Trabajo Entregable 0 - Introducción al Diseño Lógico

Participantes:
- Federico Goncalves - 03866/5
- Joaquín Guzmán - 03751/4
- Tomás Gamarra - 03852/8

1. Objetivos.

En este informe se busca estudiar la precisión y rango de valores con el que nos permite trabajar la representación en punto fijo de las variables de pendiente (**m**) y ordenada al origen (**b**) de una ecuación de una recta (**y=mx+b).**Así como también poder trabajar con distintas representaciones y poder hacer las operaciones correspondientes. Para ello, se abordarán los siguientes aspectos:

a) Rango de representación y resolución de **m**.
b) Rango de representación y la resolución de **b**.
c) Elección de una representación en punto fijo para **x** e **y** tal que permita representar **m** y **b** sin pérdidas significativas.
d) Rango de representación y resolución de **x** e **y** bajo la representación elegida.
e) Valores a los que debería acotarse **x** para que en casos límites de valores de **m** y **b** no se produzca un overflow de **y** utilizando la representación elegida.

2. A tener en cuenta :

- Asumimos para todas las representaciones de las variables **m**,**b**,**x** e **y** que su interpretación se hace en **CA2**. Para el caso de las variables **m** y **b** que se nos da una representación en punto fijo definida, asumimos que hay un bit más que representa el signo.

a)

Para **m** se adopta una representación **Q(0,15)**.

Rango de representación.

$$R_m = [-(\sum_{i=1}^{15} 2^{-i} + 2^{-15}); \sum_{i=1}^{15} 2^{-i}]$$

$$R_m = [-1; 1 - 2^{-15}] = [-1; 0.9999694824]$$

Resolución.

$$Res_m = 2^{-15}$$

b)

Para **b** se adopta una representación **Q(7,8)**.

Rango de representación.

$$R_b = [-(2^7 - 1 + \sum_{i=1}^{8} 2^{-i} + 2^{-8}); 2^7 - 1 + \sum_{i=1}^{8} 2^{-i}]$$

$$R_b = [-128; 128 - 2^{-8}] = [-128; 127.9960938]$$

Resolución.

$$Res_b = 2^{-8}$$

c)

Para que no haya pérdidas significativas para **m** y **b** siendo representadas en la misma representación que **x** e **y** (**Q(c,d)**) es necesario que el rango de ésta sea por lo menos igual al rango más grande y la resolución sea igual o mayor a la más chica entre las representaciones de m y b.

Rango más grande ------> R_b
Entonces para Q(c,d) ------> c >= 7

Resolución más chica ------> Res_m
Entonces para Q(c,d) ------> d >= 15

Buscamos representar una ecuación de una recta de la forma y=mx+b en donde buscamos que las variables x,y,m y b utilicen una única representación de punto fijo. En base a lo antes explicado, se llega a la conclusión de que para no tener pérdida de cifras significativas en el pase de una representación a otra, necesitaremos de los 32 bits totales un mínimo de **7 bits** asignados a la **parte entera** y un mínimo de **15 bits** asignados a la **parte fraccionaria**.

La asignación de los 10 bits restantes se vuelve dependiente del rango y resolución que se le desee asignar a las variables **x** e **y**. En nuestro caso, elegimos arbitrariamente una división de bits para las partes entera y fraccionaria, quedando finalmente la representación en punto fijo para **x** e **y** : **Q(15,16)** (más el bit de signo).

Ahora hemos de plantear una sistematización para poder pasar cualquier número de representación de **m (Q(0,15))** o **b (Q(7,8))** a la representación elegida para **x** e **y (Q(15,16))**.

A continuación mostramos cómo sería la distribución de los bits para una representación Q(15,16) cualquiera.

Aclaración : Los subíndices de cada bit **b** indican el peso que tiene ese bit en cuanto a potencias de 2 , así como también el subíndice **s** en el bit más significativo (MSB) indica que ese es el **bit de signo**.

$$b_s\ b_{14}\ldots b_0\ ,b_{-1}\ b_{-2}\ b_{-3}\ b_{-4}\ b_{-5}\ b_{-6}\ b_{-7}\ b_{-8}\ b_{-9}\ b_{-10}\ b_{-11}\ b_{-12}\ b_{-13}\ b_{-14}\ b_{-15}\ b_{-16}$$

Caso m: Q(0,15)

Tenemos un valor cualquiera de m tal que sus bits son :

$$x_s\ x_{-1}\ x_{-2}\ x_{-3}\ \ x_{-4}\ x_{-5}\ \ x_{-6}\ x_{-7}\ x_{-8}\ x_{-9}\ \ \ x_{-10}\ x_{-11}\ \ x_{-12}\ x_{-13}\ \ x_{-14}\ x_{-15}$$

x_s=Bit de signo.
x_{-n}=Bit que representa la potencia de 2 negativa n-ésima.

