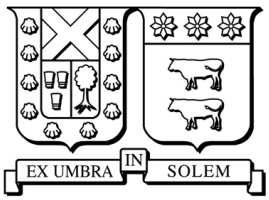

UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE ELECTRÓNICA

LECTURA 4
REPRESENTACIÓN DE NÚMEROS EN PUNTO FIJO Y FLOTANTE.

CURSO	LABORATORIO DE PROCESAMIENTO DIGITAL DE SEÑALES
SIGLA	ELO–385
PROFESOR	RODRIGO HUERTA CORTÉS
AYUDANTE	ALEJANDRO HERRERA

Valparaíso, 14 de Abril de 2003

INTRODUCCIÓN

En esta lectura consideraremos la representación de números para cálculos digitales. La principal característica de la aritmética de digital es el número limitado (generalmente fijo) de dígitos usados para representar los números. Esta restricción lleva a que la precisión numérica sea finita, y con ello los cálculos, creando así, problemas de redondeo y efectos no lineales en el funcionamiento de los algoritmos implementados. Un ejemplo clásico de problemas de representación finita se da en los filtros IIR de alto orden. En ellos un coeficiente podría ser cuantificado o redondeado de forma tal que quedara fuera de la circunferencia unitaria, haciendo que se inestabilice.

La representación de números en un formato de punto fijo es una generalización de la familiar representación decimal de un número como una cadena de dígitos con un punto decimal. En esta notación, los dígitos a la izquierda del punto representan la parte entera del número, y los dígitos a la derecha del punto decimal, representan la parte fraccional del número.

Las secciones siguientes entregan una introducción a la representación de números decimales tanto en representación de punto fijo como en representación de punto flotante.

Además se entregan algunas consideraciones necesarias para trabajar correctamente con el codec de la tarjeta DSK TMS320C6711.

1 REPRESENTACIÓN DE NÚMEROS EN PUNTO FIJO

1.1 REPRESENTACIÓN EN COMPLEMENTO 1

Complemento 1 es una forma particular de representar números positivos y negativos. Su forma es simple y bastante directa de entender. Todo número positivo posee su bit más significativo igual a 0. Los números negativos se obtienen con sólo negar (o complementar) el número positivo correspondiente. Un ejemplo para el caso de tres bits es mostrado en la tabla 1.

Tabla 1: Números binarios en complemento 1

000	0	111	0
001	1	110	–1
010	2	101	–2
011	3	100	–3

De la tabla anterior se puede observar que el número 0 posee dos formas distintas de representación. Ello ha llevado a problemas con la comparación por cero en algoritmos, por lo cual este tipo de representación no es utilizado.

El término "complemento 1" se debe a que el número negativo se obtiene sólo complementando el patrón de bits del número positivo que se quiere pasar a negativo.

1.2 REPRESENTACIÓN EN COMPLEMENTO 2

La representación en complemento 2 es una forma eficiente de representar números con signo en microprocesadores de punto fijo. La propiedad fundamental de este formato es que permite representar números negativos, por lo cual se utiliza el bit más significativo de la palabra binaria que se está manejando. Esto lleva a que en un formato de palabra de n bits, la capacidad de representación sea de hasta $2^{n-1}-1$ para números positivos y 2^{n-1} negativos. El bit de signo será siempre el más significativo. La figura 1 muestra un esquema para n bits.

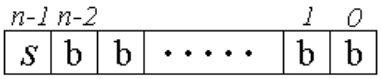

Fig.1: Palabra de n bits: *S*: bit de signo, b: bits que representan un número

Con la forma de representación antes mostrada los números positivos tendrán el bit más significativo igual a 0. Si es un número negativo éste bit será 1.

Un número positivo escrito en complemento 2 tendrá la misma representación que su equivalente en un formato sin signo (claro que el bit más significativo deberá ser 0). Por otro lado, un número negativo escrito en complemento 2 no tiene un equivalente directo en un formato sin signo. Esto quiere decir que con sólo agregar un 1 al bit más significativo de un número positivo no se logra representar un número negativo.

Para llegar a un número negativo en complemento 2 es necesario realizar las operaciones siguientes: al número positivo que se quiere convertir a negativo se le debe representar en formato binario, luego complementar cada bit (pasar los 1s a 0s y viceversa) y finalmente sumar 1. La tabla siguiente muestra un ejemplo para el caso de tres bits.

