Numbering Systems

Introduction

One of the most important factors in embedded control is determining the computing time for a given task. Because embedded control has to cope with its tasks in a given and fixed amount of time, we call this “Real-Time Computing”. And, as you know, time goes very quickly. If the device is also responsible for control actions, such as sampling sensor signals, deviation control and adjusting actuator output signals then the term “Real-Time Control” is used.

Therefore, one of the characteristics of a processor is its ability to do mathematical calculations in an optimal and efficient way. In recent years, the size of mathematical algorithms that have been implemented in embedded controller units has increased dramatically. Just take the number of pages for the requirement specification for one of the various electronic control modules for a passenger car:

- 1990: 50 pages,
- 2000: 3100 pages (Source: Volkswagen AG)

So, how does a processor operate with all these mathematical calculations? And, how does the processor access and process data?

You probably know that the ‘native’ numbering scheme for a digital controller is binary numbers. Unfortunately, all process values are either in the format of integer or real numbers. Depending on how a processor deals with these numbers in its translation into binary numbers, we distinguish between two basic types of processor core:

- Floating-Point Processors
- Fixed-Point Processors

This chapter will start with a brief comparison between the two types of processor.

After a brief discussion about binary numbers, we will then look into the different options to use the Fixed-Point unit of the F2833x. It can perform various types of mathematical operations in a very efficient way, using only a few machine clock cycles.

However, most of today’s numerical simulation systems, such as MATLAB and Simulink from The Mathworks Corp., operate on Floating-Point numbers. If such a simulation project is later implemented in a Fixed-Point microcontroller, a set of library functions is used to operate on Floating-Point numbers. The result will be a noticeably slower performance of such a system. But not so for the F2833x! This family of devices have an additional Floating-Point hardware unit, which can directly operate on Floating-Point numbers!

A second option for the Fixed-Point part of the F2833x is called “IQ-Math”. Texas Instruments provides a library that uses the internal hardware of the C28x in the most efficient way to operate with 32bit Fixed-Point numbers. Taking into account that most process data usually do not exceed a 16-bit resolution, the library gives enough headroom for advanced numerical calculations. The latest version of Texas Instruments “IQ-Math” Library can be found with literature number “SPRC087” at www.ti.com. We will discuss this library in more detail in Chapter 17.

Module Topics

Numbering Systems **4-1**

Introduction *4-1*

Module Topics *4-2*

Floating-Point, Integer and Fixed-Point *4-3*
Processor Types 4-4

IEEE 754 Floating-Point Format *4-5*

Integer Number Basics *4-8*
Two's Complement representation 4-8
Binary Multiplication 4-8

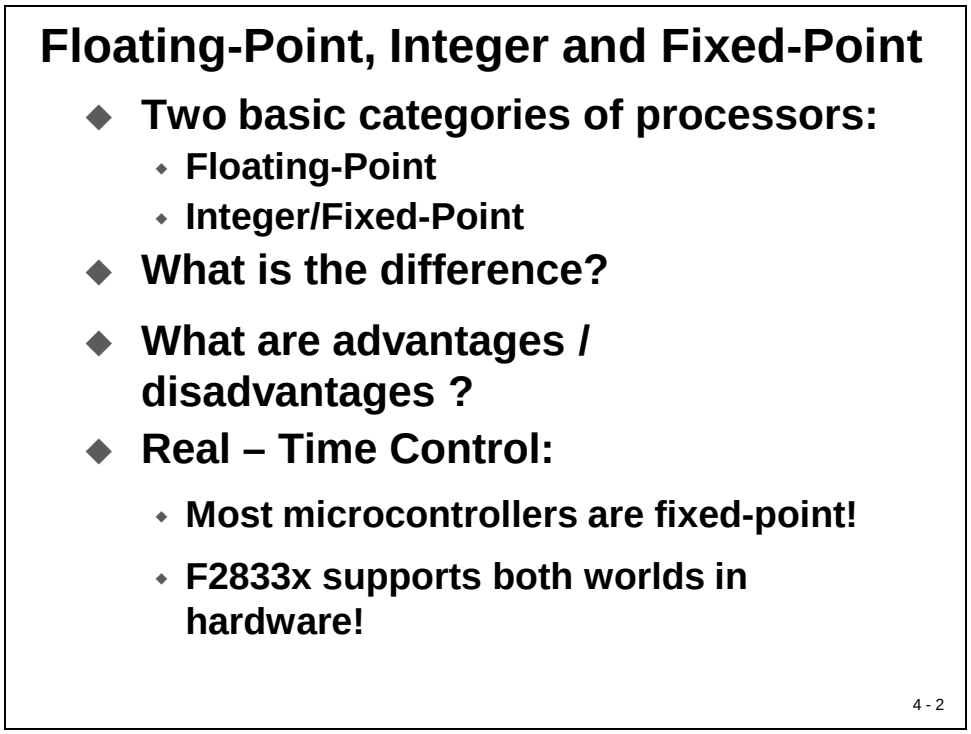
Binary Fractions *4-10*
Multiplying Binary Fractions 4-10

The "IQ"-Format *4-12*
Fractional Data in C 4-15

Lab4: Fixed-Point and Floating-Point *4-16*
Objective 4-16
Procedure 4-16
Open Files, Create Project File 4-16
Build and Load 4-17
Test the Fixed-Point solution 4-17
Floating-Point Library 4-20
Floating-Point Hardware 4-22
Summary: 4-24

Floating-Point, Integer and Fixed-Point

All processors can be divided into two groups, “Floating-Point” and “Fixed-Point”. However, recent processor designs, such as the F2833x cover both numerical schemes. The core of a Floating-Point processor is a hardware unit that supports Floating-Point operations according to the international standard IEEE-754. Intel’s x86-family of Pentium processors is probably the most popular example of this type. Floating-Point processors are very efficient when operating with Floating-Point data and allow a high dynamic range for numerical calculations. They are not so efficient when it comes to control tasks (bit manipulations, input/output control, and interrupt response) and they are usually more expensive than their Fixed-Point counter parts.