Si ahora solapamos como quedaría la cadena de bits que definimos anteriormente para **m** en una representación **Q(15,16)** cualquiera , obtendremos lo siguiente :
.

$$b_s\ b_{14}\ldots b_0\ ,b_{-1}\ b_{-2}\ b_{-3}\ b_{-4}\ b_{-5}\ b_{-6}\ b_{-7}\ b_{-8}\ b_{-9}\ b_{-10}\ b_{-11}\ b_{-12}\ b_{-13}\ b_{-14}\ b_{-15}\ b_{-16}$$
$$0_s\ 0_{14}\ldots 0_0\ ,x_s\ \ x_{-1}\ x_{-2}\ x_{-3}\ \ x_{-4}\ x_{-5}\ \ x_{-6}\ x_{-7}\ x_{-8}\ \ x_{-9}\ \ \ x_{-10}\ x_{-11}\ \ x_{-12}\ x_{-13}\ \ x_{-14}\ x_{-15}$$

Entonces la cadena de bits en representación **Q(0,15)** al querer pasarla a **Q(15,16)** queda desplazado **1 bit** hacia la derecha. Esto puede arreglarse con la operación de desplazamiento a la izquierda por 1 lugar (multiplicación por 2) :

$$b_s\ b_{14}\ldots b_0\ ,b_{-1}\ b_{-2}\ b_{-3}\ b_{-4}\ b_{-5}\ b_{-6}\ b_{-7}\ b_{-8}\ b_{-9}\ b_{-10}\ b_{-11}\ b_{-12}\ b_{-13}\ b_{-14}\ b_{-15}\ b_{-16}$$
$$0_s\ 0_{14}\ldots 0_0\ ,x_s\ \ x_{-1}\ x_{-2}\ x_{-3}\ \ x_{-4}\ x_{-5}\ \ x_{-6}\ x_{-7}\ x_{-8}\ \ x_{-9}\ \ \ x_{-10}\ x_{-11}\ \ x_{-12}\ x_{-13}\ \ x_{-14}\ x_{-15}\ * 2^1$$

$$=$$

$$b_s\ b_{14}\ldots b_0\ ,b_{-1}\ b_{-2}\ b_{-3}\ b_{-4}\ b_{-5}\ b_{-6}\ b_{-7}\ b_{-8}\ b_{-9}\ b_{-10}\ b_{-11}\ b_{-12}\ b_{-13}\ b_{-14}\ b_{-15}\ b_{-16}$$
$$0_s\ \ 0_{14}\ldots x_s\ ,x_{-1}\ x_{-2}\ x_{-3}\ \ x_{-4}\ x_{-5}\ \ x_{-6}\ x_{-7}\ x_{-8}\ x_{-9}\ \ x_{-10}\ x_{-11}\ \ x_{-12}\ x_{-13}\ x_{-14}\ x_{-15}\ 0_{-16}$$

Otro problema que surge ahora es que el bit de signo ($\mathbf{x_s}$) queda en la posición del bit menos significativo de la parte entera. La solución que planteamos es la siguiente:

$$
\begin{array}{rl}
 & 0_s\ 0_{14}\ 0_{13}\ 0_{12}\ 0_{11}\ 0_{10}\ 0_9\ 0_8\ 0_7\ 0_6\ 0_5\ 0_4\ 0_3\ 0_2\ x_s\ x_0,x_{-1}\ldots x_{-15}\ 0_{-16} \\
+ & 0_s\ 1_{14}\ 1_{13}\ 1_{12}\ 1_{11}\ 1_{10}\ 1_9\ 1_8\ 1_7\ 1_6\ 1_5\ 1_4\ 1_3\ 1_2\ 1_1\ 0_0\ ,0_{-1}\ldots 0_{-15}\ 0_{-16} \\
\hline
 & x_s\ \overline{x}_s\ \overline{x}_s\ \overline{x}_s\ \overline{x}_s\ \overline{x}_s\ \overline{x}_s\ \overline{x}_s\ \overline{x}_s\ \overline{x}_s\ \overline{x}_s\ \overline{x}_s\ \overline{x}_s\ \overline{x}_s\ \overline{x}_s\ ,x_{-1}\ldots x_{-15}\ 0_{-16} \\
\text{XOR} & 0_s\ 1_{14}\ 1_{13}\ 1_{12}\ 1_{11}\ 1_{10}\ 1_9\ 1_8\ 1_7\ 1_6\ 1_5\ 1_4\ 1_3\ 1_2\ 1_1\ ,0_{-1}\ldots 0_{-15}\ 0_{-16} \\
\hline
 & x_s\ x_s\ x_s\ x_s\ x_s\ x_s\ x_s\ x_s\ x_s\ x_s\ x_s\ x_s\ x_s\ x_s\ x_s\ ,x_{-1}\ldots x_{-15}\ 0_{-16}
\end{array}
$$

Si ahora volvemos a solapar como quedo la cadena de bits que representaba a **m** en contraste con una representación **Q(15,16)** cualquiera observamos que quedó expresada correctamente :

$$
\begin{array}{l}
b_s\ b_{14}\ldots b_0\ ,b_{-1}\ b_{-2}\ b_{-3}\ b_{-4}\ b_{-5}\ b_{-6}\ b_{-7}\ b_{-8}\ b_{-9}\ b_{-10}\ b_{-11}\ b_{-12}\ b_{-13}\ b_{-14}\ b_{-15}\ b_{-16} \\
x_s\ x_{s14}\ldots x_{s0},x_{-1}\ x_{-2}\ x_{-3}\ x_{-4}\ x_{-5}\ x_{-6}\ x_{-7}\ x_{-8}\ x_{-9}\ x_{-10}\ x_{-11}\ x_{-12}\ x_{-13}\ x_{-14}\ x_{-15}\ 0_{-16}
\end{array}
$$