Tabla 2: Números binarios de 3 bits en complemento 2

Nº Positivo			Nº Negativo
1 = 001	110	110 + 1	111 = –1
2 = 010	101	101 + 1	110 = –2
3 = 011	100	100 + 1	101 = –3
4 = 100	011	011 + 1	100 = –4

Nota: en la tabla anterior el número 100b no representa 4 sino que –4. Esto se debe a que el bit más significativo es 1. Ello lleva a que siempre exista un número negativo más que positivo.

Si se piensa en un número de 16 bits (como es el caso de la palabra binaria del codec de la tarjeta DSK TMS320C6711) el rango de números a representar es de: [-32768 : 32767]. La tabla 3 resume los rangos máximos que se pueden manejar con distintos tipos de largo de palabra.

Tabla 3: Rangos de representación para diferentes largo de palabra.

n	Min	Max
8	–128	127
16	–32.768	32.767
24	–8.388.608	8.388.608
32	–2.147.483.648	2.147.483.648

Para calcular el número decimal que se está representando en forma binaria se puede utilizar (1), refiriéndose a la figura 1:

$$x = -s \cdot 2^{n-1} + \sum_{k=0}^{n-2} b_k \cdot 2^k \qquad (1)$$

donde cada factor b_k es 0 ó 1

1.3 Representación Fraccionaria de Números en Formato de Punto Fijo

Hasta ahora se ha visto que un número en representación binaria puede ser interpretado como positivo o negativo. Sin embargo, en punto fijo es necesario poder operar con números fraccionarios también. Para ello es posible utilizar una forma muy simple de representación.

1.3.1 FORMATO NUMÉRICO DIGITAL $Q_{M.N}$ (Q_N)

En un formato $Q_{m.n}$ se utilizan *m* bits para representar en complemento 2 la parte entera de un número y *n* bits para representar en complemento 2 la parte fraccionaria. Son necesarios $m + n + 1$ bits para almacenar un número en formato $Q_{m.n}$. El bit extra es usado para almacenar, en la posición más significativa, el signo del número. El rango entero representable es $(-2^m, 2^m - 2^{-n})$ con una resolución de 2^{-n}.

Para el caso de un sistema digital de 16 bits y su uso con una representación numérica con de formato $Q_{4.12}$, o Q_{12}, se utilizan 3 bits para representar números enteros, 12 para decimales y 1 bit de signo.

S	***E***	***E***	***E***	***d***	***d***	***d***	***d***	***d***	***d***	***d***	***d***	***d***	***d***	***d***	***d***

(2)

La resolución del formato es:

$$\frac{1}{2^{12}} = 0.000244140625 \tag{3}$$

La correspondencia entre la magnitud *interpretada* y el valor de 16 bit almacenado en el DSP es la siguiente:

$$(-8 : 7.99975585938) \Leftrightarrow (-32768 : 32767) \tag{4}$$

o en formato hexadecimal

$$(-8 : 7.99975585938) \Leftrightarrow (-8000 : 7FFFh) \tag{5}$$

Gráficamente lo anterior es representado en la figura 2 de la siguiente manera:

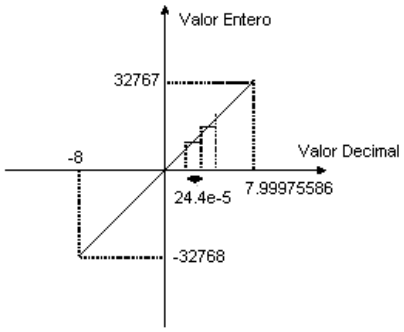

Figura 2. Correspondencia entre valores decimales y enteros para representación digital. En formato Q_{12}.

Reglas para la operación con números en este tipo de notación:

1.- Suma de cantidades con *m* bits decimales $$Q_m + Q_m = Q_m \tag{6}$$

2.- Multiplicación de cantidades con *m* y *n* bits decimales

$$Q_m \cdot Q_n = Q_{m+n} \tag{7}$$

Por ejemplo, la multiplicación de 2 números en formato Q_{12} se obtiene de la siguiente forma:

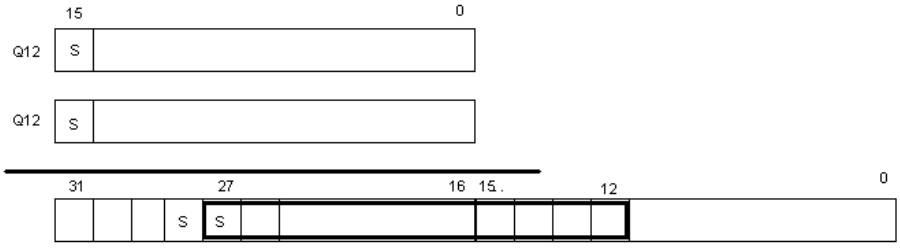

Figura 3. Multiplicación de números en formato Q_{12}.

