C28x IQ – Math Library

Introduction

One of the most important estimations in embedded control is the calculation of computing time for a given task. Since embedded control has to cope with these tasks in a given and fixed amount of time, we call this ‘Real Time Computing’. And, as you know, time goes very quickly.

Therefore, one of the characteristics of a processor is the 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 control units has increased dramatically. Just take the number of pages for the requirement specification of an electronic control unit 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 process data?

You know that the ‘native’ numbering scheme for a digital controller uses 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.

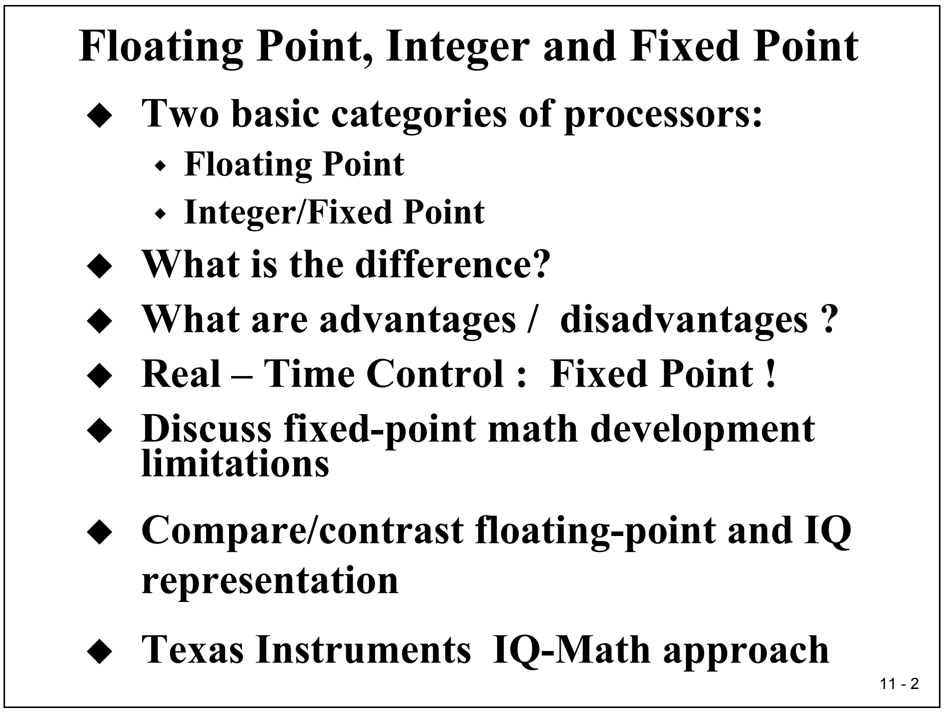
Because the C28x belongs to the fixed-point type we will focus on this type more in detail. After a brief discussion about binary numbers we will have a look into the different options to use the fixed-point unit of the C28x. It can perform various types of mathematical operations in a very efficient way, using only a few machine clock cycles.

The secret behind this approach is called “IQ-Math”. In case of the C28x 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 all process data usually do not exceed a resolution of 16 bits, 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.

Module Topics

Floating-point, Integer and Fixed-point

All processors can be divided into two groups, “floating-point” and “fixed-point”. The core of a floating-point processor is a hardware unit that supports floating-point operations according to international standard IEEE 754. Intel’s x86 – family of Pentium processors is a typical 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, interrupt response) – and, they are expensive.



Floating Point, Integer and Fixed Point

- **Two basic categories of processors:**
  - **Floating Point**
  - **Integer/Fixed Point**
- **What is the difference?**
- **What are advantages / disadvantages ?**
- **Real – Time Control : Fixed Point !**
- **Discuss fixed-point math development limitations**
- **Compare/contrast floating-point and IQ representation**
- **Texas Instruments IQ-Math approach**

11 - 2

Fixed-point Processors are based on internal hardware that supports operations with integer data. The arithmetic logic unit and, in case of digital signal processors, the hardware multiply unit, expect data to be in one of the fixed-point types. There are limitations in the dynamic range of a fixed-point processor, but they are inexpensive.

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’? A number of library functions support this data type on a fixed-point machine. These standard ANSI-C functions consume a lot of computing power. Recalling the time constrains in a real time project, we just can’t afford to use these data types in most of embedded control applications.

The solution, in case of the C28x is “IQ-Math”. The IQ-Math Library is a collection of highly optimised 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 shorten significantly an embedded control development time.

Processor Types

- **Floating Point Processors**
  - **Internal Hardware Unit to support Floating Point Operations**
  - **Examples : Intel’s Pentium Series , Texas Instruments C 6000 DSP**
  - **High dynamic range for numeric calculation**
  - **Rather expensive**
- **Integer / Fixed – Point Processors**
  - **Fixed Point Arithmetic Unit**
  - **Almost all embedded controllers are fixed point machines**
  - **Examples: all microcontroller families, e.g. Motorola HC68x, Infineon C166, Texas Instruments TMS430, TMS320C5000, C2000**
  - **Lowest price per MIPS**

11 - 3

IEEE 754 Floating-point Format

IEEE Standard 754 Single Precision Floating-Point

31	30 … 23	22 … 0
s	eeeeeeee	fffffffffffffffffffffff
1 bit sign	8 bit exponent	23 bit mantissa (fraction)

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

11 - 4

Floating-point definitions:

- **Sign Bit (S):**
  - Negative: bit 31 = 1 / Positive: Bit 31 = 0
- **Mantissa (M):**
  - $M = 1 + m_1 \cdot 2^{-1} + m_2 \cdot 2^{-2} + ... = 1 + \sum_{i=1}^{23} m_i \cdot 2^{-i}$
  - Mantissa is tailored to $m_0 = 1$; m0 will not be stored in memory!

$$1 \leq M < 2$$

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

Example1:

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

$Z = (-1)^0 * 1{,}75 * 2^{127-127} = 1.75$

Example2:

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

$$\mathbf{Z = (-1)^{1} * 1{,}375 * 2^{127-127} = -1.375}$$

Example3:

Z = -2.5

S = 1

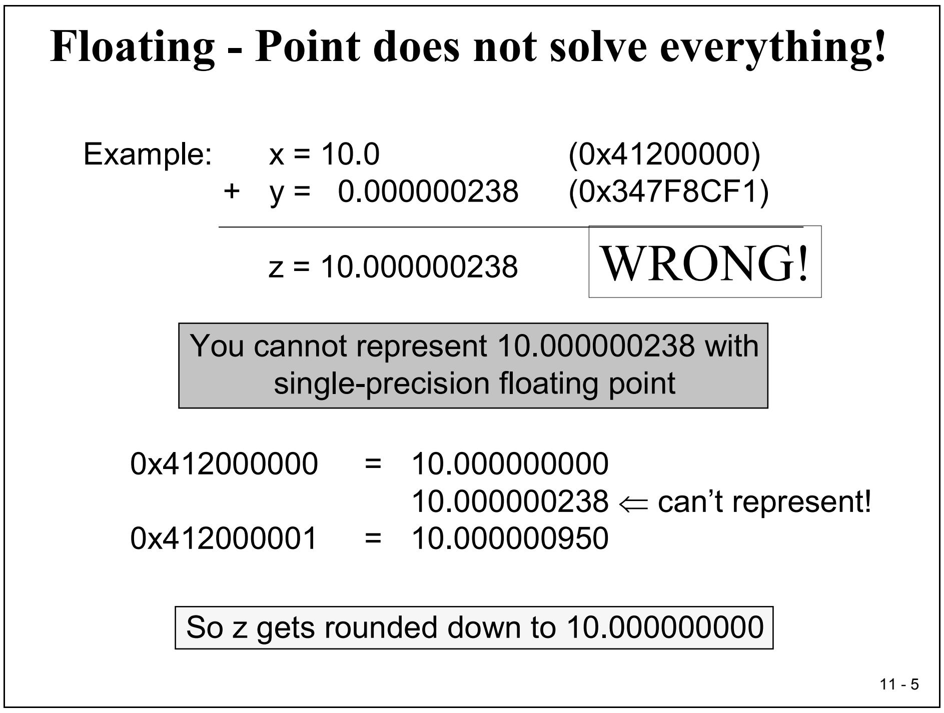
$2.5 = 1.25 * 2^{1}$

1 = E – OFFSET

E = 128

M = 1.25 = (1).01 = 1 + 0.25

Binary : 1100 0000 0010 0000 0000 0000 0000 0000 B = 0x C020 0000



Floating - Point does not solve everything!

```
Example:     x = 10.0                 (0x41200000)
           + y =   0.000000238        (0x347F8CF1)
           ---------------------------------------
             z = 10.000000238          WRONG!
```

You cannot represent 10.000000238 with single-precision floating point

```
0x412000000   =   10.000000000
                  10.000000238 ⇐ can't represent!
0x412000001   =   10.000000950
```

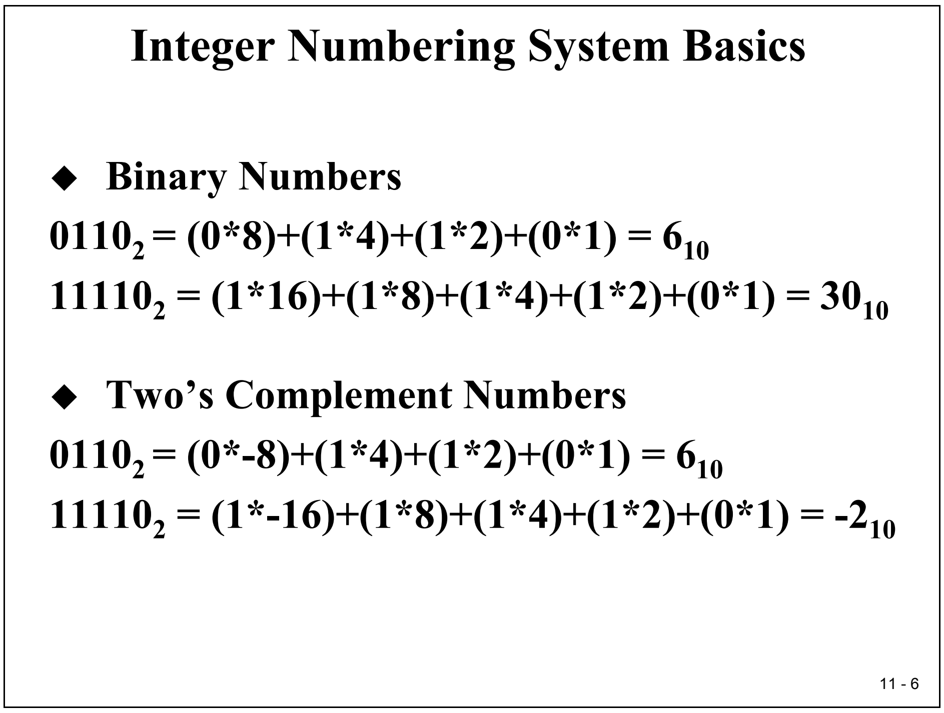
So z gets rounded down to 10.000000000

11 - 5

Integer Number Basics

Two's Complement representation

The next slides summarize the basics of the two's complement representation of signed integer numbers.