Caso b: Q(7,8)

Tenemos un valor cualquiera de **b** tal que sus bits son

$$x_s\ x_6\ x_5\ x_4\ x_3\ x_2\ x_1\ x_0,x_{-1}\ x_{-2}\ x_{-3}\ x_{-4}\ x_{-5}\ x_{-6}\ x_{-7}\ x_{-8}$$

Si ahora solapamos como quedaría la cadena de bits que definimos anteriormente para **b** en una representación **Q(15,16)** cualquiera , obtendremos lo siguiente :

$$
\begin{array}{l}
b_s\ b_{14}\ldots b_0\ ,b_{-1}\ b_{-2}\ b_{-3}\ b_{-4}\ b_{-5}\ b_{-6}\ b_{-7}\ b_{-8}\ b_{-9}\ b_{-10}\ b_{-11}\ b_{-12}\ b_{-13}\ b_{-14}\ b_{-15}\ b_{-16} \\
0_s\ 0_{14}\ldots 0_0\ ,x_s\ x_6\ x_5\ x_4\ x_3\ x_2\ x_1\ x_0\ x_{-1}\ x_{-2}\ x_{-3}\ x_{-4}\ x_{-5}\ x_{-6}\ x_{-7}\ x_{-8}
\end{array}
$$

Notamos que la cadena de bits está desplazada a la derecha 8 lugares. Por lo tanto nuestra primera operación para convertir la cadena de bits en **Q(7,8)** a **Q(15,16)** será un desplazamiento a izquierda de 8 lugares (multiplicación por 2^8):

$$
\begin{array}{l}
b_s\ b_{14}\ldots b_0\ ,b_{-1}\ b_{-2}\ b_{-3}\ b_{-4}\ b_{-5}\ b_{-6}\ b_{-7}\ b_{-8}\ b_{-9}\ b_{-10}\ b_{-11}\ b_{-12}\ b_{-13}\ b_{-14}\ b_{-15}\ b_{-16} \\
0_s\ 0_{14}\ldots 0_0\ ,x_s\ x_6\ x_5\ x_4\ x_3\ x_2\ x_1\ x_0\ x_{-1}\ x_{-2}\ x_{-3}\ x_{-4}\ x_{-5}\ x_{-6}\ x_{-7}\ x_{-8} * 2^8
\end{array}
$$

=

$b_s\ b_{14}\ldots b_8\ b_7\ b_6\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0\ ,b_{-1}\ b_{-2}\ b_{-3}\ b_{-4}\ b_{-5}\ b_{-6}\ b_{-7}\ b_{-8}\ b_{-9}\ldots b_{-16}$
$0_s\ 0_{14}\ldots 0_8\ x_s\ x_6\ x_5\ x_4\ x_3\ x_2\ x_1\ x_0\ ,x_{-1}\ x_{-2}\ x_{-3}\ x_{-4}\ x_{-5}\ x_{-6}\ x_{-7}\ x_{-8}\ 0_{-9}\ldots 0_{-16}$

Así como en el caso de **Q(0,15)**, nuevamente surge el problema de la posición del bit de signo. En este caso se encuentra en el **bit 7** cuando debería encontrarse en el bit más significativo. Las operaciones para desplazar únicamente el bit de signo serán en lógica las mismas que para el caso anterior:

$$\begin{array}{rc} & 0_s\ 0_{14}\ 0_{13}\ 0_{12}\ 0_{11}\ 0_{10}\ 0_9\ 0_8\ x_s\ x_6\ \ldots x_{-8}\ 0_{-9}\ldots 0_{-16} \\ + & 0_s\ 1_{14}\ 1_{13}\ 1_{12}\ 1_{11}\ 1_{10}\ 1_9\ 1_8\ 1_7\ 0_6\ \ldots 0_{-8}\ 0_{-9}\ldots 0_{-16} \\ \hline & x_s\ \overline{x}_s\ \overline{x}_s\ \overline{x}_s\ \overline{x}_s\ \overline{x}_s\ \overline{x}_s\ \overline{x}_s\ \overline{x}_s\ x_6\ldots x_{-8}\ 0_{-9}\ldots 0_{-16} \\ \text{XOR} & 0_s\ 1_{14}\ 1_{13}\ 1_{12}\ 1_{11}\ 1_{10}\ 1_9\ 1_8\ 1_7\ 0_6\ \ldots 0_{-8}\ 0_{-9}\ldots 0_{-16} \\ \hline & x_s\ x_s\ x_s\ x_s\ x_s\ x_s\ x_s\ x_s\ x_s\ x_6\ldots x_{-8}\ 0_{-9}\ldots 0_{-16} \end{array}$$

Si ahora volvemos a solapar como quedo la cadena de bits que representaba a **b** en contraste con una representación **Q(15,16)** cualquiera observamos que quedó expresada correctamente :

$b_s\ b_{14}\ldots b_7\ b_6\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0,b_{-1}\ b_{-2}\ b_{-3}\ b_{-4}\ b_{-5}\ b_{-6}\ b_{-7}\ b_{-8}\ b_{-9}\ldots b_{-16}$
$x_s\ x_{s14}\ldots x_{s7}\ x_6\ x_5\ x_4\ x_3\ x_2\ x_1\ x_0,x_{-1}\ x_{-2}\ x_{-3}\ x_{-4}\ x_{-5}\ x_{-6}\ x_{-7}\ x_{-8}\ 0_{-9}\ldots 0_{-16}$