Fixed-Point Processors are based on internal hardware that supports operations with integer data. The Arithmetic Logic Unit (ALU) and in case of a Digital Signal Controller (DSC), the hardware multiply unit expects data to be in one of the Fixed-Point format data types. There are limitations in the dynamic range of a Fixed-Point processor, but they are inexpensive.

But what happens, when we write a program for a Fixed-Point processor in C and we declare a Floating-Point data type ‘float’ or ‘double’? The answer is that library functions are provided to support this data type on a Fixed-Point machine. However, these standard ANSI-C functions consume a lot of computing power. If we take into account the time constrains in a real time project, we just cannot afford to use these data types in most embedded control applications.

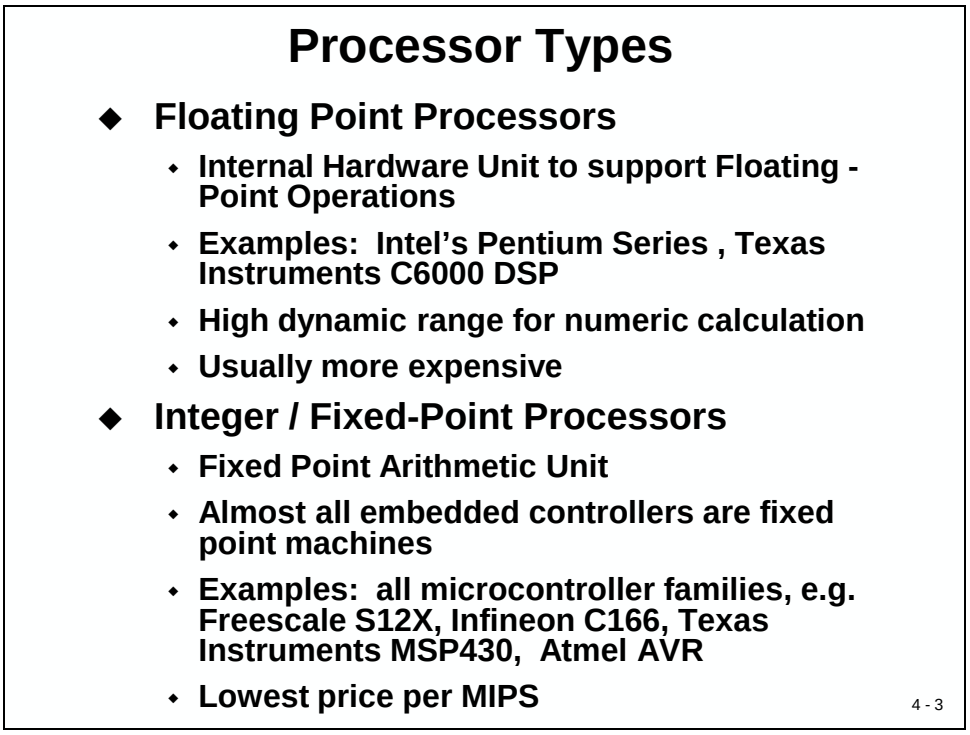
But there is good news: the F2833x offer two solutions to reduce the computing time on Floating-Point numbers: (1) an optimized library called “IQ-Math” and (2) an additional Floating-Point hardware unit. The IQ-Math Library is a set of highly optimized and high precision mathematical functions used to seamlessly port Floating-Point algorithms into Fixed-Point code. In addition, by incorporating the ready to use high precision functions, the

IQ-Math library can significantly shorten an embedded control development time. We will discuss this in more detail in Chapter 17.

Processor Types

Most of today's microprocessors fall into the category of Fixed-Point types. There is a wide range of semiconductor manufacturers that offer devices of this type. Just to name a few (the list is in random order and not exhaustive):

- Atmel AVR, ARM7 and Cortex M3 based devices
- Freescale HCS12X, MC56F83x, MCF523x
- Renesas SH4
- Texas Instruments MSP430, TMS320F280xx, Stellaris M3
- Infineon XE166, XC878
- ST Microelectronics STM32
- NEC V850ES / IE2
- Fujitsu MB91480
- Microchip dsPIC 33FJxx
- NXP LPC2900
- Toshiba TMP370



The world of Floating-Point processors is not as widespread as the Fixed-Point group. The most famous member is Intel's Pentium family, but there are also others (again, the list is in random order and not exhaustive):

- Intel x86 Pentium
- Freescale MPC556, PowerPC
- Texas Instruments C6000, DaVinci , TMS320F2833x

IEEE 754 Floating-Point Format

The **IEEE Standard for Floating-Point Arithmetic** (**IEEE-754**) is the most widely-used standard for Floating-Point computation, and is followed by many hardware and software implementations. Many computer languages allow or require that some or all arithmetic be carried out using IEEE-754 formats and operations. The current version is IEEE-754-2008, which was published in August 2008; it includes nearly all of the original IEEE-754-1985 (which was published in 1985) and the IEEE Standard for Radix-Independent Floating-Point Arithmetic (IEEE-854-1987).

The standard defines:

- *arithmetic formats:* sets of binary and decimal Floating-Point data, which consist of finite numbers, (including signed zeros and subnormal numbers), infinities, and special 'not a number' values (NaNs)
- *interchange formats:* encodings (bit strings) that may be used to exchange Floating-Point data in an efficient and compact form
- *rounding algorithms:* methods to be used for rounding numbers during arithmetic and conversions
- *operations:* arithmetic and other operations on arithmetic formats
- *exception handling:* indications of exceptional conditions (such as division by zero, overflow, *etc.*)

The standard also includes extensive recommendations for advanced exception handling, additional operations (such as trigonometric functions), expression evaluation, and for achieving reproducible results.