Dado que la multiplicación suma el número de cifras decimales, el resultado puede y debe ser truncado para poder representarlo en los 16 bits de un DSP. Los 12 primeros bits, los menos significativos, se eliminan quedando sólo los 16 bits más significativos que representan el resultado de la multiplicación. Cabe destacar que todo procesador con una unidad independiente de multiplicación posee un registro de 2*n* bits para almacenar el resultado de una operación. Además algunos incluyen 1 o hasta 3 bits extra para manejar posibles overflow.

Se debe tener especial cuidado con la multiplicación, ya que es fácil que ocurran overflow debido a la operación con números grandes, por ejemplo:

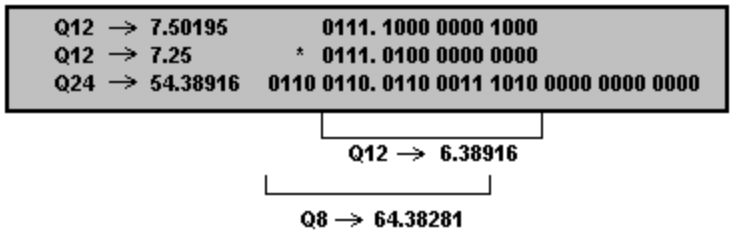

Figura 4: Problemas de representación luego de una operación de multiplicación.

En la figura 4, el resultado Q_{12} es erróneo, mientras que el resultado Q_8 no es del todo preciso. Esto se debe a que el resultado no es representable en el formato Q_{12}. Por ello es importante tener claro el tipo de cantidades que se operan y los rangos que éstas ocupan. El resultado en Q_8 es impreciso por la cantidad de bits decimales que actúan como cifras significativas que éste puede representar (2^{-8} = 0,00390625).

A modo de ejemplo de la necesidad de tener en cuenta que se está trabajando con números de punto fijo, observe el siguiente extracto de código. En él se han definido variables de tipo short para la lectura de datos (desde el codec) y los coeficientes del filtro FIR también son de tipo short. Sin embargo, la variable yn, que almacena el resultado de la multiplicación y acumulación, es de tipo integer (32 bits).

```
int yn = 0;                          Declaración de variables
short dly[N];
short h[N];
interrupt void c_int11()             Inicio de rutina de interrupción
  {
        short i;
        dly[0] = input_sample();     Lectura de dato desde codec
        yn = 0;
        for (i = 0; i< N; i++)
        yn +=(h[i]*dly[i]);          Multiplicación y acumulación
        for (i = N-1; i > 0; i--)
                dly[i] = dly[i-1];   Corrimiento de datos en buffer
        output_sample(yn >> 15);     Eliminación por corrimiento de los bits de
                                     precisión y posterior envío al codec.
        return;                      Retorno de la interrupción
  }
```

Cada vez que se multiplican dos variables de tipo short es necesario un buffer o variable de almacenamiento, 32 bits para este caso. Al final de la operación de multiplicación y acumulación característica de los filtros FIR se toma el resultado obtenido y se corre en 15 posiciones para sólo entregar los bits más significativos del resultado. Notar que el programador sabe que el manejo de los datos fue hecho en formato Q_{15} y de ahí que el resultado de la multiplicación sea corrido en 15 posiciones a la derecha. El resto es ignorado ya que sólo representan bits de precisión que para el caso no son útiles.

2 REPRESENTACIÓN DE NÚMEROS EN PUNTO FLOTANTE

La representación en punto flotante posee una gran ventaja sobre los números de punto fijo: escalar las variables no es una preocupación. Un número de punto flotante puede ser representado usando precisión simple (SP) con 32 bits, o doble precisión (DP) con 64 bits, cómo se muestra en la figura 5(a).