Como se puede ver sumamos una cadena de bits con los bits que representan “la nueva parte entera en 1”. Esto genera un acarreo gracias a la suma, con el cual “movemos” el bit de signo al bit más significativo (MSB) y con la operación XOR junto con la máscara propuesta convertimos cualquier valor que haya quedado en los bits de parte entera a x_s. Para el caso negativo está bien que los “nuevos bits” queden en 1 ya que con la interpretación CA2 no se le da “peso”, por otro lado en el caso positivo, los “nuevos bits” quedan en 0, lo que nuevamente no les da “peso”.

d)

Para **x** e **y** se adopta una representación **Q(15,16)**.

Rango de representación.

$$R_{x,y}=[-\ (2^{15}-1+\sum_{i=1}^{16}2^{-i}+2^{-16})\ ;\ 2^{15}-1+\sum_{i=1}^{16}2^{-i}]$$

$$R_{x,y}=[-\ 32768;\ 32768\ -\ 2^{-16}]=[-\ 32768;\ 32767.99998]$$

Resolución.

$$Res_{x,y}=2^{-16}$$

e)
Determinaremos en base a la representación que elegimos para **x** e **y** (**Q(15,16)**) los **valores máximos** que puede tomar la variable **x** en distintos **casos límites**.
Para hacer esto imponemos valores maximos a las variables **m** y **b** , luego fijamos el valor máximo que podemos representar en la variable **y** en base a los signos de m y de b, finalmente despejamos de la ecuación de la recta el valor máximo de x que puede introducirse para que la variable **y** no se desborde (overflow).

Ecuación de la recta :
$y = mx + b$

Despejando **x** de la ecuación nos quedaría :
$x_{max} = \frac{y-b}{m}$

En base a esta última ecuación calcularemos el valor de **x máximo** que puede tomar la variable para los casos extremos siguientes :

Caso 1:

-El valor más positivo de m → $m = 1 - 2^{-15}$
-El valor más positivo de b → $b = 128 - 2^{-8}$
-El valor más positivo de y → $y = 32768 - 2^{-16}$

Obtenemos como resultado que en estas circunstancias el valor máximo que puede tomar la variable x es → $x_1 = 32641,00002$

Caso 2:

-El valor más negativo de m → $m = -1$

-El valor más positivo de b $\rightarrow b = 128 - 2^{-8}$

-El valor más positivo de y $\rightarrow y = 32768 - 2^{-16}$

Obtenemos como resultado que en estas circunstancias el valor máximo que puede tomar la variable x es $\rightarrow x_2 = -32640,00389$

Caso 3:

-El valor más positivo de m $\rightarrow m = 1 - 2^{-15}$

-El valor más negativo de b $\rightarrow b = -128$

-El valor más negativo de y $\rightarrow y = -32768$

Obtenemos como resultado que en estas circunstancias el valor máximo que puede tomar la variable x es $\rightarrow x_3 = -32640,00389$

Caso 4:

-El valor más negativo de m $\rightarrow m = -1$

-El valor más negativo de b $\rightarrow b = -128$

-El valor más negativo de y $\rightarrow y = -32768$

Obtenemos como resultado que en estas circunstancias el valor máximo que puede tomar la variable x es $\rightarrow x_4 = 32640$

En cuanto al control que llevaremos de los valores máximos que puede tomar la variable **x** en el programa trabajaremos con un intervalo definido en base a estos valores límites calculados anteriormente. Este intervalo irá desde el número menos negativo que obtuvimos ($x_2 = -32640,00389$) hasta el valor menos positivo que obtuvimos ($x_4 = 32640$). Utilizando esta cota podremos asegurarnos que el programa será preciso en el cálculo del valor de **y** evitando desbordamiento en el resultado final, lo que generaría errores.

Código

Cómo compilarlo

Al trabajar con librerías es necesario que el archivo ejecutable cuente con el código principal y el código fuente de la librería compilados en uno mismo. Ejemplo de compilación desde el directorio **TEO_GRUPO17**:

```
~/Universidad/IDL/TE0_GRUPO17$ gcc Grupo17h.c src/Grupo17funciones.c -o puntoh
```

Aclaraciones en cuanto al código:

A la hora de resolver los problemas de los enunciados *f* y *g*,habíamos implementado las funciones para esos casos específicos, lo cual funcionaba, pero al querer reutilizar las funciones para el inciso h esto era poco eficiente. Por lo que decidimos hacer una librería de la cual podamos usar las funciones para todos los incisos. A su vez decidimos generalizar los métodos para que funcionen para cualquier representación **Q(c,d)** en 32 bits , enviando como parámetros a **c** y **d** a las distintas funciones. Sin embargo esto complicó un poco el desarrollo del código, lo cual podríamos haber evitado si hubiésemos hecho las funciones para los casos específicos de representaciones que se nos solicitaba (Q(0,15) y Q(7,8)). De igual forma, vimos una oportunidad de intentar desafiarnos en los algoritmos y desarrollarlo de esta manera la cual sería más que eficiente y nos dará la oportunidad de en un futuro trabajar con distintas representaciones.