Standard IEEE-754 Single Precision Floating-Point

Case 1:	if e = 255 and f ≠ 0,	then v = NaN
Case 2:	if e = 255 and f = 0,	then v = $[(-1)^s]$*infinity
Case 3:	if 0 < e < 255,	then v = $[(-1)^s]*[2^{(e-127)}]*(1.f)$
Case 4:	if e = 0 and f ≠ 0,	then v = $[(-1)^s]*[2^{(-126)}]*(0.f)$
Case 5:	if e = 0 and f = 0,	then v = $[(-1)^s]$*0

Advantage ⇒ Exponent gives large dynamic range
Disadvantage ⇒ Precision of a number depends on its exponent

4 - 4

In the following slides we will focus on the arithmetic numbering formats only.

32-bit Floating-Point format (C data type "float):

- **Sign Bit (S):**
  - Negative: bit 31 = 1 / Positive: Bit 31 = 0
- **Mantissa (M):**

$$M = 1 + m_1 \cdot 2^{-1} + m_2 \cdot 2^{-2} + ... + m_{23} \cdot 2^{-23} = 1 + \sum_{i=1}^{23} m_i \cdot 2^{-i}$$

  - Mantissa is normalized to $m_0 = 1$; m0 will not be stored in memory!

$$1 \le M < 2$$

- **Exponent (E):**
  - 8 Bit signed exponent, stored with offset, OFFSET = +127
- **Summary**:
  - $Z = (-1)^S \cdot M \cdot 2^{E-OFFSET}$

Example 1:

0x 3FE0 0000 = 0011 1111 1110 0000 0000 0000 0000 0000 B
S = 0
E = 0111 1111 = 127
M = (1).11000 = 1 + 0.5 + 0.25 = 1.75
$Z = (-1)^0 * 1.75 * 2^{127-127} = 1.75$

Example 2:

0x BFB0 0000 = 1011 1111 1011 0000 0000 0000 0000 0000 B
S = 1
E = 0111 1111 = 127
M = (1).011 = 1 + 0.25 + 0.125 = 1.375
$Z = (-1)^1 * 1.375 * 2^{127-127} = -1.375$

Example 3:

Z = - 2.5
S = 1
$2.5 = 1.25 * 2^1$
1 = E – OFFSET
E = 128
M = 1.25 = (1).01 = 1 + 0.25
Binary Result: 1100 0000 0010 0000 0000 0000 0000 0000 B = 0x C020 0000

The advantage of Floating-Point is its huge dynamic range, which is given by the most positive exponent (+127, base 2). This exponent plus the maximum mantissa leads to a range of:

$$Z = \pm(1 - 2^{24}) * 2^{128} \approx \pm 3.403 * 10^{38}$$

The resolution of a single precision Floating-Point number is given by the smallest number that can be represented in this format:

$$Z = 2^{-23} * 2^{-126} = 2^{-149} \approx 1.401 * 10^{-45}$$

It seems that with this dynamic range and resolution we should be able to solve any mathematical operation. However, when it comes to a simple add operation of a large number and a very small number, even a Floating-Point device can fail! Look at the following example for z = x + y:

Floating-Point does not solve everything!

Example: x = 10.0 (0x41200000)
\+ y = 0.000000240 (0x3480D959)

z = 10.000000240 WRONG!

You cannot represent 10.000000240 with single-precision floating-point

0x412000000 = 10.000000000
10.000000240 ⇐ can't represent!
0x412000001 = 10.000001000

So z gets rounded down to 10.000000000

4 - 5

Such a rounding error can happen, when we have to add a compensation value (small) to a larger set point value in a closed control loop! The result would be a somewhat sluggish behavior of our digital controller.

In the second part of this chapter you will learn that Fixed-Point numbers do not show this behavior, if we limit the dynamic range of the numbers to the expected area of a closed loop control system. When we use the Texas Instruments IQ-Math Fixed-Point hardware, it will add 10.0 and 0.00000024 to give the exact result of 10.00000024! This is a considerable advantage of Fixed-Point numbers over Floating-Point numbers!

Integer Number Basics

Two's Complement representation

The next slides summarize the basics of the two's complement representation of signed integer numbers. You should already be familiar with these schemes from basic lessons on computer engineering or digital systems. If not, use Wikipedia to update yourself!