En un formato de precisión simple, el bit 31 es el signo, los bits 30 al 23 representan el exponente y los bits 22 al 0 representan los bits decimales. Números tan grandes como 10^{-38} y tan grandes como 10^{+38} pueden ser representados.

En formato de precisión doble, más bit para el exponente y los decimales están disponibles, como se muestra en la figura 5(b). Un flotante de 64 bits debe ser formado a partir de dos registros de 32 bits. El primer registro con sus bits 31 al 0 son sólo para decimales, los bits 19 al 0 del segundo registro también son para decimales, posiciones 30 a la 20 se utilizan para exponente y el bit de signo está en la posición 31, siempre del segundo registro. Como resultado, números tan pequeños como 10^{-308} y tan grandes como 10^{+308} pueden ser representados.

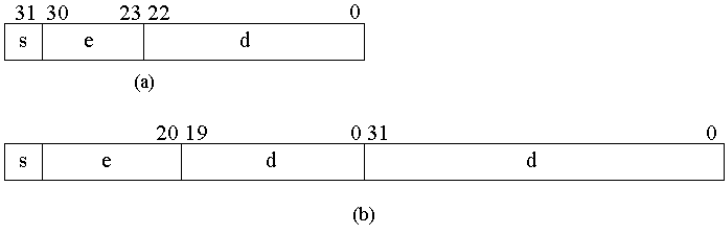

Figura 5: Formato de tipos flotantes

3 CONSIDERACIONES NECESARIAS PARA TRABAJAR CON EL CODEC EN LA TARJETA DSK

Como fue visto en la experiencia 2, el CODEC TLC320AD535 trabaja con un formato numérico de 16 bit. Esto presenta ciertas dificultades que deben ser observadas al momento de leer datos y escribir datos desde y hacia el CODEC.

3.1 LECTURA DE DATOS

Cada vez que el CODEC realiza una conversión A/D el resultado es enviado al DSP, donde es tomado desde el puerto serial y dispuesto en una variable. Es este momento donde se debe tener claro qué es lo que se desea hacer. Primero es necesario tener en cuenta qué tipo de variable física se está leyendo.

En términos generales, el CODEC sólo realiza conversiones A/D y entrega su resultado en un formato complemento 2, lo cual significa que envía valores que pueden ser interpretados como números positivos o negativos. Ello dice que el valor leído desde el codec debiera ser almacenado en un tipo de variable que contemple el signo. Una buena elección es un tipo *integer* (int). Ahora bien, si se asigna a una variable de tipo *int* se deberá tener en cuenta que el dato de 16 bit que entregó el CODEC ahora es de 32 bits, por lo cual la nueva variable tendrá un largo de palabra de 32 bit con los 16 bits superiores llenos de ceros. Algo similar ocurre cuando la variable utilizada es de tipo *flotante* (float). Ello hace que la variable float almacene el signo y la magnitud que posee el dato enviado por el CODEC. Recordar que el rango de salida de los datos del CODEC es (-32768 : 32767).

3.2 MANEJO DE DATOS

A raíz de la lectura de datos se deben tener en cuenta una serie de consideraciones que eviten problemas a la hora de empezar a manejar y operar los datos adquiridos. ¿Pero cuál es el problema con todo lo anterior?

Suponga que desea hacer un algoritmo que cree una señal AM a partir de una portadora generada con un oscilador digital y una señal modulante proveniente del CODEC. Una señal modulada en amplitud tiene una forma general dada por (8).

$$S(t) = A_c\left[1 + mf(t)\right]\cos(w_c t) \tag{8}$$

donde $f(t) \leq |1|$ es la señal modulante adquirida desde el CODEC.

Es claro que si el CODEC justo realizó una conversión que arrojó un número "grande" como por ejemplo 12.389, la suma con 1 no tendrá sentido. Una condición necesaria es que la función o señal *f(t)* esté en el rango [–1 y 1]. Ello implica un necesario escalamiento de la variable antes de ser operada; y más aún, conocer y entender la naturaleza de los datos que están siendo manejados y operados para así asociar correctamente la variable física medida con su manejo digital dentro del procesador.