Punto f

Constantes:

```
#define nBitsE 7
#define nBitsF 8
#define nDigE 3
#define nDigF 4
```

- **nBitsE:** Número de bits de la parte entera **Q(7,8)**
- **nBitsF:** Número de bits de la parte fraccionaria **Q(7,8)**
- **nDigE:** Número de dígitos de parte entera que se ingresan por teclado
- **nDigF:** Número de dígitos de parte fraccionaria que se ingresan por teclado

Función `ingresarEnDecimal_16:`

Esta función es la encargada de pedir el ingreso de un string que represente un número float que se pueda representar en 16 bits, y almacenar en una variable ese mismo número expresado en el string pero con su verdadero valor en binario.
La función recibe como parámetros un puntero a la variable donde se almacenará el resultado, la cantidad de bits que tendrá la parte entera, la cantidad de bits que tendrá la parte fraccionaria, y la cantidad de dígitos que se leen de parte entera, y cantidad de dígitos que se leen de parte fraccionaria.

```c
if (nBitsE + nBitsF > 15)
{
    return 0; // Devuelve error si la representación Q(nBitsE,nBitsF) no es válida para 16 bits
}
```

Primero evalúa si la representación Q(nBitsE,nBitsF) se excede de los 16 bits, ya que en este caso esta función utiliza variables de hasta 16 bits.
Luego se pide el ingreso mediante la función `pedirEntrada`

Función `pedirEntrada:`
Esta función se encarga de pedir la entrada del usuario imprimiendo el formato válido que se debe ingresar, y luego se pide ingresar el string.
Esta función recibe como parámetros el string, y la cantidad de dígitos que tendrá la parte entera de número y la cantidad de dígitos que tendrá la parte fraccionaria.

```c
void pedirEntrada(char *entrada, int16_t nDigE, int16_t nDigF)
{
    char formato[10];                           // Espacio suficiente para almacenar el formato
    sprintf(formato, " %%%ds", nDigE + nDigF + 2); // Construye la cadena de formato
    printf("\nIngrese un valor decimal ±");
    for (int i = 0; i < nDigE; i++)
    {
        printf("e");
    }
    printf(".");
    for (int i = 0; i < nDigF; i++)
    {
        printf("f");
    }
    printf("\n");
    scanf(formato, entrada); // Usa la cadena de formato generada
}
```

Primero se crea una cadena que contenga el formato que se debe utilizar para ingresar el string. Esto hace uso del `sprintf` que almacena en una variable un formato específico dado los parámetros ingresados. En este caso el formato es, la cantidad de dígitos enteros y fraccionarios sumando el caracter de punto que divide estos dos, y el caracter de signo al comienzo. De esta forma, el formato da seguridad que el usuario no podrá romper el ingreso por teclado ingresando un string que al almacenarlo ingrese a direcciones protegidas de memoria, y el formato es generalizado mediante `sprintf` para que sea válido para cualquier número que se quiera expresar.

Función `verificarEntrada:`
Al ingresar el string con la función anterior, se deberá verificar que esta entrada es válida.
Se procesan los caracteres del string.

```c
if ((entrada[0] == '-') || (entrada[0] == '+'))
{ // Si tiene un +- no se procesa ese caracter
    if (entrada[0] == '-')
    { // Caso de ser - se guarda flag
        *negativo = 1;
    }
    i = 1;
}
```

Primero se analiza si el primer caracter es de signo. En caso de que lo sea, este caracter no se procesa (*i=1*), y además si es negativo, se guarda una flag teniendo en cuenta que este número ingresado representa un número negativo. En caso que no sea un signo debe de ser un número, por lo tanto en el caso que no sea un número ni signo la función retorna 0 indicando que la entrada fue inválida.
Luego de estas condiciones, sabemos que si la entrada es válida, deberían de haber una cantidad de dígitos enteros dadas por la constante nDigE, una cantidad de digitos fraccionarios dados por nDigF, y que estos estarán separados por un único punto. Por lo que podemos procesar este string con un for y estas constantes.
Para simplificar un poco los datos del string hacemos uso de este struct

```
typedef struct
{
    int16_t posPunto;
    int16_t cantEnteros;
    int16_t cantFraccion;
} dataString;
```

Ahora hacemos un for, hasta el tamaño del string, y comenzando desde 0 si el string no tenía signo y era un número, o desde 1 si el string tenía signo.
Dentro del for analizamos cada caracter en la posición i del string, y tenemos en cuenta que solo pueden ser o un número, o un punto. Por lo que hacemos uso de esto

```
int16_t es_num = esNumero(entrada[i]);

if (!es_num)
{                                                   // No es numero
    if ((entrada[i] == '.') && (!(arreglo->posPunto))) // Único caracter válido es un punto
    {
        arreglo->posPunto = i; // Se tiene en cuenta posición del punto
    }
    else
    {
        printf("\nEntrada Invalida");
        printf("\nSolo puede haber un punto dentro del numero");
        return 0;
    }
}
else
{ // Es numero
    if (arreglo->posPunto)
    { // A partir de la posición del punto se analiza Enteros y Fracciones
        cantFraccion++;
    }
    else
        cantEnteros++;
}
```