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**

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

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

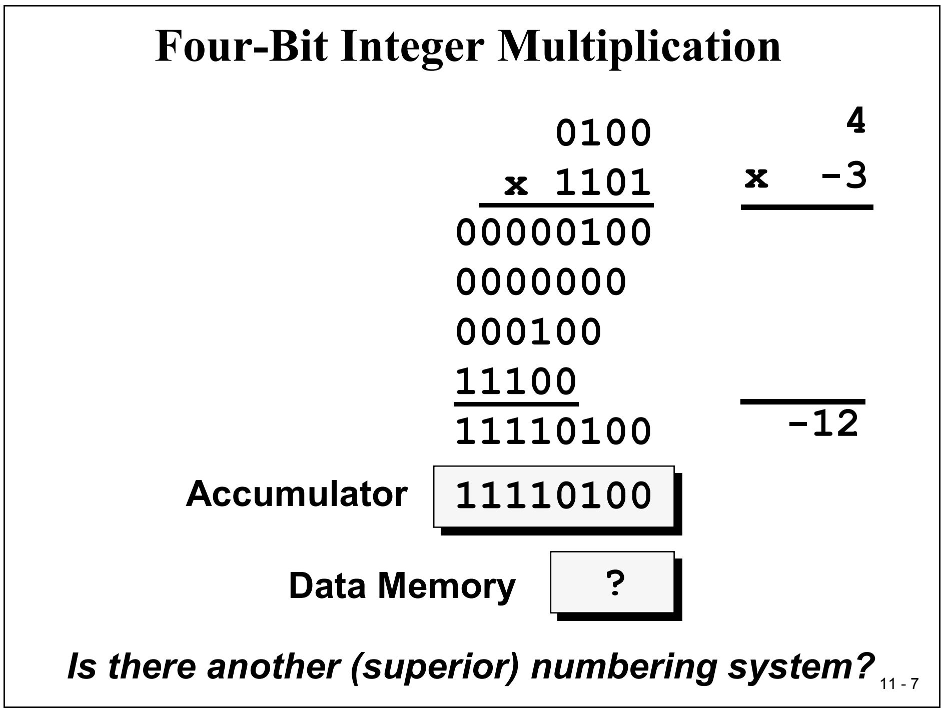
11 - 6

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 C28x uses to multiply numbers — it is merely a way of observing how binary numbers work in arithmetic processes.

The C28x 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:



Four-Bit Integer Multiplication

```
     0100          4
   x 1101      x  -3
 00000100
 0000000
 000100
 11100
 11110100        -12
```

Accumulator 11110100

Data Memory ?

Is there another (superior) numbering system? 11 - 7

In this example, consider the following:

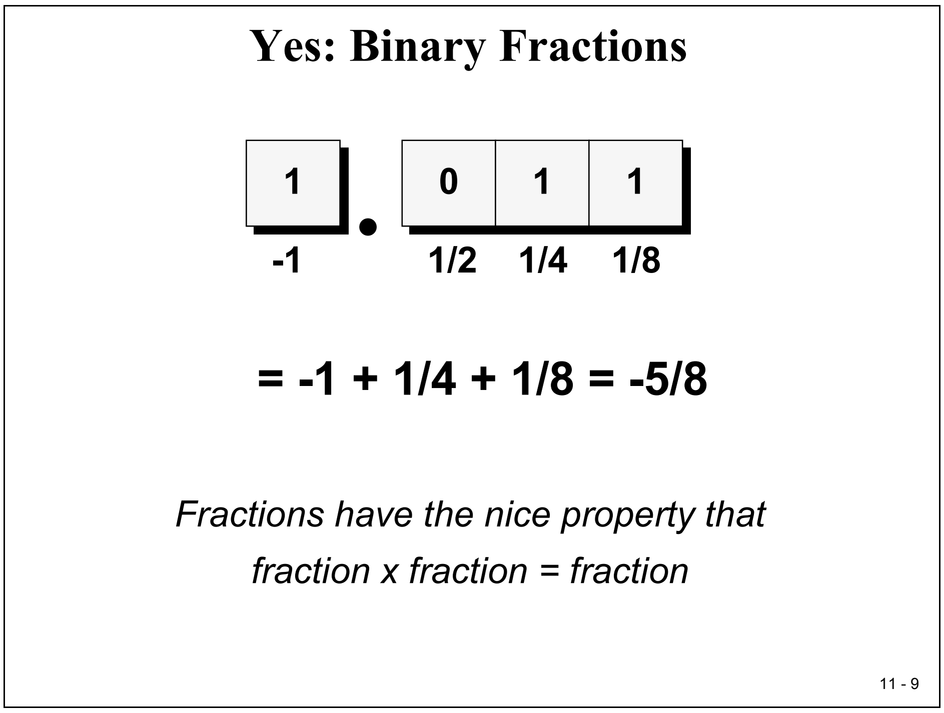
- 4 multiplied by (-3) gives (-12) in decimal
- Size of the product is twice as long as the input values (4 bit * 4 bit = 8 bit)
- If this product is to be used in a next loop of a calculation, how can the result be stored back to memory in the same length as the inputs?
  - Store back upper 4 Bit of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

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

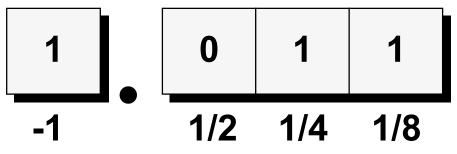
Might some other type of number system behave better? Is there a number system where the results of a multiplication have bounds?

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.



Yes: Binary Fractions



= -1 + 1/4 + 1/8 = -5/8

Fractions have the nice property that
fraction x fraction = fraction

11 - 9

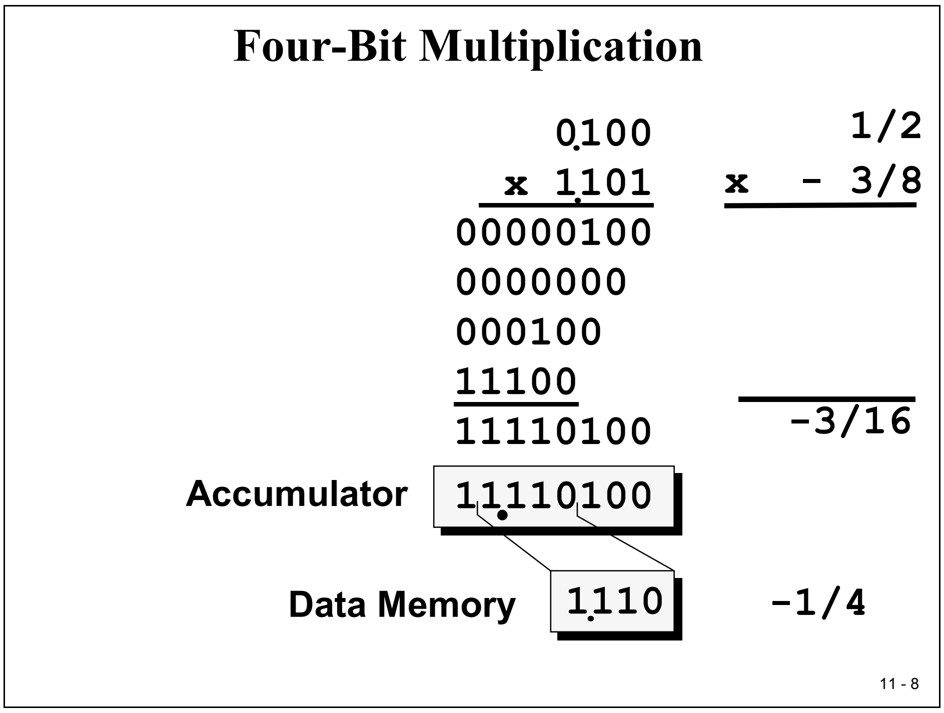
Multiplying Binary Fractions

When the C28x performs 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) and fractional part (Q – quotient). These type of fixed-point numbers are often called “IQ”-numbers, or for simplicity just Q-numbers.

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

I1Q3 * I1Q3 = I2Q6



Four-Bit Multiplication

```
               0.100          1/2
             x 1.101       x  - 3/8
            00000100
            0000000
            000100
            11100
            11110100        -3/16
Accumulator 11.110100
Data Memory   1.110          -1/4
```

11 - 8

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. No need to re-scale any intermediate results!

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

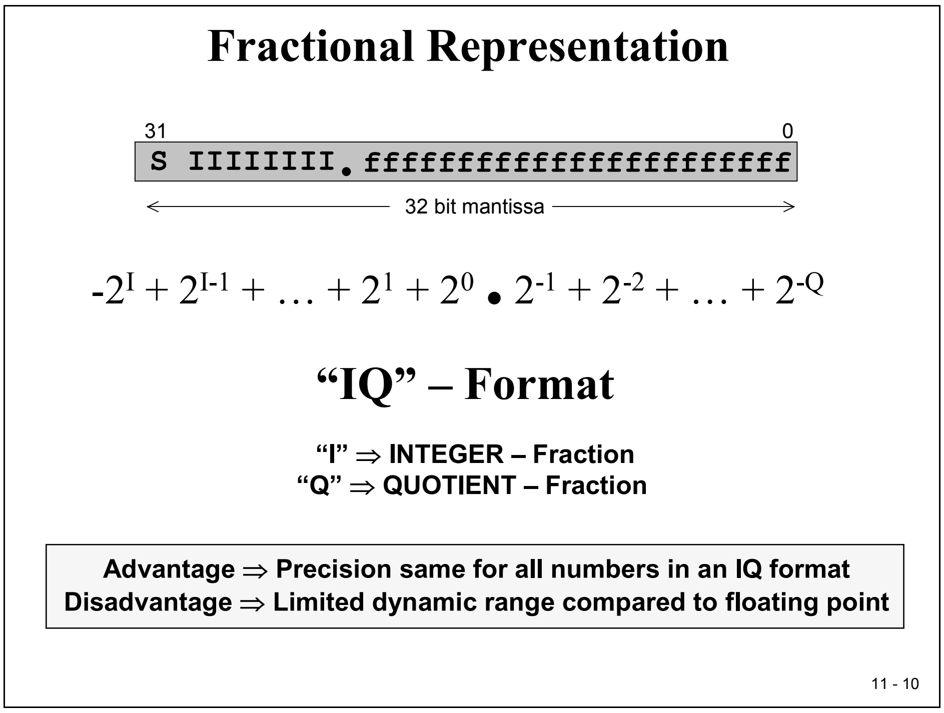
Disadvantage: The result might not be the exact one. As you can see from the slide above we will end up with (-4/16) stored back to Data Memory. Bits 2^{-4} to 2^{-6} are truncated. The correct result would have been (-3/16).