Integer Numbering System Basics

- **Binary Numbers**

$0110_2 = (0*8)+(1*4)+(1*2)+(0*1) = 6_{10}$

$11110_2 = (1*16)+(1*8)+(1*4)+(1*2)+(0*1) = 30_{10}$

- **Two's Complement Numbers**

$0110_2 = (0*-8)+(1*4)+(1*2)+(0*1) = 6_{10}$

$11110_2 = (1*-16)+(1*8)+(1*4)+(1*2)+(0*1) = -2_{10}$

4 - 6

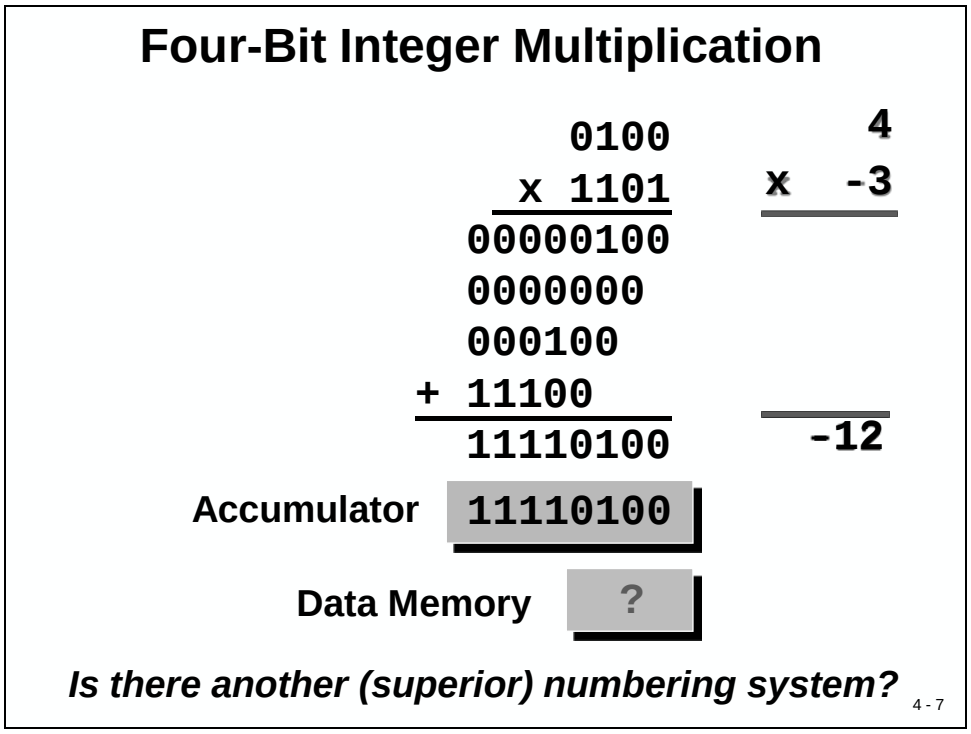
In the signed integer format, the most significant bit (MSB) carries a negative weight of -1. If the MSB is set, we have to multiply its coefficient representation by -1 (compare example in the 2nd half of Slide 4-6).

Binary Multiplication

Now consider the process of multiplying two two's complement values, which is one of the most often used operations in digital control. As with "long hand" decimal multiplication, we can perform binary multiplication one "place" at a time, and sum the results together at the end to obtain the total product.

Note: The method shown at the following slide is not the method the F22833x uses to multiply integer numbers - it is merely a way of observing how binary numbers behave in arithmetic processes.

The F2833x uses 32-bit operands and an internal 64-bit product register. For the sake of clarity, consider the example below where we shall investigate the use of 4-bit values and an 8-bit accumulation:



In this example, consider the following:

- 4 multiplied by (-3) gives (-12) in decimal.
- The size of the product is twice as long as the input values (4 bits * 4 bits = 8 bits).
- If this product is to be used in a next loop of a calculation, how can the result be stored back to memory in the same length as the inputs?
    - Store back upper 4 Bit of Accumulator? ➔ -1
    - Store back lower 4 Bit of Accumulator? ➔ +4
    - Store back all 8 Bit of Accumulator? ➔ overflow of length
- As a result, scaling of intermediate results is needed!

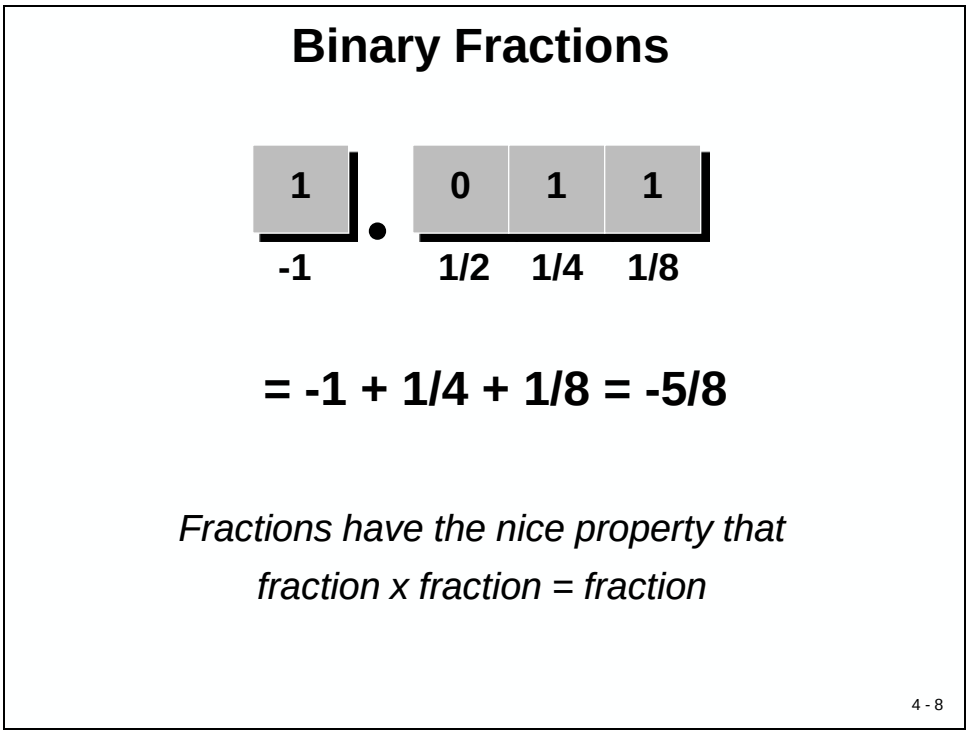
From this analysis, it is clear that integers do not behave well when multiplied.

The question is: might some other type of integer number system behave better? Is there a number system where the results of a multiplication have bounds?

The answer is: yes, there is.

Binary Fractions

In order to represent both positive and negative values, the two's complement process will again be used. However, in the case of fractions, we will not set the LSB to 1 (as was the case for integers). When we consider that the range of fractions is from -1 to ~+1, and that the only bit which conveys negative information is the MSB, it seems that the MSB must be the “negative ones position”. Since the binary representation is based on powers of two, it follows that the next bit would be the “one-half” position, and that each following bit would have half the magnitude again.