Primero analizamos si el caracter es un número. Caso verdadero, analizamos si este número está antes o después del punto. Esto lo hacemos con un if que tiene como condición posPunto. Esta variable está inicializada en cero antes de entrar al for, hace referencia a la posición *i* en la que está el punto en el string y solo se actualiza una vez que se encuentra que en el string hay un punto. De manera que si el número que se analiza está antes o después del punto, se contabiliza de manera correcta teniendo en cuenta esta variable. Ahora en el caso que el caracter no sea un número, la única posibilidad es que sí o

sí sea un punto. Pero esto solo puede pasar una vez (no puede haber más de un punto en una expresión decimal). Con esto en cuenta, si el caracter no fue un número, y tampoco es un punto, entonces la entrada es inválida. Si es un punto, entonces se actualiza la variable posPunto con el valor de *i*, y si es un punto pero ya estaba actualizada esta variable, entonces la entrada es inválida. Luego de esto se hace una pequeña validación de la cantidad de dígitos de parte entera y parte fraccionaria se ingresaron para que respeten el formato.

Función `conversionValidacion_16:`
Esta función convierte el string que ya está validado, a el número real que se debera almacenar en 16 bits, mientras que valida que este número expresado esté dentro del rango válido para expresar.
Primero se evalúa que la representación que busca no exceda los 16 bits, ya que esta función utiliza 16 bits.
Para convertir el string a número, nos posicionamos en el caracter de punto, cuya posición actualizamos con posPunto anteriormente, y hacemos uso de dos for. Uno para convertir la parte entera y otro para convertir la parte fraccionaria.

```
for (i = arreglo->posPunto - 1; i >= fin; i--)
```

Desde el punto, retrocedemos un caracter y tenemos la unidad de la parte entera. La condición de corte de este for se da con fin = posPunto-cantEnteros de manera que el for pase por todos los digitos enteros, y no analice por ejemplo el signo en caso que el string lo tuviera.

```
for (i = (arreglo->posPunto) + 1; i < strlen(entrada); i++)
```

Desde el punto, avanzamos un caracter y tenemos el primer digito fraccionario, y este for avanza hasta llegar al fin del string.
En ambos casos, para convertir el caracter a número integer, hacemos uso de la diferencia entre el valor ascii de los caracteres y el valor numérico real, que es una diferencia de 48.
Veamos los dos for, primero el for para la parte entera.

```
valor = entrada[i] - 48; // Diferencia Ascii a Integer
resulEntero = resulEntero + multiplicador * valor;

if (((resulEntero > ((1 << nBitsE) - 1)) && (!negativo)) || (resulEntero > (1 << nBitsE)))
{ // Validación de rango
    return 0;
}

multiplicador = multiplicador * 10;
```

Primero obtenemos el valor con la diferencia de 48, luego como sabemos que con cada iteración avanzamos de unidad a decena y luego a centenas, podremos hacer uso de esto teniendo un multiplicador inicializado en 1 y con cada iteración se multiplique por 10. De esta forma podemos convertir los caracteres que representan la parte entera como una suma con un multiplicador. Unidad * 1 + Decena * 10 + Centena * 100.
For para la parte fraccionaria.
En este caso, no es tan simple la conversión. Si se hiciera uso de la lógica usada anteriormente, se llegaría a un valor erróneo. Esto se da ya que la lógica anterior nos serviría si la parte fraccionaria se tratara de un número entero, pero se trata de la fracción. Por lo que debemos modificar la solución agregando un paso más.

```
multiplicador = 1000;
int16_t conversion = 0;
for (i = (arreglo->posPunto) + 1; i < strlen(entrada); i++)
{
    valor = entrada[i] - 48; // Diferencai Ascii a Integer
    conversion = conversion + multiplicador * valor;
    multiplicador = multiplicador / 10;
}
```

Primero utilizamos la misma lógica que para la parte entera, pero en vez de comenzar con la unidad, sabemos que nuestro formato admite hasta 4 dígitos fraccionarios. Por lo que comenzamos haciendo valor * 1000 + valor * 100 + valor* 10 + valor * 1. Esto se hace para que siempre se diferencie entre un ingreso como 1.05 y 1.5 ya que la parte fraccionaria de cada uno es 1.05 -> 500 y la parte fraccionaria de 1.5 -> 5000 evitando errores en la conversión siguiente.

```
int16_t resulFraccion = 0;
for (i = nBitsF - 1; i >= 0; i--) // Conversión de Fracción a Binario
{
    conversion = conversion << 1;
    if (conversion >= 10000)
    {
        resulFraccion = resulFraccion | 1 << i;
        conversion = conversion - 10000;
    }

    if (resulFraccion > ((2 << nBitsF) - 1))
    { // Validación de rango
        return 0;
    }
}
```

Queremos convertir nuestro número entero en el equivalente en binario que reside en la parte fraccionaria. Para esto normalmente, se multiplica por 2 un número fraccionario como lo es 0,8 y la parte entera resultante es la cadena de bits que se almacena en la parte fraccionaria. Como nosotros no podemos usar float ni double, escalamos esta técnica de manera que se multiplica en vez de 0,8 -> 8000 y se analiza si se excede de 10000. Para esto hacemos en cada iteración un desplazamiento hacia izquierda que equivale a multiplicar por 2, analizamos si excede 10000, caso verdadero, se hace un OR a una variable inicializada en cero, con un 1 desplazado el número de iteración. Esto nos sirve ya que en el caso que en la primera iteración resulta que excede 10000, quiere decir que el bit más significativo de la parte fraccionaria necesita estar en 1. Por lo tanto desplazar 1 a unas 3 posiciones (asumiendo en este caso que nBitsF es igual a 4) nos daría 1000, que es la máscara que buscamos para hacer un OR con la variable inicializada en cero.