Recall that the 4-bit input operand multiplication operation is not the real size for the C28x, 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's 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 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 to use 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

```
31                                  0
S IIIIIII•ffffffffffffffffffffffff
<----------- 32 bit mantissa ----------->
```

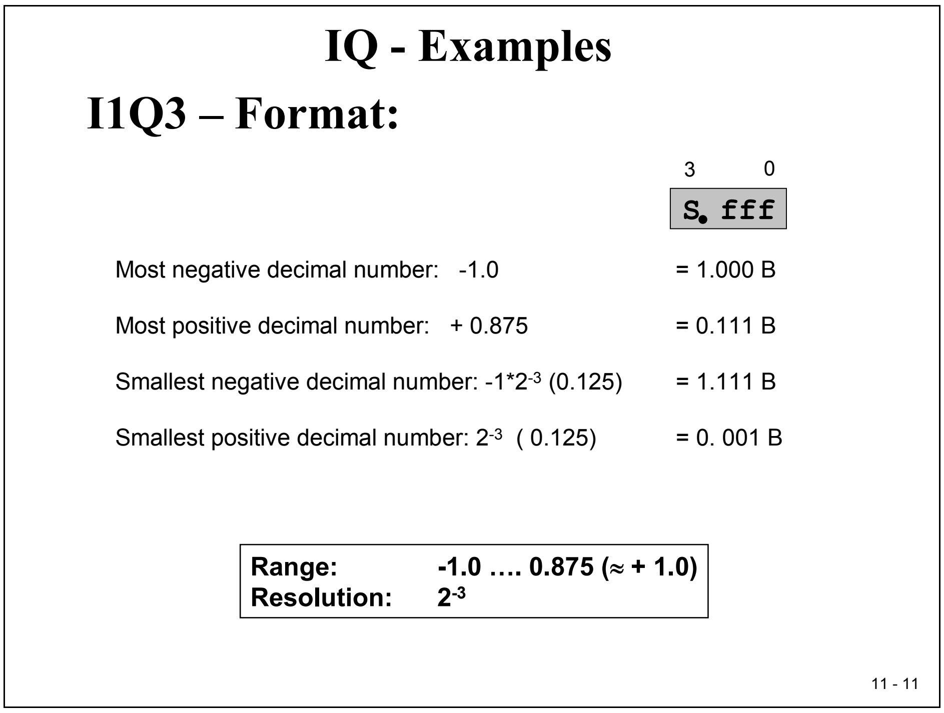
$$-2^{I} + 2^{I-1} + \dots + 2^{1} + 2^{0} \bullet 2^{-1} + 2^{-2} + \dots + 2^{-Q}$$

“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

11 - 10



IQ - Examples

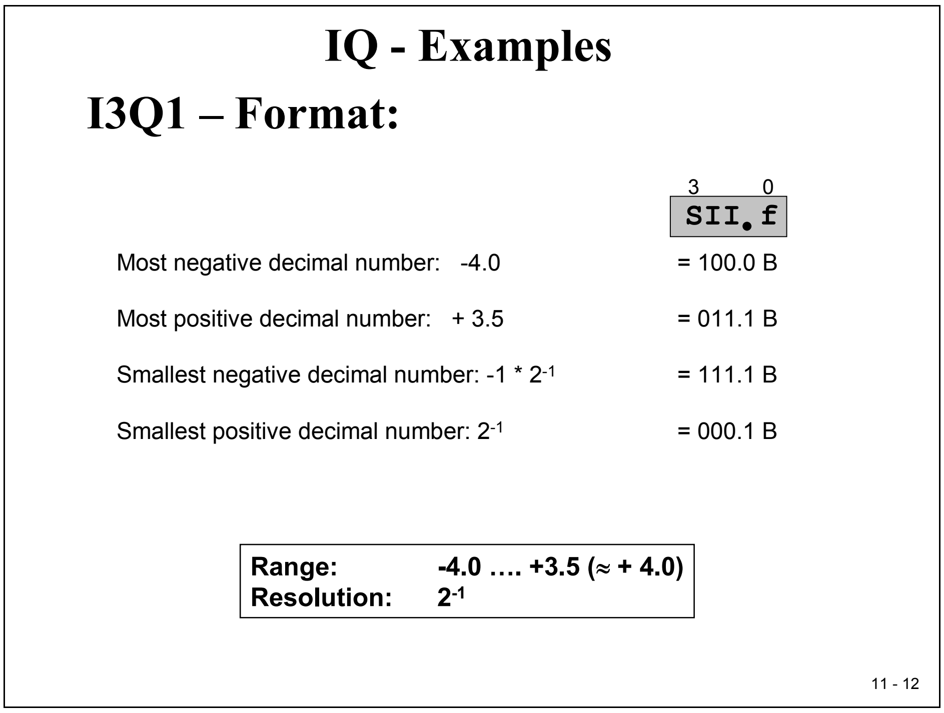
I1Q3 – Format:

```
3    0
S• fff
```

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}**

11 - 11



IQ - Examples

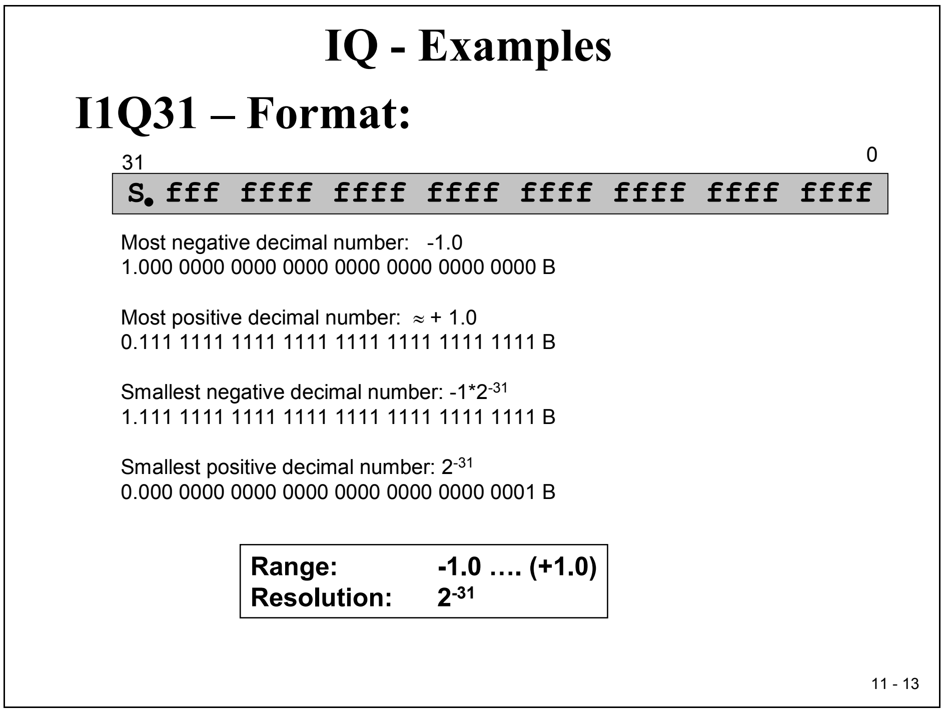
I3Q1 – Format:

```
3   0
SII.f
```

Most negative decimal number: -4.0	= 100.0 B
Most positive decimal number: + 3.5	= 011.1 B
Smallest negative decimal number: $-1 * 2^{-1}$	= 111.1 B
Smallest positive decimal number: 2^{-1}	= 000.1 B

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

11 - 12



IQ - Examples

I1Q31 – Format:

```
31                                         0
S.fff ffff ffff ffff ffff ffff ffff ffff
```

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

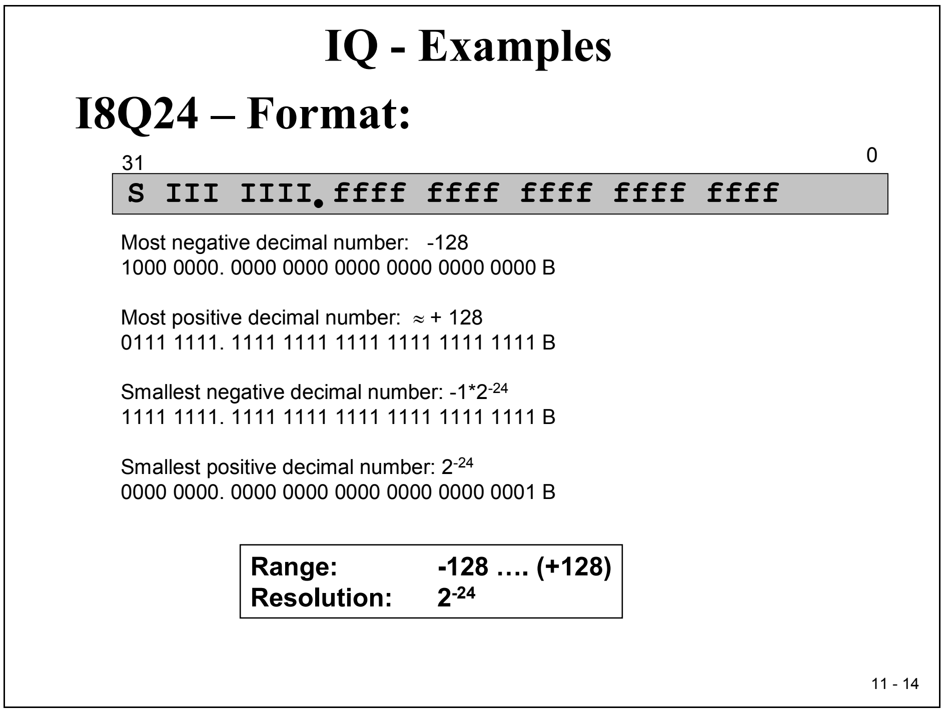
Most positive decimal number: ≈ + 1.0
0.111 1111 1111 1111 1111 1111 1111 1111 B