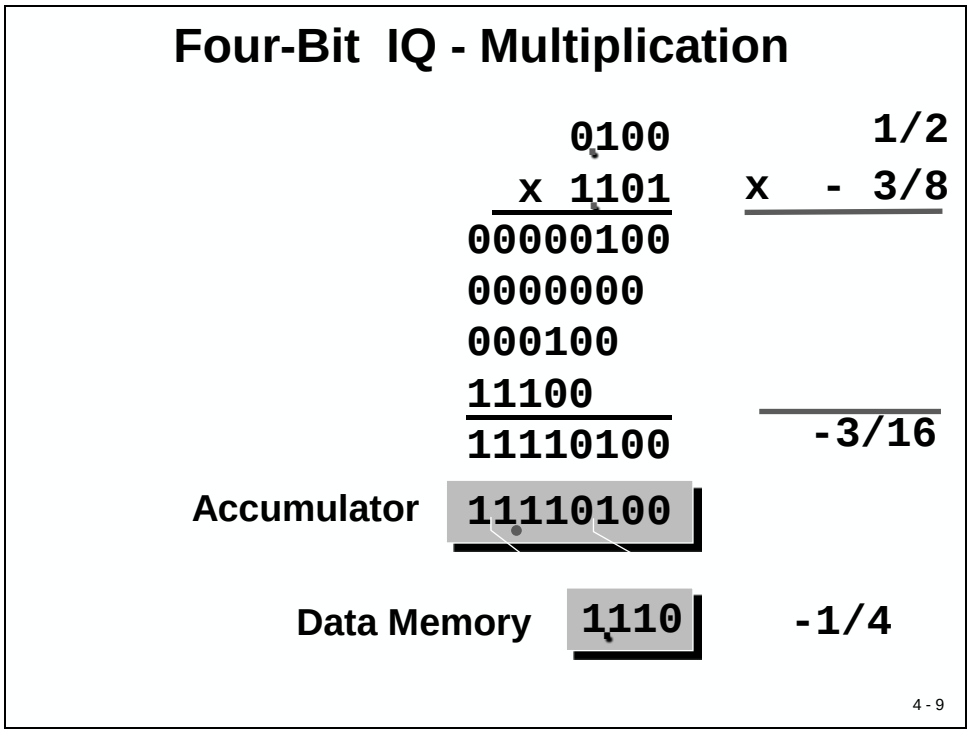
Multiplying Binary Fractions

When the F2833x performs an integer multiplication, the process is identical for all operands, integers or fractions. Therefore, the user must determine how to interpret the results. As before, consider the 4-bit multiply example:

The input numbers are now split into two parts - integer part (I-“integer”) and fractional part (Q-“quotient”). These type of Fixed-Point numbers are often called “IQ”-numbers, or for simplicity sometimes just Q-numbers.

The example below shows 2 input numbers in I1Q3-Format. When multiplied, the length of the result will add both I and Q portions (see also next slide):

I1Q3 * I1Q3 = I2Q6



If we store back the intermediate product with the four bits around the binary point we keep the data format (I1Q3) in the same shape as the input values. There is no need to re-scale any intermediate results!

Advantage: With Binary Fractions we will gain a lot of speed in closed loop calculations.

Disadvantage: The result might not be the exact one. As you can see from the slide above we will end up with (-4/16) stored back to data memory. Bits 2^{-4} to 2^{-6} are truncated. The correct result would have been (-3/16).

Recall that the 4-bit input operand multiplication operation is not the real size for the F2833x, which operates on 32-bit input values. In this case, the truncation will affect bits 2^{-32} to 2^{-64}. Given the real size of process data with, let us say 12-bit ADC measurement values, there is plenty of room left for truncation.

In most cases we will truncate noise only. However, in some feedback applications like Infinite Impulse Response (IIR)-Filters the small errors can add and lead to a given degree of instability. It is designer's responsibility to recognize this potential source of failure when using binary fractions.

The “IQ”-Format

So far we have discussed only the option of using fractional numbers with the binary point at the MSB-side of the number. In general, we can place this point anywhere in the binary representation. This gives us the opportunity to trade off dynamic range against resolution.

Fractional Representation

$$-2^{I} + 2^{I-1} + \ldots + 2^{1} + 2^{0} \bullet 2^{-1} + 2^{-2} + \ldots + 2^{-Q}$$

“IQ” – Format

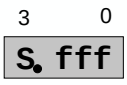
“I” ⇒ INTEGER – Fraction
“Q” ⇒ QUOTIENT – Fraction

Advantage ⇒ Precision same for all numbers in an IQ format
Disadvantage ⇒ Limited dynamic range compared to floating-point

4 - 10

IQ - Examples

I1Q3 – Format:



Most negative decimal number: -1.0 = 1.000 B

Most positive decimal number: +0.875 = 0.111 B

Smallest negative decimal number: $-1*2^{-3}$ (-0.125) = 1.111 B

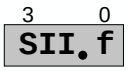
Smallest positive decimal number: 2^{-3} (+0.125) = 0.001 B

Range: -1.0 …. 0.875 (≈ + 1.0)
Resolution: 2^{-3}

4 - 11

IQ - Examples

I3Q1 – Format:



Most negative decimal number: -4.0 = 100.0 B

Most positive decimal number: + 3.5 = 011.1 B

Smallest negative decimal number: $-1 * 2^{-1}$ (-0.5) = 111.1 B

Smallest positive decimal number: 2^{-1} (+0.5) = 000.1 B

Range:	**-4.0 …. +3.5 (≈ + 4.0)**
Resolution:	**2^{-1}**

4 - 12

IQ - Examples

I1Q31 – Format:

Most negative decimal number: -1.0
1.000 0000 0000 0000 0000 0000 0000 0000 B

Most positive decimal number: ≈ + 1.0
0.111 1111 1111 1111 1111 1111 1111 1111 B

Smallest negative decimal number: $-1*2^{-31}$
1.111 1111 1111 1111 1111 1111 1111 1111 B

Smallest positive decimal number: 2^{-31}
0.000 0000 0000 0000 0000 0000 0000 0001 B

Range:	**-1.0 …. (+1.0)**
Resolution:	**2^{-31}**

4 - 13

IQ - Examples

I8Q24 – Format:

31 0

S III IIII. ffff ffff ffff ffff ffff

Most negative decimal number: -128
1000 0000. 0000 0000 0000 0000 0000 0000 B

Most positive decimal number: ≈ + 128
0111 1111. 1111 1111 1111 1111 1111 1111 B

Smallest negative decimal number: $-1*2^{-24}$
1111 1111. 1111 1111 1111 1111 1111 1111 B

Smallest positive decimal number: 2^{-24}
0000 0000. 0000 0000 0000 0000 0000 0001 B

Range:	**-128 …. (+128)**
Resolution:	**2^{-24}**

4 - 14

Now let us resume the failing Floating-Point example from the beginning of this module; IQ-Math can do much better:

IQ-Math can do better!