No se mencionó que antes del for explicado anteriormente, se valida un caso límite del rango. Dentro de cada for de parte entera y parte fraccionaria se evalúa que el rango sea válido dadas las variables y constantes del programa por separado. El caso límite es que si el número ingresado por teclado fue el número más negativo posible, la parte fraccionaria

debería de valer cero. Por lo tanto se agregó esta condición antes de convertir la parte fraccionaria.

```
if (negativo && (conversion != 0) && (resulEntero ==128)){
    return 0;
}
```

Esta condición nos dice que si el número ingresado fue negativo, y el resultado entero fue 128 (más negativo posible), entonces la parte fraccionaria debería de ser cero. Por lo tanto, la conversión de string a integer debería de ser cero en la parte fraccionaria, incluso antes de pasar por el for siguiente que convierte el valor integer a binario.
Por último, se juntan ambas partes en una misma variable, y se tiene en cuenta el complemento a 2 si el número ingresado fue negativo, negarlo y sumarle su resolución.

```
*resultado = (int16_t)resulEntero;
*resultado = *resultado << nBitsF;
*resultado = *resultado | resulFraccion;

if ((negativo) && (resultado != 0))
{
    *resultado = ~(*resultado);
    (*resultado)++;
}

return 1;
```

Punto g

Función `printInDecimal_32:`
Esta función toma un número de 32 bits (resul) que está representando en formato Q(nBitsE, nBitsF), su objetivo es mostrar ese número en formato decimal.
Primero se verifica que el número total de bits no exceda los 31 bits, ya que el bit 32 se usa como bit de signo.

```
if (nBitsE + nBitsF > 31)
```

Luego mediante máscaras y operaciones lógicas nos quedamos con la respectiva parte entera y fraccionaria del número en base a su representación Q. La máscara para la parte entera la representamos mediante un 1 desplazado a izquierda tantas veces como cantidad de bits para la parte fraccionaria , todo eso menos 1. De esta forma tenemos todos bits en 1 en la parte fraccionaria. Por otro lado para la máscara de la parte entera , hacemos la resta entre una máscara con todos bits en 1 y la máscara para la parte fraccionaria calculada anteriormente.

```
mascaraF = (1 << nBitsF) - 1;
mascaraE = 0xFFFFFFFF - mascaraF;
```

Si el número es negativo imprimimos el menos , luego negamos el número por estar en CA2 y le sumamos 1 (Estariamos sumandole la resolución). Haciendo esto estamos trabajando

con el signo y el módulo por separado para poder mostrar el número decimal que representa adecuadamente..

```
if ((aux & mascaraSigno) != 0)
{
    printf("-"); // Imprimo el menos
    // Invierto los bits por ser CA2 y al ya haber imprimido el signo  en vez de restarle la resolucion se la sumo (trabajo con modulos)
    aux = ~aux + 1; // Sumo uno ya que como tengo toda la variable en 16 bits al sumarle 1 es como que le estoy sumando la resolucion
    // Ya tengo el valor en BSS para poder imprimir
}
```

Ahora imprimimos la parte entera por un lado , y luego la parte fraccionaria. Para la parte entera basta con imprimirlo interpretándolo como decimal, pero para la fraccionaria debemos imprimir digito a digito , ya que sino podríamos tener números que tengan ceros adelante y no se vean impresos(Ej 1.05 podria interpretarse como 1.5). Para esto la tomamos como un valor entero y calculamos una aproximación decimal multiplicando por 10, 100, 1000, etc. y dividiendo por 2^nBitsF para obtener los primeros dígitos decimales, que se imprimen uno a uno.

```
for (size_t i = 1; i < 5; i++, j = j * 10)
{
    nroImprimir = (parteFraccionaria * j / (1 << nBitsF)) % 10; // Imprimo uno a uno los digitos de la parte fraccionaria
    printf("%d", nroImprimir);
}
printf("\n");
```

Punto h

Constantes:

```
 3
 4   #define cantDigF 4
 5
 6   #define valorMaxX 0x7f800000
 7   #define valorMinX 0x80800000
 8
 9   #define cantDigE_m 1
10   #define nBitsE_m 0
11   #define nBitsF_m 15
12
13   #define cantDigE_b 3
14   #define nBitsE_b 7
15   #define nBitsF_b 8
16
17   #define cantDigE_x 5
18   #define nBitsE_x 15
19   #define nBitsF_x 16
20
```

- **cantDigF** -> Cantidad de dígitos decimales fraccionarios a ser leídos para el ingreso de todos los números
  .
- **valorMaxX** -> El valor máximo de **x** expresado en hexadecimal (Se halló su valor en el informe).
- **valorMinX** -> El valor mínimo de **x** expresado en hexadecimal (Se halló su valor en el informe).