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}**

11 - 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

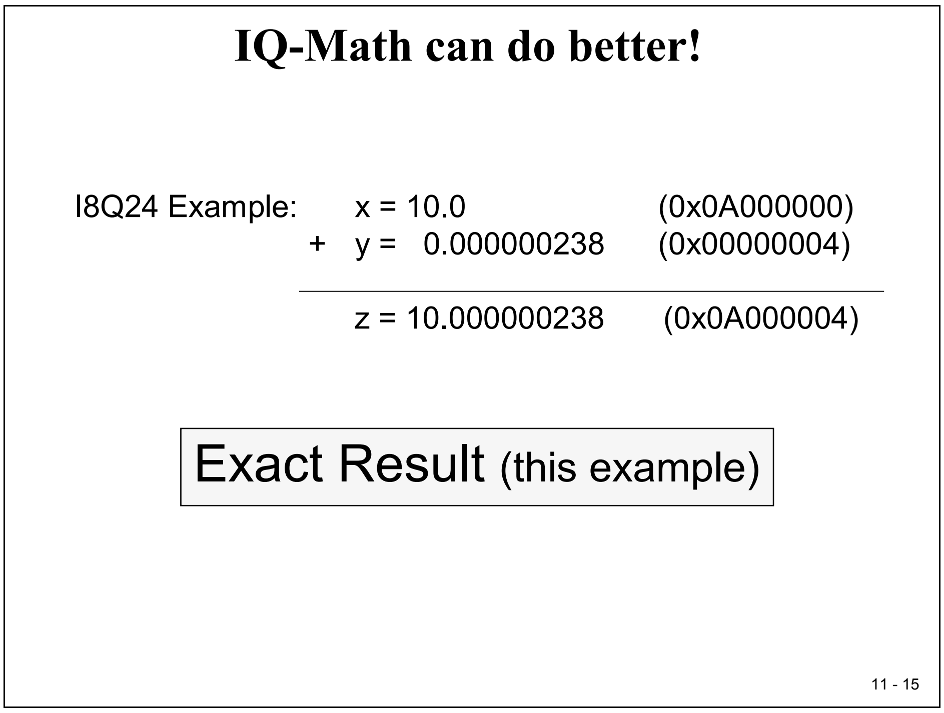
Smallest negative decimal number: $-1*2^{-24}$
1111 1111. 1111 1111 1111 1111 1111 1111 B

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

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

11 - 14

And to come back to 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 =   0.000000238      (0x00000004)
                 ------------------------------------
                   z = 10.000000238       (0x0A000004)
```

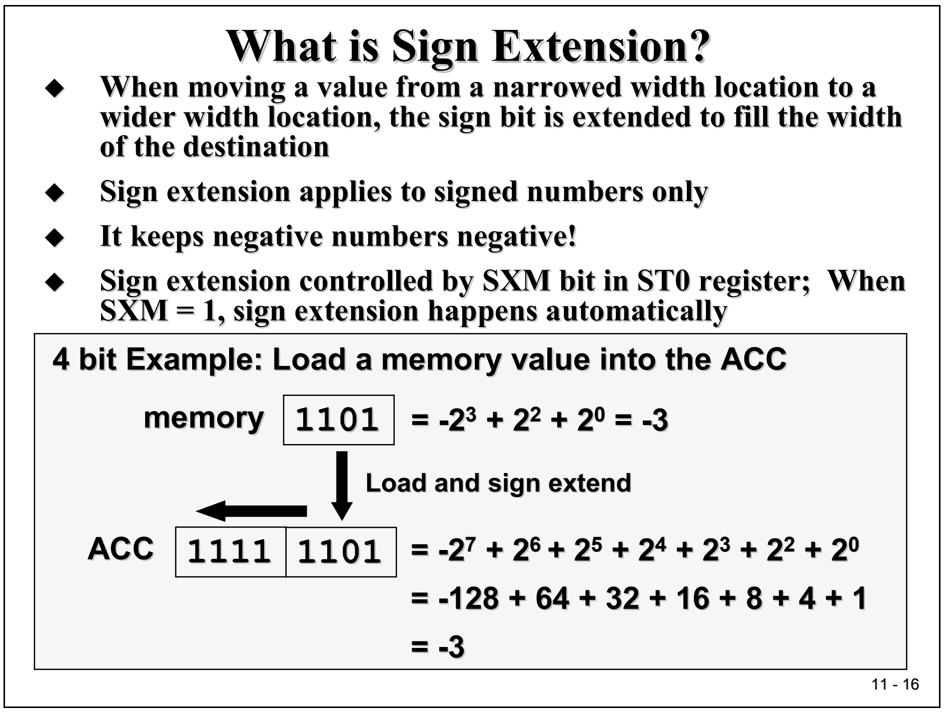
Exact Result (this example)

11 - 15

Sign Extension

When working with signed numbers it is important to keep the sign information when expanding an operand in its binary representation, for example from 4-bit to 8-bit, as shown in the next slide.

The C28x can operate on either unsigned binary or two's complement operands. The so-called "Sign Extension Mode (SXM)" identifies whether or not the sign extension process is used automatically when a number is processed internally. It is a good programming practice to always select the desired operating mode of SXM at the beginning of a subroutine or a module.



What is Sign Extension?

- When moving a value from a narrowed width location to a wider width location, the sign bit is extended to fill the width of the destination
- Sign extension applies to signed numbers only
- It keeps negative numbers negative!
- Sign extension controlled by SXM bit in ST0 register; When SXM = 1, sign extension happens automatically

4 bit Example: Load a memory value into the ACC

memory `1101` $= -2^3 + 2^2 + 2^0 = -3$

Load and sign extend

ACC `1111` `1101` $= -2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^0$

$= -128 + 64 + 32 + 16 + 8 + 4 + 1$

$= -3$

11 - 16

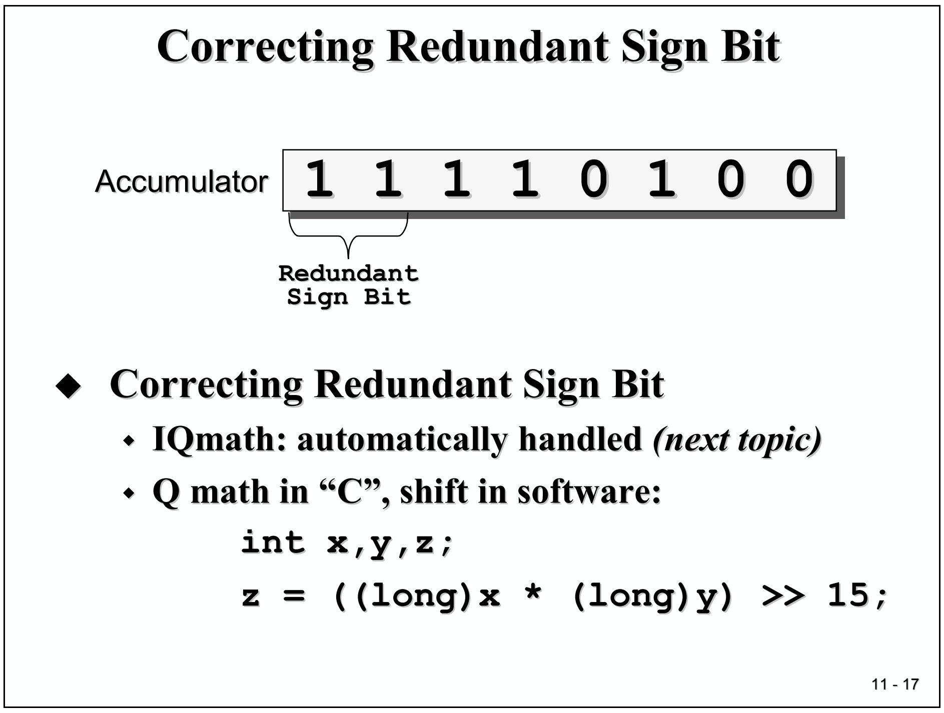
The SXM-Bit is part of ST0, one of the C28x status- and control registers. It can be accessed in assembly language only. To set or to clear it out of a C environment one can use the inline assembly function:

```
asm("        SETC SXM");

asm("        CLRC SXM");
```

Correcting the redundant sign bit

As we have already seen, when we multiply two I1Q3-numbers we end up with an I2Q6-result. Or, by multiplying two I1Q15 – numbers we end up with an I2Q30 – result. The second sign bit is always redundant. To adjust the result back to the format of the inputs we would have to apply shift operations, as shown in the next slide in a C environment. The shift operator (>> 15) will shift the intermediate result 15 times before it is typecast back to the data format of result variable z.

Texas Instruments "IQ-Math"-library, which will be explained in the rest of this module, takes care of this shift procedure internally. Again, we gain speed by using "IQ-Math".



Correcting Redundant Sign Bit

Accumulator $\underbrace{1\ 1}_{\text{Redundant Sign Bit}}\ 1\ 1\ 0\ 1\ 0\ 0$

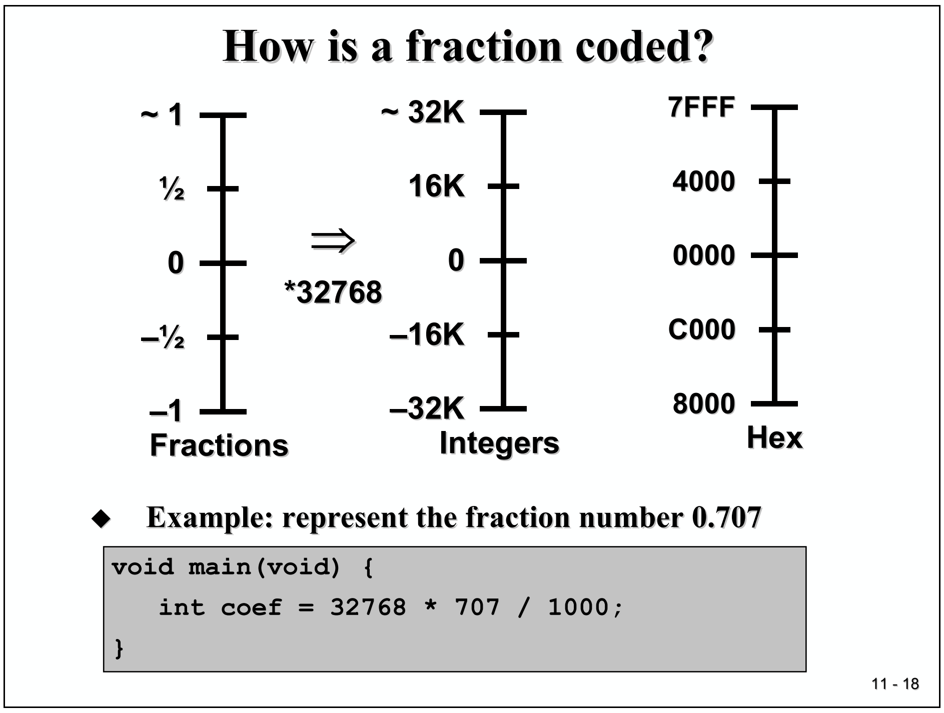
- Correcting Redundant Sign Bit
  - IQmath: automatically handled *(next topic)*
  - Q math in "C", shift in software:

```
int x,y,z;
z = ((long)x * (long)y) >> 15;
```

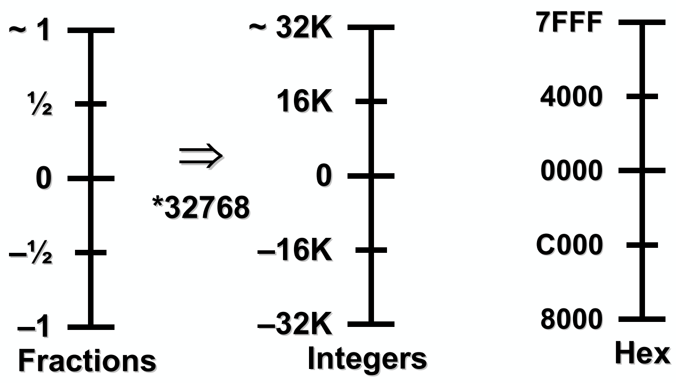
11 - 17

How do we code fractions in an ANSI-C environment? We do not have a dedicated data type, called '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, see next slide:



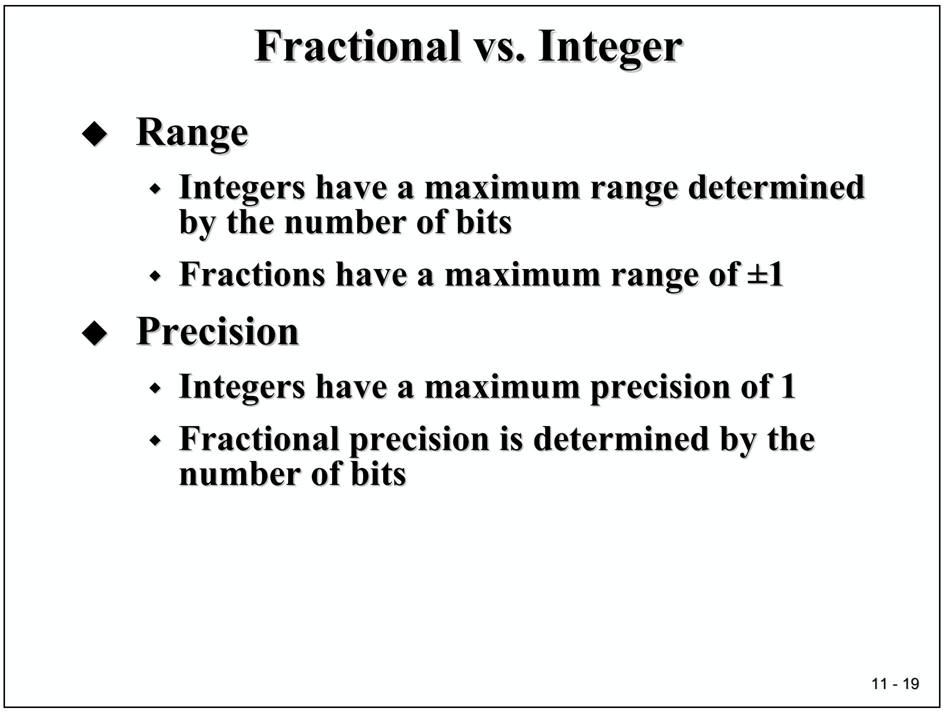
How is a fraction coded?



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

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

11 - 18



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**

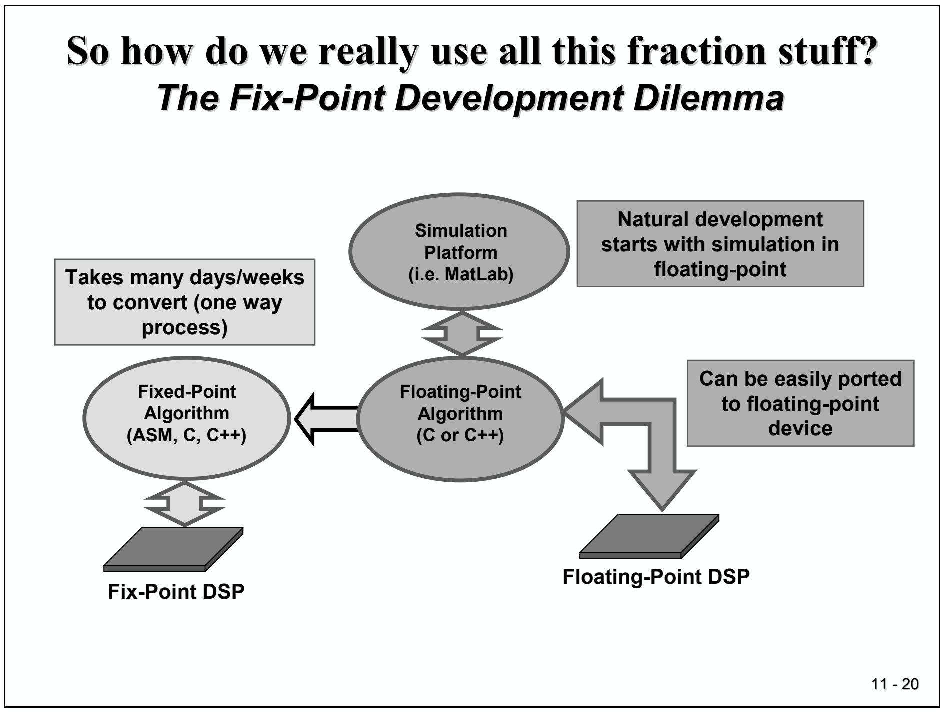
11 - 19

IQ – Math – Library

Implementing complex digital control algorithms on a Digital Signal Processor (DSP), or any other DSP capable processor, typically we come across the following issues:

- Algorithms are typically developed using floating-point math's
- Floating-point devices are more expensive than fixed-point devices
- Converting floating-point algorithms to a fixed-point device is very time consuming
- Conversion process is one way and therefore backward simulation is not always possible

The diagram below illustrates a typical development scenario in use today:



So how do we really use all this fraction stuff?

The Fix-Point Development Dilemma

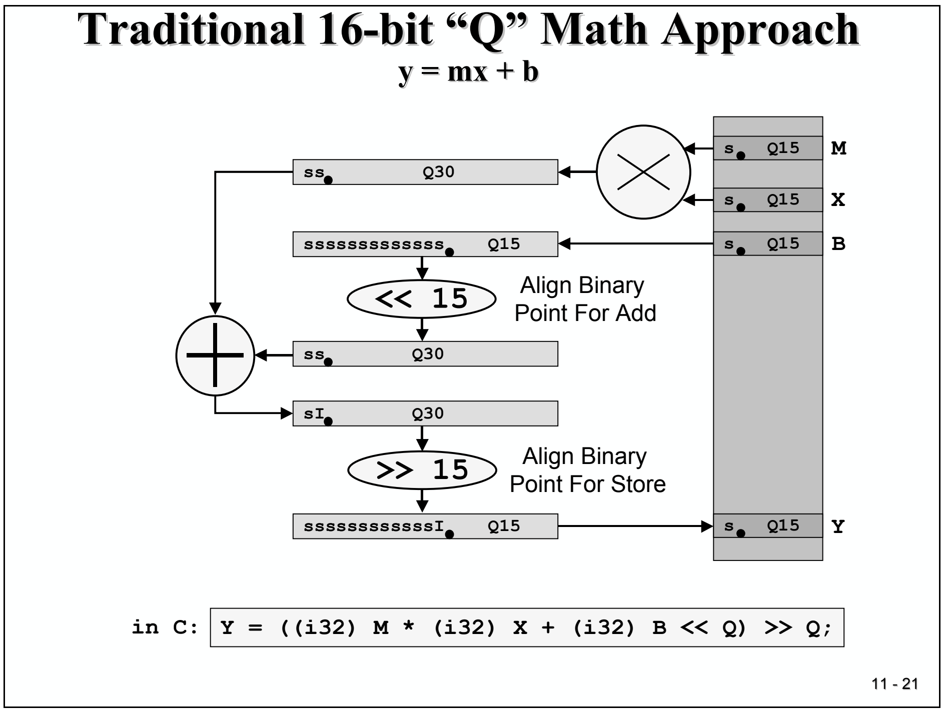
Simulation Platform (i.e. MatLab)
Natural development starts with simulation in floating-point
Takes many days/weeks to convert (one way process)
Fixed-Point Algorithm (ASM, C, C++)
Floating-Point Algorithm (C or C++)
Can be easily ported to floating-point device
Fix-Point DSP
Floating-Point DSP
11 - 20

The design may initially start with a simulation (i.e. MatLab) of a control algorithm, which typically would be written in floating-point math (C or C++). This algorithm can be easily ported to a floating-point device. However, because of the commercial reality of cost constraints, most likely a 16-bit or 32-bit fixed-point device would be used in many target systems.

The effort and skill involved in converting a floating-point algorithm to function using a 16-bit or 32-bit fixed-point device is quite significant. A great deal of time (many days or weeks) would be needed for reformatting, scaling and coding the problem. Additionally, the final implementation typically has little resemblance to the original algorithm. Debugging is not an easy task and the code is not easy to maintain or document.

Standard ANSI – C 16-Bit Mathematics

If the processor of your choice is a 16-bit fixed-point and you don't want to include a lot of library functions in your project, a typical usage of binary fractions is shown next. We assume that the task is to solve the equation Y = MX + B. This type of equation can be found in almost every mathematical approach for digital signal processing.