I8Q24 Example:		x = 10.0	(0x0A000000)
	+	y = 0.000000240	(0x00000004)
		z = 10.000000240	(0x0A000004)

Exact Result (this example)

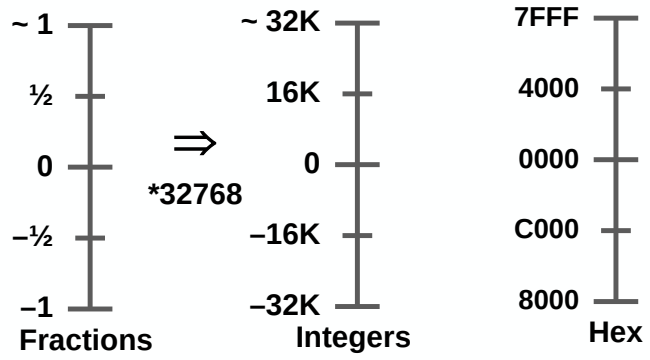
4 - 15

Fractional Data in C

If by now you are convinced that fractional data has advantages over other number representations, the next question is, how do we code fractions in an ANSI-C environment? The ANSI-C standard does not define a dedicated data type, such as “fractional”. There is a new ANSI-standard under development, called “embedded C”, which will eventually use this type.

For now we can use the following trick, as shown in Slide 4-16:

How is a fraction coded?



- **Example: represent the fraction number 0.707**

```
void main(void)
{
   int coef = 32768 * 707 / 1000;
}
```

4 - 16

Fractional vs. Integer

- **Range**
  - **Integers have a maximum range determined by the number of bits**
  - **Fractions have a maximum range of ±1**
- **Precision**
  - **Integers have a maximum precision of 1**
  - **Fractional precision is determined by the number of bits**

4 - 17

Lab4: Fixed-Point and Floating-Point

Objective

The objective of this lab is to practice and benchmark the different options for the F2833x in terms of numerical systems. We have already discussed that the F2833x supports both Fixed-Point and Floating-Point numbers in hardware. In the following lab we will use the simple code example from Chapter 3 and compile it for the different numbering systems. To benchmark the results, we will use a time measurement tool, called "Profiler", which is part of Code Composer Studio. The following procedure will summarize all steps discussed in this chapter.

Lab4: Fixed-Point and Floating-Point

- **Benchmark Multiply Operation**
- **k = i * i**
- **Test setup:**

1. **Integer multiply operation**
2. **Floating-Point multiply by Floating-Point library**
3. **Floating-Point multiply by Floating-Point hardware unit**

Benchmark result:

	Fixed-point	Floating-Point-Library	Floating-Point-Hardware
code size (words)	5	89	8
clock cycles (6.67 ns)	4	112	4

4 - 18

Procedure

Open Files, Create Project File

1. Using Code Composer Studio, create a new project, called **Lab4.pjt** in C:\DSP2833x\Labs (or another working directory used during your class, ask your instructor for specific location!)
2. Open the file "lab3.c" from project "lab3" and save it as "lab4_1.c" in the working directory of project "lab4":

   ➔ File ➔ Open… ➔ "lab3.c"
   ➔ File ➔ Save As… ➔ "lab4_1.c"

3. Add the following files to your project:

   ➔ Project ➔ Add Files to project
   - Lab4_1.c
   - C:\CCStudio_v3.3\C2000\cgtools\lib\rts2800_ml.lib
   - C:\tidcs\c28\DSP2833x\v131\DSP2833x_common\cmd\28335_RAM_lnk.cmd

4. Verify that in Project ➔ Build Options ➔ Linker the "Autoinit" Model field is set to: "Run-time-Autoinitialisation [-c]"

5. Set the stack size to 0x400: Project ➔ Build Options ➔ Linker ➔ Stack Size

6. Close the Build Options Menu by clicking OK

Build and Load

7. Click the "Rebuild All" button or perform: Project ➔ Build and watch the tools run in the build window. Debug as necessary.

8. Load the output file to the device. Click: File ➔ Load Program and choose the output file "Lab4.out", which you just generated. Note: the file is stored in the sub-directory "Debug" in the project folder.

   Note: Code Composer can automatically load the output file after a successful build. To do this by default, click on the menu bar: Option ➔ Customize ➔ Program/Project/CIO ➔ Program Load and select: "Load Program after Build", then click OK.

Test the Fixed-Point solution

9. Reset the DSP by clicking on ➔ Debug ➔ Reset CPU, followed by ➔ Debug ➔ Restart

10. Disable the watchdog (Note: the watchdog will be discussed in the next chapter):
    ➔ GEL ➔ Watchdog ➔ Disable_WD

11. Run the program until the first line of your C code by clicking: Debug ➔ Go main. Verify that in the working region of the source code "Lab4_1.c" is highlighted and that the yellow arrow for the current Program Counter is positioned at the first line of function "main()".