Por otro lado, si se quiere manejar números fraccionarios, lo más simple es primero transformar la variable que contiene el dato adquirido a un tipo que resista fracciones. Recordar que un tipo entero no puede manejarlas. El siguiente código ejemplifica la transformación y posterior escalamiento utilizando tipos flotantes:

```
Float a, c;

Main
{
 a = (float) McBSP_read( );      Cast de integer a flotante.
 a = a / 32767;                  Escalamiento al rango [-1:1]
 ...
 c = 1.43 * a;                   Operación en números flotantes
 ...
 ...
}
```

La arquitectura del DSP TMS320C6711 está diseñada para operar a nivel de hardware con números flotantes (de hecho, puede realizar hasta 900 MFLOPS @ 150MHz clock) por lo que operaciones con números flotantes no implica introducir extensos códigos que manejen tipos flotantes a partir de una arquitectura de punto fijo (notar que ello lo realiza el compilador y es prácticamente transparente para el programador).

Sin perjuicio de lo anterior, es posible operar todo en tipo integer o short. Esto obliga a tener especiales cuidados en el manejo de los datos. Un tipo integer necesariamente deberá ser manejado teniendo en cuenta que las operaciones matemáticas como multiplicaciones devuelven resultados de $2n$ bits cuando los dos operandos son de n bits. Los números fraccionarios no son representables ni en los tipos integer ni short, por lo cual deben ser traspasados a algún formato Q e interpretados de acuerdo a ello. Una consideración necesaria es no sumar números en distinto formato Q ya que no tendrá sentido su resultado. Ello no ocurre con la multiplicación que, como fue visto anteriormente, entregará un número con $m+n$ bits decimales. De acuerdo al corrimiento efectuado será el formato del resultado final.

3.3 ESCRITURA DE DATOS

Una vez que ya se cuenta con el dato necesario para ser enviado al CODEC (o a algún tipo de conversor D/A) se debe llevar a un formato numérico de 16 bits. El compilador utilizado asigna a un tipo short 16 bits, por lo cual es necesario, si se ha trabajado con tipos float o int, realizar un cast. Cuando se trabaja con números dentro del rango [-1:1) se deberá

amplificar el dato *antes* de realizar el cast. Eso es necesario ya que si no se hace en ese orden se perderá precisión. Por ejemplo:

Suponga una variable de tipo float con un valor de 0,85. Cada vez que se haga un cast a tipo short para enviar el dato por el codec el valor asignado a la variable short será 0 ó 1 según sea el redondeo propio del número por parte del compilador. Lo mismo pasará con cualquier otro valor y además no se estará utilizando todo el rango de 16 bits para la salida del CODEC. Ello obliga a amplificar los datos procesados para no perder la precisión.

3.4 DOS EJEMPLOS PRÁCTICOS

3.4.1 OSCILADOR DIGITAL DE 1 KHZ UTILIZANDO TIPO SHORT Y FORMATO Q_{14}.

Sea (9) la ecuación de diferencias para un oscilador digital.

$$y(n) = Ay(n-1) - y(n-2) \tag{9}$$

La constante A para una oscilación de 1 KHz será: $A = 2\cos\left(2\pi\frac{1000}{8000}\right) = 0.7071$. Según se vio en la sección 1.3.1, los números decimales, y en particular aquellos menores de 1, no son representables en punto fijo en forma directa. Para este caso se escogió un formato Q_{14}. Este permite que una buena cantidad de decimales sean representados y evitará el riesgo de overflow en las multiplicaciones. El nuevo valor de A en la representación Q_{14} es: $A = 0.7071 \cdot 2^{14} = 11585$.

Observar el código siguiente:

```
short y[3] = {0,11585,0};                  Declaración de variables y asignación de
const short A = 11585;                     condiciones iniciales
int n = 2;

interrupt void c_int11()                   Inicio de rutina de interrupción que calcula el
{                                          tono de 1[KHz]
 y[n] = (((int)A*(int)y[n-1])>>14) -       Algoritmo de ecuaciones de diferencia.
y[n-2];                                    Multiplicación e inmediato corrimiento
 y[n-2] = y[n-1];
 y[n-1] = y[n];                            Salida del dato calculado a través del codec.
 output_sample(y[n]);                      Notar que no se escala
 return;
}
                                           Inicialización de codec y McBSP
void main()                                Loop infinito
{
 comm_intr();
 while(1);
}
```

Observar que en la multiplicación de las variables A e y[n-a] se ha realizado previamente un cast y luego se ha corrido en 14 posiciones el resultado. Recordar que multiplicar 2 números en formato Q_{14} entrega un número en formato Q_{28}, y para volver al formato original se deben eliminar los bits menos significativos del resultado. Este tipo de consideraciones se deben tener presente al momento de escoger tanto una plataforma de desarrollo como el tipo de datos que se usarán para la implementación de un algoritmo.