```
in C: Y = ((i32) M * (i32) X + (i32) B << Q) >> Q;
```

The diagram shows the transformations, which are needed to adjust the binary point in between the steps of this solution. We assume that the input numbers are in I1Q15-Format. After M is multiplied by X, we have an intermediate product in I2Q30-format. Before we can add variable B, we have to align the binary point by shifting b 15 times to the left. Of course we need to typecast B to a 32-bit long first to keep all bits of B. The sum is still in I2Q30-format. Before we can store back the final result into Y we have to right shift the binary point 15 times.

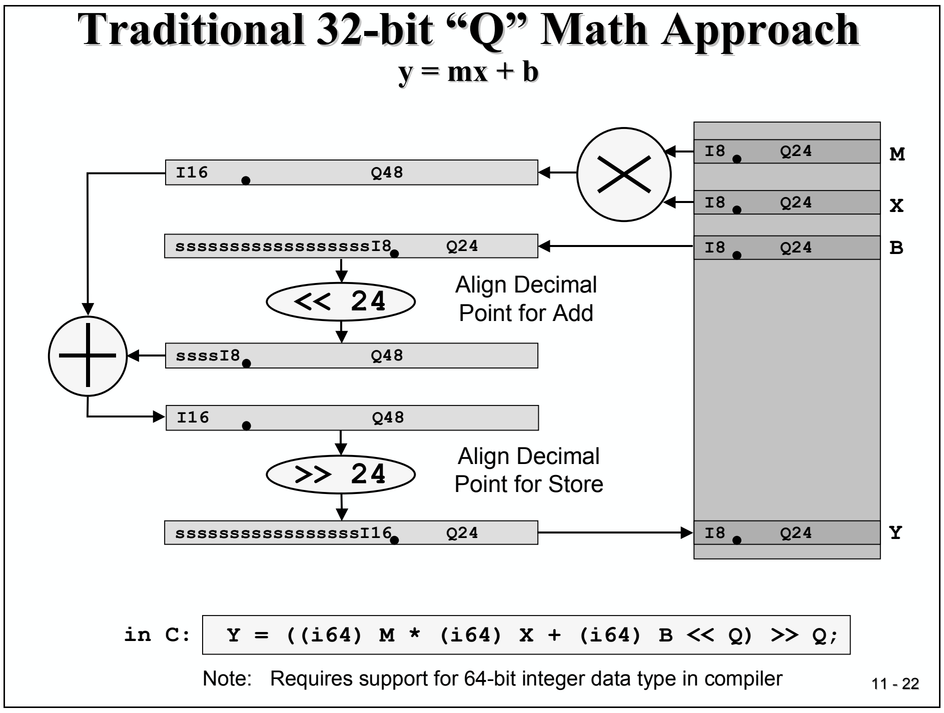
The last line of the slide shows the equivalent syntax in ANSI-C. "i32" stands for a 32-bit integer, usually called 'long'. 'Q' is a global constant and gives the number of fractional bits; in our example Q is equal to 15.

The disadvantage of this Q15 – approach is its limitation to 16 bits. A lot of projects for digital signal processing and digital control will not be able to achieve stable behavior due to the lack of either resolution or dynamic range.

The C28x as a 32-bit processor can do better – we just have to expand the scheme to 32-bit binary fractions!

Standard ANSI – C 32-Bit Mathematics

The next diagram is an expansion of the previous scheme to 32-bit input values. Again, the task is to solve equation Y = MX +B. In the following example the input numbers are in an I8Q24-format.



Traditional 32-bit "Q" Math Approach

y = mx + b

```
in C:   Y = ((i64) M * (i64) X + (i64) B << Q) >> Q;
```

Note: Requires support for 64-bit integer data type in compiler

11 - 22

The big problem with the translation into ANSI-C code is that we do not have a 64-bit integer data type! Although the last line of the slide looks pretty straight forward, we can't apply this line to a standard C-compiler!

What now?

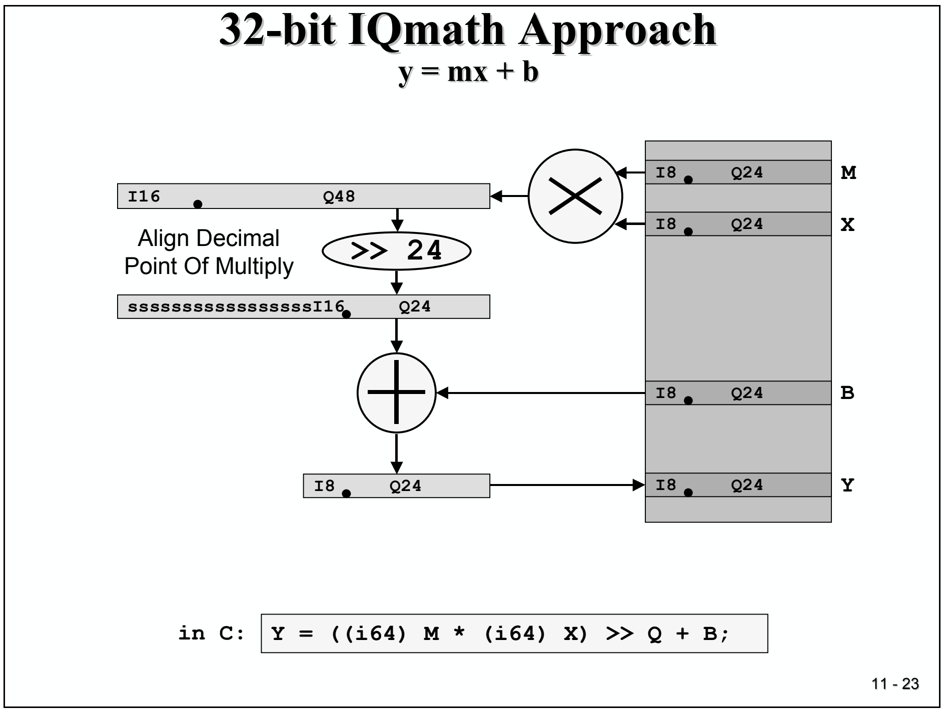
The rescue is the internal hardware arithmetic (Arithmetic Logic Unit and 32-bit by 32-bit Hardware Multiply Unit) of the C28x. These units are able to deal with 64-bit intermediate results in a very efficient way. Dedicated assembly language instructions for multiply and add operations are available to operate on the integer part and the fractional part of the 64-bit number.

To be able to use these advanced instructions, we have to learn about the C28x assembly language in detail. Eventually your professor offers an advanced course in C28x assembly language programming -

OR, just use Texas Instruments "IQ-Math"-library, which is doing nothing more than using these advanced assembly instructions!

32-Bit IQ – Math Approach

The first step to solve the 64-bit dilemma is to refine the last diagram for the 32-bit solution of Y = MX +B. As you can see from the next slide the number of shift operations is reduced to 1. Again, the C-line includes a 64-bit 'long', which is not available in standard C.



32-bit IQmath Approach

y = mx + b

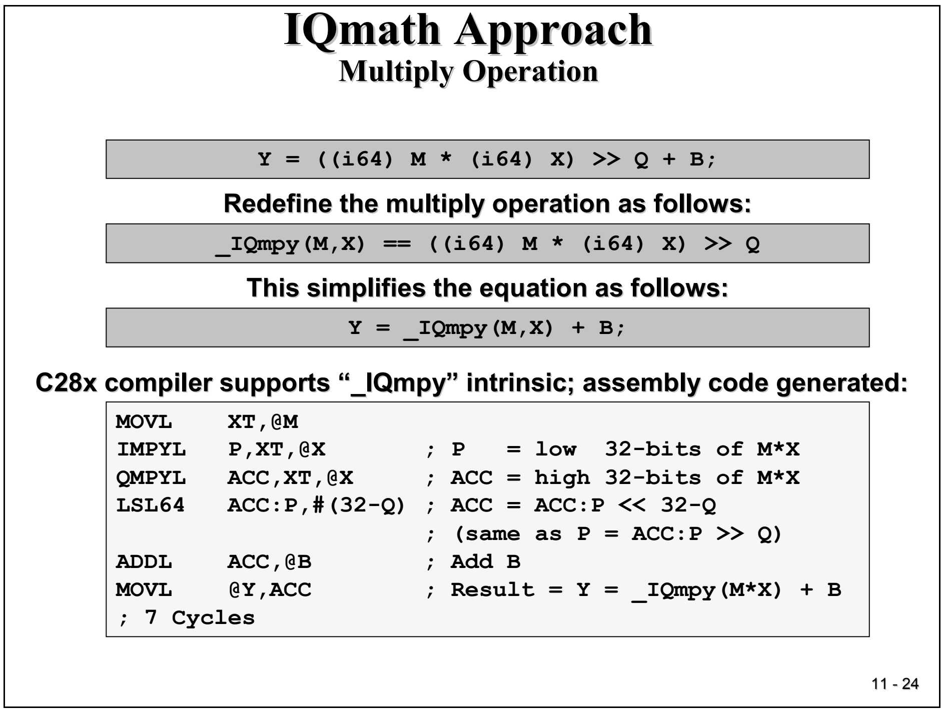
in C:
```
Y = ((i64) M * (i64) X) >> Q + B;
```

11 - 23

The "IQ"-Math approach 'redefines' the multiply operation to use the advantages of the internal hardware of the C28x. As stated, the C28x is internally capable of handling 64-bit fixed-point numbers with dedicated instruction sets. Texas Instruments provides a collection of intrinsic functions, one of them to replace the standard multiply operation by an _IQmpy(M,X) –line. Intrinsic means, we do not 'call' a function with a lot of context save and restore; instead the machine code instructions are directly included in our source code.

As you can see from the next slide the final C-code looks much better now without the cumbersome shift operations that we have seen in the standard C approach.

AND: The execution time of the final machine code for the whole equation Y = MX + B takes only 7 cycles – with a 150MHz C28x this translates into 46 nanoseconds!



IQmath Approach

Multiply Operation

```
Y = ((i64) M * (i64) X) >> Q + B;
```

Redefine the multiply operation as follows:

```
_IQmpy(M,X) == ((i64) M * (i64) X) >> Q
```

This simplifies the equation as follows:

```
Y = _IQmpy(M,X) + B;
```

C28x compiler supports "_IQmpy" intrinsic; assembly code generated:

```
MOVL    XT,@M
IMPYL   P,XT,@X        ; P   = low  32-bits of M*X
QMPYL   ACC,XT,@X      ; ACC = high 32-bits of M*X
LSL64   ACC:P,#(32-Q)  ; ACC = ACC:P << 32-Q
                       ; (same as P = ACC:P >> Q)
ADDL    ACC,@B         ; Add B
MOVL    @Y,ACC         ; Result = Y = _IQmpy(M*X) + B
; 7 Cycles
```

11 - 24

Let's have a closer look to the assembly instructions used in the example above.

The first instruction 'MOVL XT,@M' is a 32-bit load operation to fetch the value of M into a temporary register 'XT'.

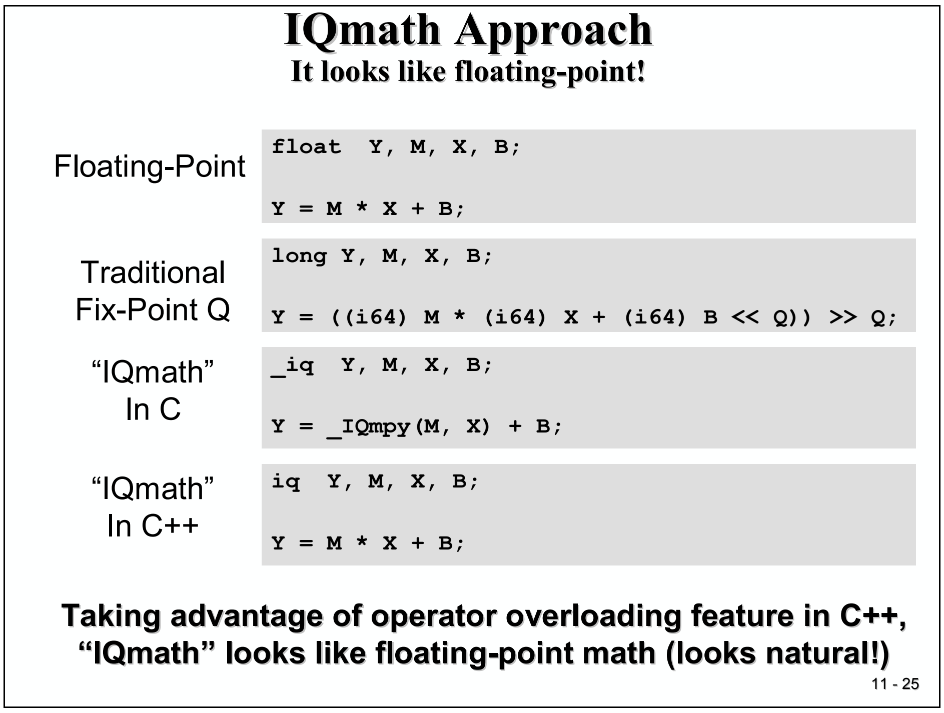
Next, 'XT' is multiplied by another 32-bit number taken from variable X ('IMPYL P,XT,@X'). When multiplying two 32-bit numbers, the result is a 64-bit number. In the case of this instruction, the lower 32-bit of the result are stored in a register 'P'.

The upper 32 bits are stored with the next instruction ('QMPYL ACC,XT,@X') in the 'ACC' register. 'QMPYL' is doing the same multiplication once more but keeps the upper half of the result only. At the end, we have stored all 64 bits of the multiplication in register combination ACC:P.

What follows is the adjustment of the binary point. The 64-bit result in ACC:P is in I16Q48-fractional format. Shifting it 32-24 times to the left, we derive an I8Q56-format. The instruction 'ADDL ACC,@B' uses only the upper 32 Bits of the 64-bit, thus reducing our fractional format from I8Q56 to I8Q24 – which is the same format as we use for B and all our variables!

The whole procedure takes only 7 cycles!

The next slide compares the different approaches. The IQ-Math library also defines a new data type '_iq' to simplify the definition of fractional data. If you choose to use C++ the floating-point equation and the C++ equation are identical! This is possible due to the overload feature of C++. The floating-point multiply operation is overloaded with its IQ-Math replacement – the code looks 'natural'.



IQmath Approach

It looks like floating-point!

Floating-Point