12. Remember, that both variables are defined as data type "unsigned int" (16-bit integer numbers).

13. Now, benchmark the results. First switch into "Mixed-Mode" (Right click into "lab4_1.c" and select "Mixed-Mode"). Inspect the code-line, which we used to multiply i*i:

```
{
    k = i*i;
     00907D       C$DW$L$_main$3$B:
     00907D 2D41          MOV         T,*-SP[1]
     00907E 761F0300      MOVW        DP,#0x0300
     009080 1241          MPY         ACC,T,*-SP[1]
     009081 9608          MOV         @8,AL
}
```

The C line "k = i*i" has been translated into a set of four assembly language instructions.

- The first line moves a 16-bit value from stack memory (to be exact: stack-pointer address minus 1; that is our local variable 'i') to internal register 'T'.
- Next, register "Data Page (DP)" is loaded with value 0x300. This instruction prepares the correct access in direct addressing mode for the address of variable 'k' in line 4.
- Line 3 multiplies the value in register T by the value from the same stack memory location, which was used in line 1 (variable 'i'). The 32-bit product is stored in register "Accumulator (ACC)".
- Line 4 stores the lower 16-bits of the 32-bit product (register "Accumulator-low", AL) back into memory at address 8 of the active page. Obviously, address 8 on page 0x300 is the location of global variable 'k'.

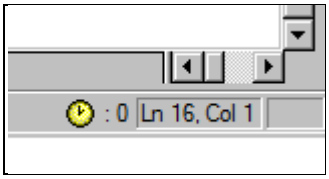
Benchmark #1 (code-size): As you can see from the numbers at the left hand side, our code snippet "k = i * i" occupies the code memory addresses 0x907D to 0x9081, which gives a code size of 5 words. (Note: the absolute address numbers might be different on your CCS-session; however the size should be identical).

Turn off the mixed mode (right mouse click and select "Source Mode").

Benchmark #2 (execution speed): To measure the number of execution clock cycles, we can use the CCS "Profiler":

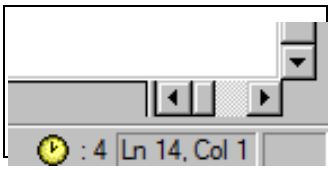
➔ Profile ➔ Clock ➔ Enable
➔ Profile ➔ Clock ➔ View

A small yellow profiler clock will appear in the lower right corner of CCS.



This is our time measurement system. Using single step (F11), run the code until you reach the line "k = i * i". The number to the right of the clock gives the number of elapsed clock cycles. To clear this number, double click on the yellow profiler clock.

Now, with the yellow arrow still on line "k = i * i", perform a single step (F11). The profiler clock should show a 4, which indicates that one execution of the line "k = i * i" took 4 clock cycles. This result corresponds to the four machine code instructions, which we inspected above. Each instruction is executed in 1 CPU clock cycle.



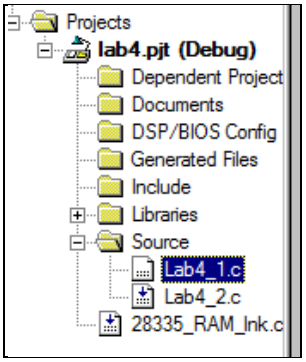
Result 1 (Fixed-Point math):

- Code size: 5 words
- Clock cycles: 4

Floating-Point Library

14. Now let us change the code from Fixed-Point to Floating-Point. Change the data type for 'k' from **"unsigned int"** to **"float"**.
15. In function "main()" add a new local variable **"float f = 1.0;"**
16. Change the code line "k = i * i;" into **"k = f * f;"**.
17. After this line but still inside the for-loop, add a new line **"f = f + 1.0;"**
18. Save the file as "Lab4_2.c" and add this file to project "Lab4".

    Exclude "Lab4_1.c" from the build. In the project window, right click on "Lab4_1.c", select "File Specific Options", "General" and enable "Exclude file from Build". This technique allows us to keep more than one source code file in the project tree and we can change between the different files. Note the missing arrow in the icon for "Lab4_1.c", which indicates that this file has been excluded:



19. Rebuild the project and reload the new code:

    ➔ Project ➔ Rebuild All
    ➔ File ➔ Load Program ➔ Lab4.out
    ➔ Debug ➔ Reset CPU
    ➔ GEL ➔ Watchdog ➔ Disable_WD
    ➔ Debug ➔ Restart
    ➔ Debug ➔ Go main

20. Now, benchmark the results. Again, switch into "Mixed-Mode" (Right click into "lab4_2.c" and select "Mixed-Mode"). Inspect the code-line, which we used to multiply f * f:

```
k = f*f;
00911E       C$DW$L$_main$3$B:
00911E 0646          MOVL        ACC,*-SP[6]
00911F 1E42          MOVL        *-SP[2],ACC
009120 0646          MOVL        ACC,*-SP[6]
009121 76409078      LCR         FS$$MPY
009123 761F0300      MOVW        DP,#0x0300
009125 1E08          MOVL        @8,ACC
```

The line “k = f * f” has been translated in a series of six assembly language instructions.

- The 1st line reads the value from variable ‘f’ (stack memory location [SP-6]) into register “ACC”
- The 2nd line stores the value from ACC in stack memory location [SP-2].
- The 3rd line reads again the value from variable ‘f’ into register “ACC”-
- Line 4 calls a Floating-Point multiply function “FS$$MPY”. The assembly instruction “LCR-**L**ong **C**all with **R**eturn” calls a function from library “rts2800_ml.lib”, which performs a Floating-Point multiply on a Fixed-Point device.
- The last two lines are used to store the result of the function call, which is returned in register ACC, into memory address 8 of the active data page (address of variable ‘k’).