3.4.2 OSCILADOR DIGITAL DE 1 KHZ UTILIZANDO TIPO FLOAT

Esta vez el código es el mismo salvo en los tipos utilizados y la forma de manejar los números y la manera de aplicarlos. Como los números en punto flotante FLOAT pueden representar números decimales, esta vez es posible sólo entregar los resultados directamente, vale decir, la variable $A = 0.7071$ no debe ser escalada y las condiciones iniciales tampoco. Con esto en mente el código queda así:

```
Float y[3] = {0,7071,0};                Declaración de variables y asignación de
const float A = 0.7071;                 condiciones iniciales
int n = 2;
interrupt void c_int11()                Inicio de rutina de interrupción que calcula el tono
{                                       de 1[KHz]
 y[n] = A* y[n-1] - y[n-2];             Algoritmo de ecuaciones de diferencia
 y[n-2] = y[n-1];
 y[n-1] = y[n];

output_sample((short)32767*y[n]);       Salida del dato calculado a través del codec,
 return;                                previamente escalado y "casteado" a tipo short.
}

void main()
{
 comm_intr();                           Inicialización de codec y McBSP
 while(1);                              Loop infinito
}
```

Se observa inmediatamente que el tipo float facilita las cosas cuando se trata de multiplicar o dividir datos, ya que la aritmética de punto flotante se encarga de realizar todas las operaciones anexas para que los resultados sean los correctos.
Notar que para la salida del dato por el codec se requiere el "casteo", según se vio en la sección 3.3. El valor 32767 no es arbitrario ya que éste se encuentra dentro de representación de 16 bits contemplando que la mitad es para números negativos y la otra para positivos, en complemento 2. Así el rango análogo que maneja el codec será completamente cubierto ya que la variable y[n] está en el rango [-1 : 1].

Una mención especial: El Code Composer puede manejar diferentes opciones de compilación. Una de ellas permite que las operaciones de punto flotante sean hechas a nivel de hardware, bien con las unidades de punto fijo o con las de punto flotante. Por ejemplo, si no se escoge (configura) que las operaciones de punto flotante se realicen en las unidades

del mismo tipo, las multiplicaciones se harán a través de código almacenado en una biblioteca RTS en vez de hacerla con la instrucción especializada MPYSP.

4 TIPOS DE DATOS

1. *Short*: 16 bits representados en complemento 2.
   Rango $[-2^{15} : 2^{15}-1]$.

2. *Int* o *signed int*: 32 bits representados en complemento 2.
   Rango $[-2^{31} : 2^{31}-1]$.

3. *Float*: representados en formato IEEE de 32 bits.
   Rango $[2^{-126} = 1.175494 \times 10^{-38} : 2^{+128} = 3.40282346 \times 10^{+38}]$.

4. *Double*: representados en formato IEEE de 64 bits.
   Rango $[2^{-1022} = 2.22507385 \times 10^{-308} : 2^{+1024} = 1.79769313 \times 10^{+308}]$.

Datos de tipo short para operaciones en punto fijo pueden ser más eficientes (menos ciclos) que los tipos integer.

5 CONSIDERACIONES ESPECIALES

Se debe poner especial atención al tamaño tipo de dato cuando se escribe un código. El compilador C6000 define un tamaño para cada tipo de dato (con y sin signo):

Tipo de Datos	Cantidad de Bits
Char	8 bits
short	16 bits
int	32 bits
long	40 bits
float	32 bits
double	64 bits

Basado en el tamaño de cada tipo de dato, es conveniente seguir las siguientes consideraciones al escribir un código:

- No asumir que los tipos INT y LONG son del mismo tamaño porque el compilador C6000 asume un tamaño de 40 bits para los tipos LONG. Esto puede añadir instrucciones extra, así como limitar la selección de unidades funcionales de hardware.
- Para multiplicación de punto fijo usar tipos SHORT cada vez que sea posible ya que este tipo utiliza más eficientemente el multiplicador de 16 bits (hardware) del procesador (1 ciclo para short * short v/s 5 ciclos para int * int).
- Usar tipos INT o UNSIGNED INT para contadores en lazos, en vez de tipos SHORT o UNSIGNED SHORT para evitar instrucciones de extensión de signo innecesarias.