```
float  Y, M, X, B;

Y = M * X + B;
```

Traditional Fix-Point Q

```
long Y, M, X, B;

Y = ((i64) M * (i64) X + (i64) B << Q)) >> Q;
```

"IQmath" In C

```
_iq  Y, M, X, B;

Y = _IQmpy(M, X) + B;
```

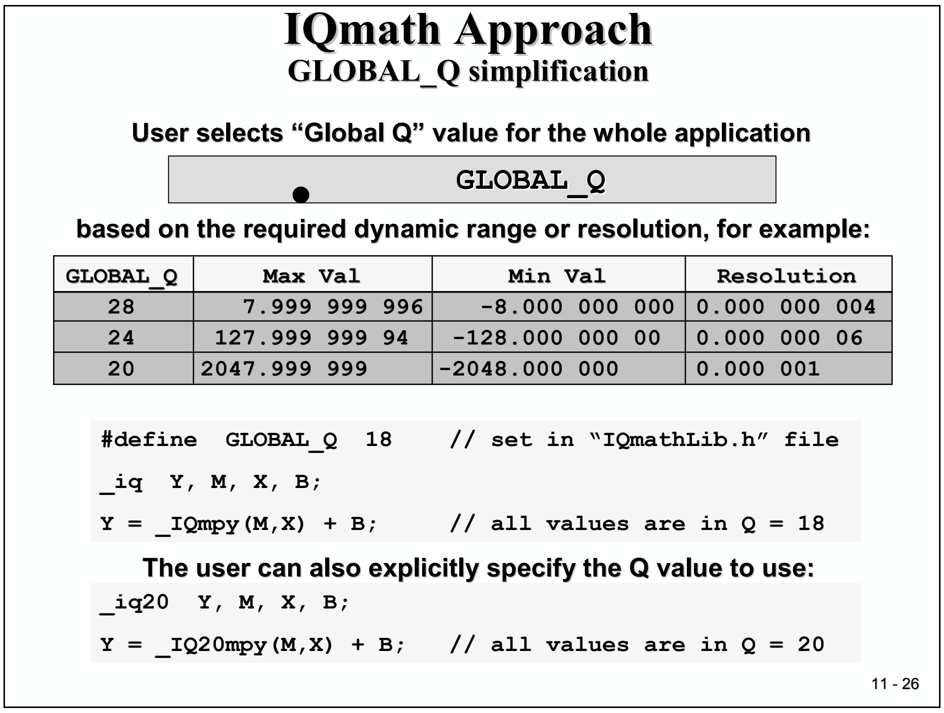
"IQmath" In C++

```
iq  Y, M, X, B;

Y = M * X + B;
```

Taking advantage of operator overloading feature in C++, "IQmath" looks like floating-point math (looks natural!)

11 - 25

This technique opens the way to generate a unified source code that can be compiled in a floating-point representation as well as into a fixed-point output solution. No need to translate a floating-point simulation code into a fixed-point implementation – the same source code can serve both worlds.



IQmath Approach
GLOBAL_Q simplification

User selects "Global Q" value for the whole application

```
GLOBAL_Q
```

based on the required dynamic range or resolution, for example:

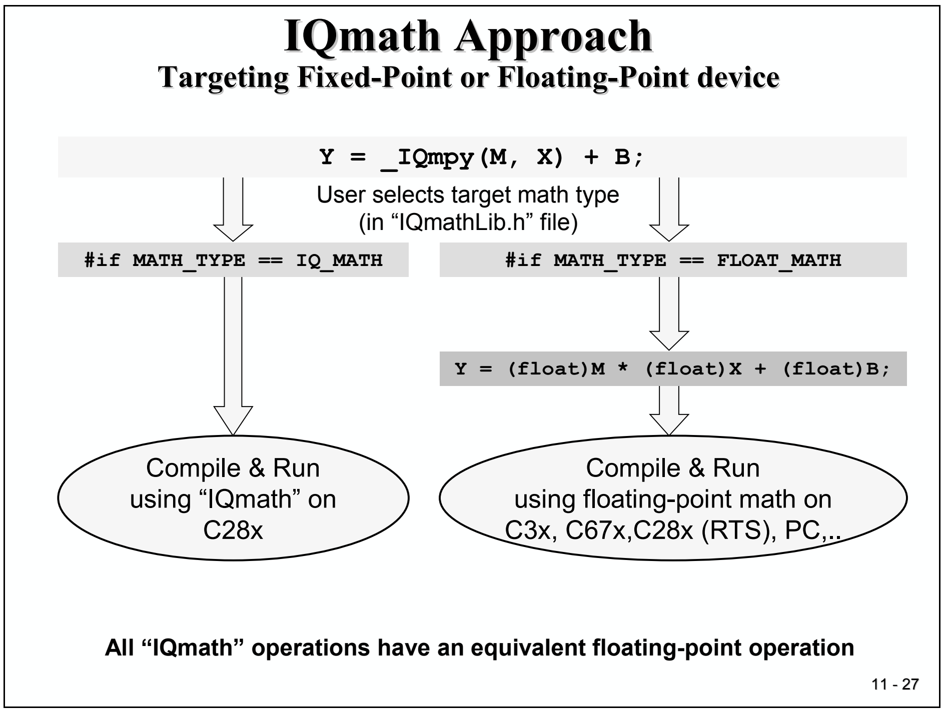
GLOBAL_Q	Max Val	Min Val	Resolution
28	7.999 999 996	-8.000 000 000	0.000 000 004
24	127.999 999 94	-128.000 000 00	0.000 000 06
20	2047.999 999	-2048.000 000	0.000 001

```
#define  GLOBAL_Q  18    // set in "IQmathLib.h" file
_iq  Y, M, X, B;
Y = _IQmpy(M,X) + B;     // all values are in Q = 18
```

The user can also explicitly specify the Q value to use:

```
_iq20  Y, M, X, B;
Y = _IQ20mpy(M,X) + B;   // all values are in Q = 20
```

11 - 26



IQmath Approach
Targeting Fixed-Point or Floating-Point device

```
Y = _IQmpy(M, X) + B;
```

User selects target math type (in "IQmathLib.h" file)