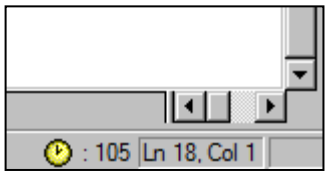
Benchmark #3 (code-size, Floating-Point library function):

As you can see from the numbers at the left hand side, our code-snippet “k = f * f” occupies the code memory addresses 0x911E to 0x9125, which gives a code size of 8 words. (Note: the absolute address numbers might be different on your CCS-session; however the size should be identical). However, this result is not the full story! For code size we have to add the site of function “FS$$MPY”. When you assembly single step (F11) until you reach the instruction “LCR” and continue with another “F11”, CCS will open another disassembly window with the instructions of function “FS$$MPY”. If you scroll down this window, you will find an instruction “LRETR”, which is the return instruction of this function. The difference between start- and end- address (0x90C9 - 0x9078) is the size of function “FS$$MPY”.

Result: 8 + 81 = 89 words code size.

Benchmark #4 (execution speed, Floating-Point library):

Using the same profiler steps as for the integer code, measure the number of clock cycles for one Floating-Point multiplication. From the beginning of “main()”, single step until you reach the line “k = f * f”. Clear the clock counter and perform another source single step (F10). Note, that for the first loop, where ‘f’ is zero, you will obtain a relatively small number (25). If you repeat the test for other values of ‘i’, you will obtain a much greater number (105) of clock cycles. Obviously, the library function “FS$$MPY” shortens the calculation, if one factor is zero. Note: The numbers were measured with C compiler - and library version 5.2.2. They might be different on your installation, but they should be in the same sort of range.



Results (Floating-Point library):

- Code size: 89 words
- Clock cycles: 105

Floating-Point Hardware

As a final step we will use the F2833x Floating-Point hardware unit and replace the Floating-Point library. This should reduce both the code size and the number of clock cycles back to the integer results.

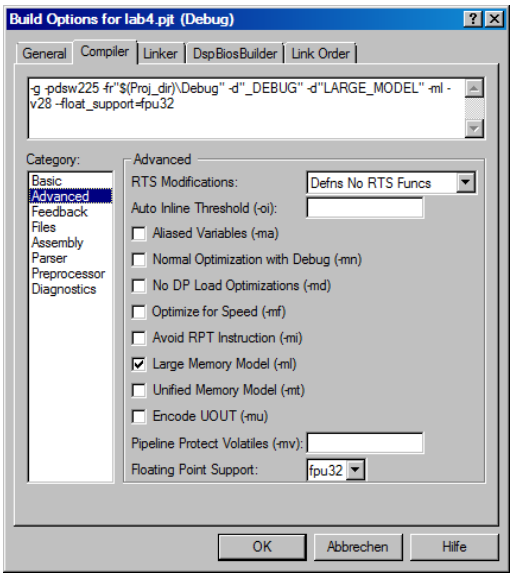
21. Add the Floating-Point support function to your project:

    ➔ Project ➔ Add Files to Project ➔ C:\CCStudio_v3.3\C2000\cgtools\lib\rts2800_fpu32.lib

    Set the “file specific options” of “rts2800_ml.lib” to “Exclude file from Build”

22. Set the project build options to support Floating-Point:

    ➔ Project ➔ Build Options ➔ Compiler ➔ Advanced ➔ Floating Point Support: fpu32



23. Rebuild the project and reload the new code:

    ➔ Project ➔ Rebuild All
    ➔ File ➔ Load Program ➔ Lab4.out
    ➔ Debug ➔ Reset CPU
    ➔ GEL ➔ Watchdog ➔ Disable_WD
    ➔ Debug ➔ Restart
    ➔ Debug ➔ Go main

24. Now, benchmark the results. Again, switch into “Mixed-Mode” (Right click in “lab4_2.c” and select “Mixed-Mode"). Inspect the code lines, which we used to multiply f * f:

```
k = f*f;
00904E      C$DW$L$_main$2$E:
00904E E2AF0044    MOV32      R0H, *-SP[4], UNCF
009050 E2AF0144    MOV32      R1H, *-SP[4], UNCF
009052 E7000008    MPYF32     R0H, R1H, R0H
009054 761F0300    MOVW       DP,#0x0300
009056 E2030008    MOV32      @8, R0H
```

The Floating-Point line "k = f * f" has been translated in a series of 5 assembly instructions, which use the Floating-Point hardware unit:

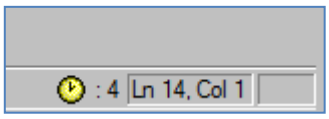
- The 1st line moves a value from stack memory location [SP-4] into Floating-Point register R0H. This is variable 'f' as a 1st multiply factor.
- The 2nd line loads again the value from variable 'f', but now into register R1H. This is the 2nd multiply factor.
- The next line is a Floating-Point multiply operation of R0H * R1H. The product is stored in register R0H.
- The last two lines store the product (R0H) back in data memory at address 8 of the active page 0x300.

Benchmark #5 (code-size, Floating-Point hardware):

As you can see from the numbers at the left hand side, our code-snippet "k = f * f" occupies the code memory addresses 0x904E to 0x9057, or 10 words.

Benchmark #6 (execution speed, Floating-Point hardware):

Using the profiler, measure the number of clock cycles for one Floating-Point multiplication. From the beginning of "main()", single step until you reach the line "k = f * f". Clear the clock counter and do another source single step(F10). The result is 4!



Summary:

In Lab4 we benchmarked the 3 possible solutions that can be used to multiply two values. For a Fixed-Point processor the native numbering scheme is integer. As you can see from the numbers, both code size and clock cycles are minimal; we can generate an optimal solution for real-time control, where speed always has the highest priority.

	Fixed-Point	Floating-Point-Library	Floating-Point-Hardware
code size (words)	5	89	10
clock cycles (6.67 ns)	4	105	4

However, if the software designer decides to use Floating-Point data types for variables k and f, the library function will dramatically increase both code size and number of clock cycles. Such a solution could lead to code, which could well be too slow for use in real-time control. For most microcontrollers this is the end of the road.

Not so for the F2833x! If we enable Floating-Point hardware support, we easily can use Floating-Point data types with the same speed factor as in Fixed-Point! The code size is a little bit larger than for Fixed-Point numbers, but in most cases this does not matter.