- **cantDigE_m** -> Cantidad de dígitos decimales enteros a ser leídos en el ingreso de **m**.
- **nBitsE_m** -> Cantidad de bits asignados a la parte entera de **m**.
- **nBitsF_m** -> Cantidad de bits asignados a la parte fraccionaria de **m**.

- **cantDigE_b** -> Cantidad de dígitos decimales enteros a ser leídos en el ingreso de **b**.
- **nBitsE_b** -> Cantidad de bits asignados a la parte entera de **b**.
- **nBitsF_b** -> Cantidad de bits asignados a la parte fraccionaria de **b**.

- **cantDigE_x** -> Cantidad de dígitos decimales enteros a ser leídos en el ingreso de **x**.
- **nBitsE_x** -> Cantidad de bits asignados a la parte entera de **x**.
- **nBitsF_x** -> Cantidad de bits asignados a la parte fraccionaria de **x**.

Función `normalizar_16_32`:

```
75
76  int32_t normalizar_16_A_32(int16_t n, int16_t nBitsE_16, int16_t nBitsF_16, int16_t nBitsE_32, int16_t nBitsF_32)
77  {
78      // Al realizar la operación redudante 'n % 0xFFFF' la variable 'n' de 16 bits se carga en 'resultado' de 32 bits
79      // con el resto de bits superiores al bit de posición 16 en 0
80      int32_t resultado = n & 0xFFFF;
81
82      // El desfasaje entre las representaciones en punto fijo Q(nBitsE_16,nBitsF_16) y Q(nBitsE_32,nBitsF_32) se puede
83      // escribir como 'nBitsF_32 - nBitsF_16'. Al desplazar 'n' dentro de 'resultado' acomodamos las partes fraccionaria
84      // y entera a la nueva representación
85      resultado = resultado << (nBitsF_32 - nBitsF_16);
86
87      // Habiendo hecho el desplazamiento, queda ahora "acomodar" el bit de signo, procedemos a aplicar el
88      // método explicado en el informe
89      int32_t mascara = (0x7FFFFFFF) - ((1 << (nBitsF_32 + nBitsE_16)) - 1);
90
91      resultado = (resultado + mascara) ^ mascara;
92
93      return resultado;
94  }
95
```

Parámetros:

- **n** -> Número de representación Q(**nBitsE_16**,**nBitsF_16**) en 16 bits a normalizar a Q(**nBitsE_32**,**nBitsF_32**) en 32 bits.
- **nBitsE_16** -> Número de bits de n asignados a la parte entera.
- **nBitsF_16** -> Número de bits de n asignados a la parte fraccionaria.
- **nBitsE_32** -> Número de bits enteros que tendrá n en la nueva representación.
- **nBitsF_32** -> Número de bits fraccionarios que tendrá n en la nueva representación.

Ésta función implementa la lógica ya explicada en el informe para la normalización de **m** y **b** de sus representaciones Q(0,15) y Q(7,8), respectivamente, a Q(15,16).

Pasos:

1- Línea 80
Comienza cargándose la variable **n** de 16 bits en la variable **resultado** de 32 bits

2- Línea 85
La variable **n** se desplaza a izquierda dentro de resultado la cantidad de lugares que le faltan a la posición de la coma en Q(**nBitsE_16**, **nBitsF_16**) con respecto a la posición de la coma en Q(**nBitsE_32**, **nBitsF_32**). Siendo esa cantidad de lugares:

$$nBitsF32 \; - \; nBitsF16 \; = \; desplazamiento$$

3- Línea 89
Se genera la máscara propuesta. Esta variable **mascara** tiene únicamente valor 1 en los bits que corresponden a “la nueva parte entera” de la representación Q(**nBitsE_32**,**nBitsF_32**).

4- Línea 91
Finalmente, realizamos el conjunto de operaciones de la cadena de bits **resultado** junto con **mascara** y obtenemos el número **n** normalizado a 32 bits dentro de **resultado**.

Función `multiplicacion_32:`

```
96  int32_t multiplicacion_32(int32_t a, int32_t b, int16_t nBitsF_32)
97  {
98      // Multiplicación de 64 bits
99      int64_t temp = (int64_t)a * (int64_t)b;
100
101     // Ajustar la escala de vuelta a 32 bits
102     temp = temp >> nBitsF_32;
103
104     // Truncar a 32 bits
105     return (int32_t)temp;
106 }
```

Parámetros:

- **a** y **b** -> Números a multiplicar. Ambos de una misma representación.
- **nBitsF_32** -> Número de bits de la parte fraccionaria de la representación de **a** y **b** (ambos han de estar en una misma representación).

Pasos:

1- Línea 99
Se realiza la multiplicación con un casteo a int64_t, el cuál realiza el corrimiento de signo. Este resultado se guarda dentro de una variable temporal **temp**.

2- Línea 102
Como la multiplicación de dos números de 32 bits da un resultado de 64 bits con la parte fraccionaria siendo de 2 * **nBitsF_32** bits, desplazamos el resultado **nBitsF_32** veces a la derecha. Con esto nos deshacemos de esa parte fraccionaria que ocupa más de los **nBitsF_32** bits fraccionarios de la representación de **a** y **b**.

3- Línea 105
Se castea el número dentro de **temp** a int32_t y se lo retorna. Este casteo hace que el bit de signo se mueva al bit más significativo dentro de los 32 bits.