $d$  indicada a  $R_d$  ( $R_{hi}:R_{lo} >> R_d$ , equivalent a dividir un valor sense signe entre  $2^d$ ):



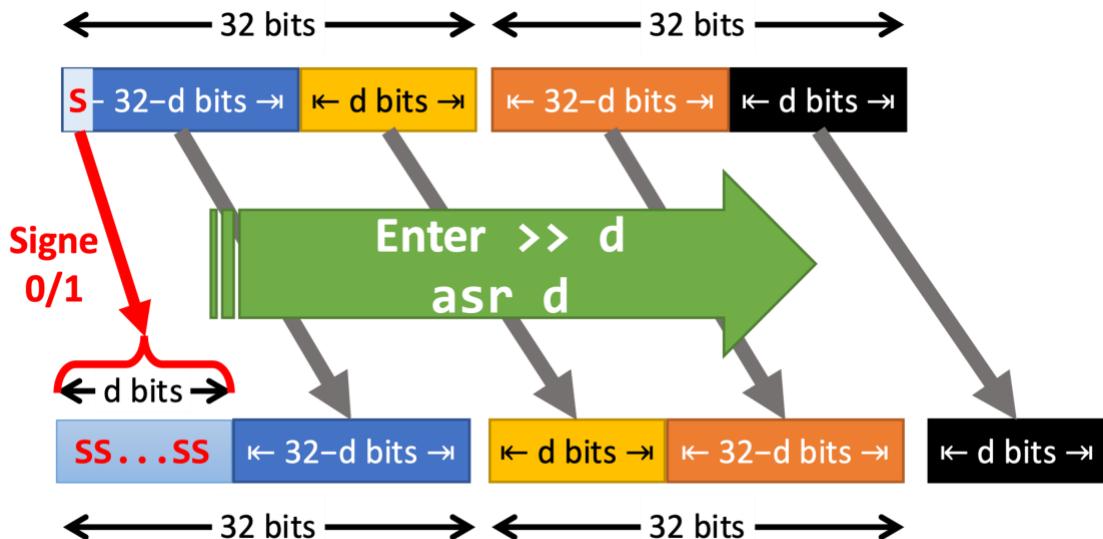
Un possible codi en assemblador GAS que permet desplaçar a la dreta  $R_d$  bits (1-31) continguts a  $R_{hi}:R_{lo}$  seria el següent (els subíndexs dels registres és per indicar la informació que contenen; en un cas real, cal utilitzar registres “reals” R0-R12):

```

rsb R32-d, Rd, #32          @; R32-d ← 32-d
mov Rout, Rlo, lsl R32-d    @; (Opcional: obtenir ...
mov Rout, Rout, lsr R32-d    @; ... els d bits “que surten”)
mov Rlo, Rlo, lsr Rd        @; Desplaçar a la dreta part baixa
orr Rlo, Rhi, lsl R32-d     @; Afegir a Rlo els d bits entre Rhi i Rlo
mov Rhi, Rhi, lsr Rd        @; Desplaçar a la dreta part alta

```

Finalment, el **desplaçament aritmètic a la dreta** dels bits continguts a  $R_{hi}:R_{lo}$  la quantitat de posicions  $d$  indicada a  $R_d$  ( $R_{hi}:R_{lo} \gg R_d$ , equivalent a dividir un valor amb signe entre  $2^d$ ; els bits que “entren” per l’esquerra, han de mantenir el signe **S** original del bit 63):



Un possible codi en assemblador GAS que permet desplaçar aritmèticament (mantenint el signe) a la dreta  $R_d$  bits (1-31) continguts a  $R_{hi}:R_{lo}$  seria el següent (els subíndexs dels registres és per indicar la informació que contenen; en un cas real, cal utilitzar registres “reals” R0-R12):

```

rsb R32-d, Rd, #32      @; R32-d ← 32-d
mov Rout, Rlo, lsl R32-d    @; (Opcional: obtenir ...
mov Rout, Rout, lsr R32-d    @; ... els d bits “que surten”)
mov Rlo, Rlo, lsr Rd      @; Desplaçar a la dreta part baixa
orr Rlo, Rhi, lsl R32-d    @; Afegir a Rlo els d bits entre Rhi i Rlo
mov Rhi, Rhi, asr Rd      @; Desplaçar a la dreta part alta ...
@; ... mantenint el Signe

```

### 6.3. Variables locals a memòria en assemblador GAS

En alguns punts d'aquesta pràctica és necessari passar paràmetres per referència que contindran valors de sortida (en el cas de `div_mod()` es retorna el quotient i el residu; a les rutines aritmètiques, l'*overflow* es retorna per referència) i, per tant, cal indicar adreces de memòria on es desaran els resultats.

Les variables locals de les rutines en assemblador, es recomana desar-les a registres, que són més ràpids d'accendir que si s'emmagatzemen a memòria. Però si cal passar una variable local per referència, els registres no tenen “adreça” (doncs no estan a memòria, sinó dins de la CPU). En aquests casos, les variables locals afectades cal emmagatzemar-les a memòria. Ho podem aconseguir de dues formes:

1. Declarar les variables a les seccions `.data` o `.bss`. Aquesta “solució” permet tenir un espai a memòria on emmagatzemar el valor de les variables, però aquestes variables deixen de ser “locals”, en el sentit que les variables locals només haurien d'existir mentre s'executa la rutina on estan declarades. Definint-les a les seccions `.data` o `.bss`, les variables passen a comportar-se com “globals” i continuen existint i conservant el valor fins i tot després d'haver-se completat l'execució de la rutina on estan declarades. És una solució simple “per sortir del pas” però no tan bona com la que es planteja a continuació.
2. Declarar les variables a la pila (*stack*). Aquesta és la millor solució, tot i que una mica més complexa d'implementar. L'espai de memòria necessari per a les variables locals que s'han de passar per referència, s'agafa de la pila (accessible amb el registre `SP`: *Stack Pointer*). Quan es retorna de la rutina, l'espai de la pila per a variables locals s'elimina i deixen d'existir, tal com s'espera de les variables locals.  
Els detalls complets es poden consultar a l'apartat 2.4.11 de la Teoria del Tema 2 “Variables locales en la pila (*stack*)”. I també al problema 4.19 del Tema 2 on es disposa d'un exemple d'aplicació (per cridar a `div_mod`).