```
#if MATH_TYPE == IQ_MATH
```

```
#if MATH_TYPE == FLOAT_MATH
```

```
Y = (float)M * (float)X + (float)B;
```

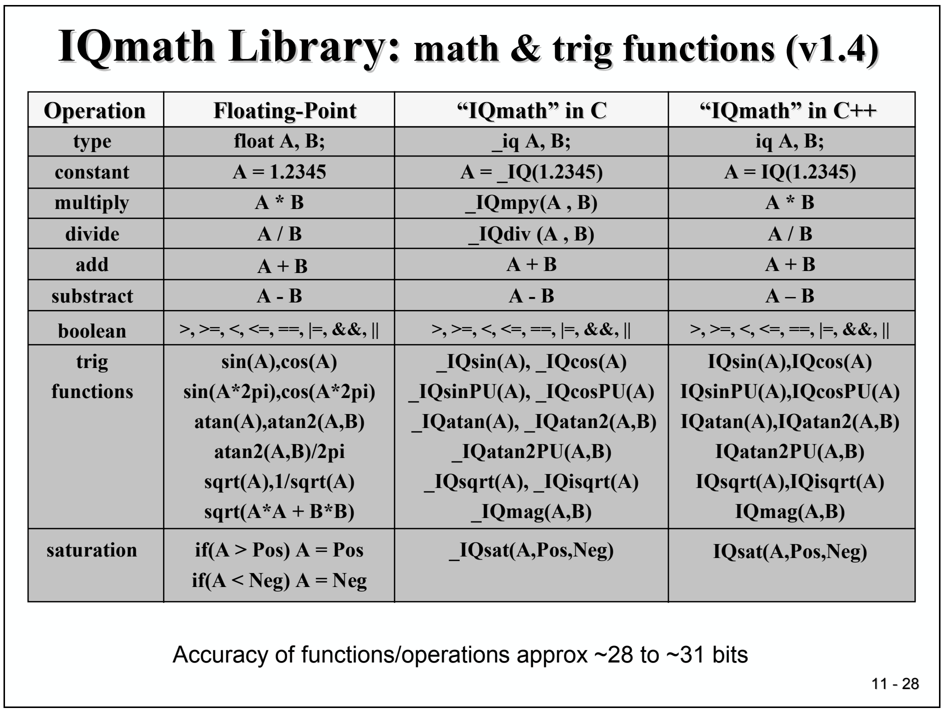
Compile & Run using "IQmath" on C28x
Compile & Run using floating-point math on C3x, C67x,C28x (RTS), PC,..

All "IQmath" operations have an equivalent floating-point operation

11 - 27

IQ – Math Library Functions

The next two slides summarize the existing library functions of IQ-Math.



IQmath Library: math & trig functions (v1.4)

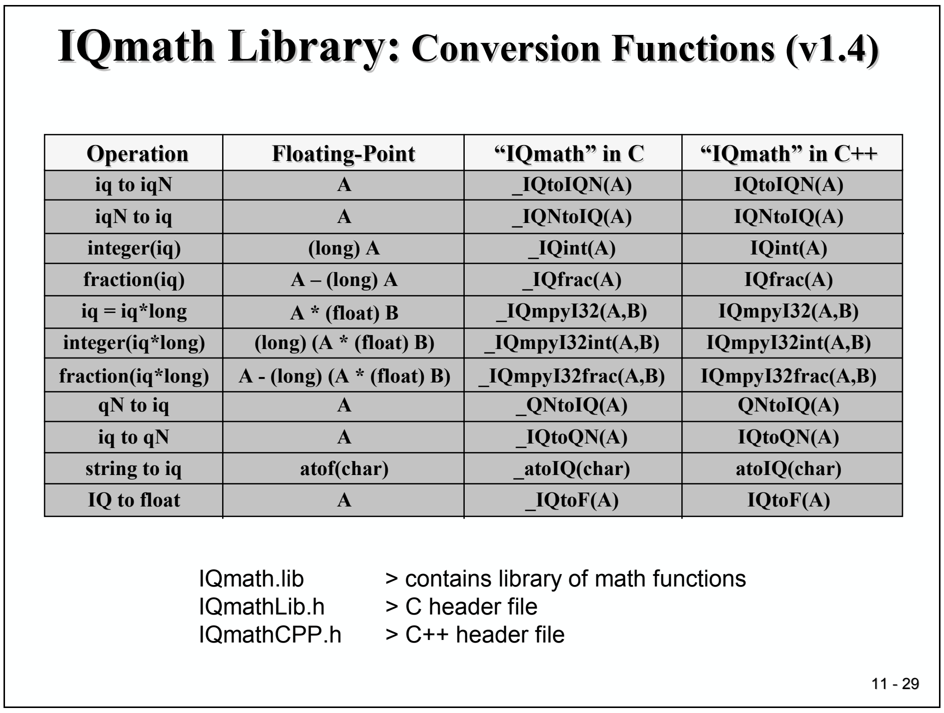
Operation	Floating-Point	"IQmath" in C	"IQmath" in C++
type	float A, B;	_iq A, B;	iq A, B;
constant	A = 1.2345	A = _IQ(1.2345)	A = IQ(1.2345)
multiply	A * B	_IQmpy(A , B)	A * B
divide	A / B	_IQdiv (A , B)	A / B
add	A + B	A + B	A + B
substract	A - B	A - B	A – B
boolean	>, >=, <, <=, ==, |=, &&, ||	>, >=, <, <=, ==, |=, &&, ||	>, >=, <, <=, ==, |=, &&, ||

trig functions	sin(A),cos(A)
	sin(A*2pi),cos(A*2pi)
	atan(A),atan2(A,B)
	atan2(A,B)/2pi
	sqrt(A),1/sqrt(A)
	sqrt(A*A + B*B)
		_IQsin(A), _IQcos(A)
		_IQsinPU(A), _IQcosPU(A)
		_IQatan(A), _IQatan2(A,B)
		_IQatan2PU(A,B)
		_IQsqrt(A), _IQisqrt(A)
		_IQmag(A,B)
			IQsin(A),IQcos(A)
			IQsinPU(A),IQcosPU(A)
			IQatan(A),IQatan2(A,B)
			IQatan2PU(A,B)
			IQsqrt(A),IQisqrt(A)
			IQmag(A,B)

saturation	if(A > Pos) A = Pos
	if(A < Neg) A = Neg
		_IQsat(A,Pos,Neg)	IQsat(A,Pos,Neg)

Accuracy of functions/operations approx ~28 to ~31 bits

11 - 28



IQmath Library: Conversion Functions (v1.4)

Operation	Floating-Point	"IQmath" in C	"IQmath" in C++
iq to iqN	A	_IQtoIQN(A)	IQtoIQN(A)
iqN to iq	A	_IQNtoIQ(A)	IQNtoIQ(A)
integer(iq)	(long) A	_IQint(A)	IQint(A)
fraction(iq)	A – (long) A	_IQfrac(A)	IQfrac(A)
iq = iq*long	A * (float) B	_IQmpyI32(A,B)	IQmpyI32(A,B)
integer(iq*long)	(long) (A * (float) B)	_IQmpyI32int(A,B)	IQmpyI32int(A,B)
fraction(iq*long)	A - (long) (A * (float) B)	_IQmpyI32frac(A,B)	IQmpyI32frac(A,B)
qN to iq	A	_QNtoIQ(A)	QNtoIQ(A)
iq to qN	A	_IQtoQN(A)	IQtoQN(A)
string to iq	atof(char)	_atoIQ(char)	atoIQ(char)
IQ to float	A	_IQtoF(A)	IQtoF(A)

IQmath.lib	> contains library of math functions
IQmathLib.h	> C header file
IQmathCPP.h	> C++ header file

11 - 29

IQ- Math Application : Field Orientated Control

The next slides are just to demonstrate the ability of “IQ-Math” to solve advanced numeric calculations in real time. The example is taken from the area of digital motor control. We will not go into the details of the control scheme and we will not discuss the various options to control an electrical motor. If you are a student of an electrical engineering degree you might be familiar with these control techniques. Eventually your university also offers additional course modules with this topic. The field of motor and electrical drive control is quite dynamic and offers a lot of job opportunities.

The next slide is a block diagram of a control scheme for an alternating current (AC) induction motor. These types of motors are based on a three-phase voltage system. Modern control schemes are introduced these days to improve the efficiency of the motor. One principle, called “Space Vector Modulation” or “Field Orientated Control” is quite popular today. In fact this theory is almost 70 years old now, but in the past it was impossible to realize a real time control due to the lack of computing power. Now with a controller like the C28x, it can be implemented!

AC Induction Motor Example

One of the more complex motor control algorithms

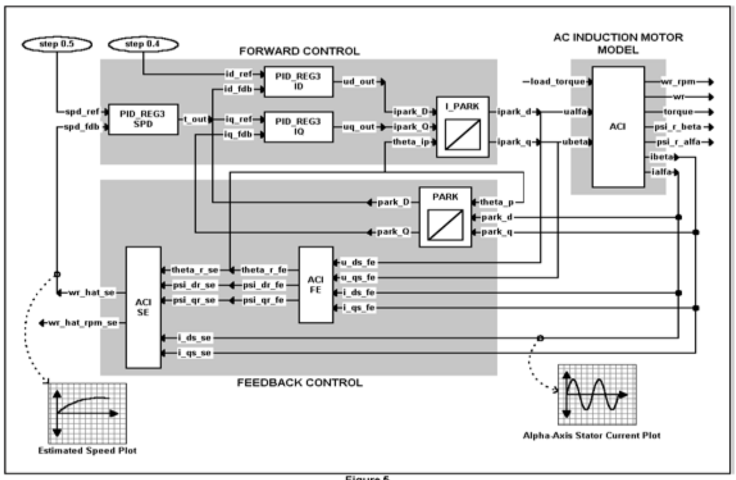


Figure 5

- **Sensorless, ACI induction machine direct rotor flux control**
- **Goal: motor speed estimation & alpha-axis stator current estimation**

11 - 30

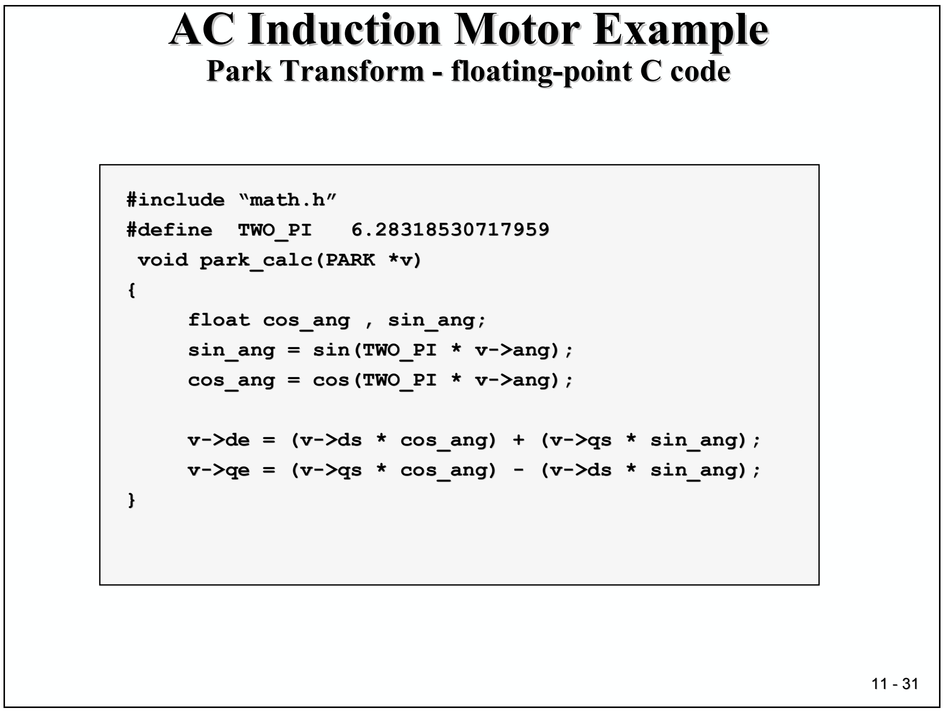
The core control system consists of three digital PID-controllers, one for the speed control of the motor (“PID_REG3 SPD”), one to control the torque (“PID_REG3 IQ”) and one for the flux (“PID_REG3 ID”). Between the control loops and the motor two coordinate transforms are performed (“PARK” and “I_PARK”).

Let’s have a look into a standard C implementation of the PARK transform, which converts a 3-D vector to a 2-D vector. For now, it is not necessary to fully understand this transform, just have a look into the mathematical operations involved.

All variables are data type “float” and the functions included are:

- Six multiply operations,
- Two trigonometric function calls,
- An addition and
- A subtraction.

This code can easily be compiled by any standard C compiler and downloaded into a simulation or into any processor, for example the C28x. It will work, but it will not be the most efficient way to use the C28x because it will involve floating-point library function calls that will consume a considerable amount of computing time.



AC Induction Motor Example

Park Transform - floating-point C code

```
#include "math.h"
#define  TWO_PI   6.28318530717959
 void park_calc(PARK *v)
{
     float cos_ang , sin_ang;
     sin_ang = sin(TWO_PI * v->ang);
     cos_ang = cos(TWO_PI * v->ang);

     v->de = (v->ds * cos_ang) + (v->qs * sin_ang);
     v->qe = (v->qs * cos_ang) - (v->ds * sin_ang);
}
```

11 - 31

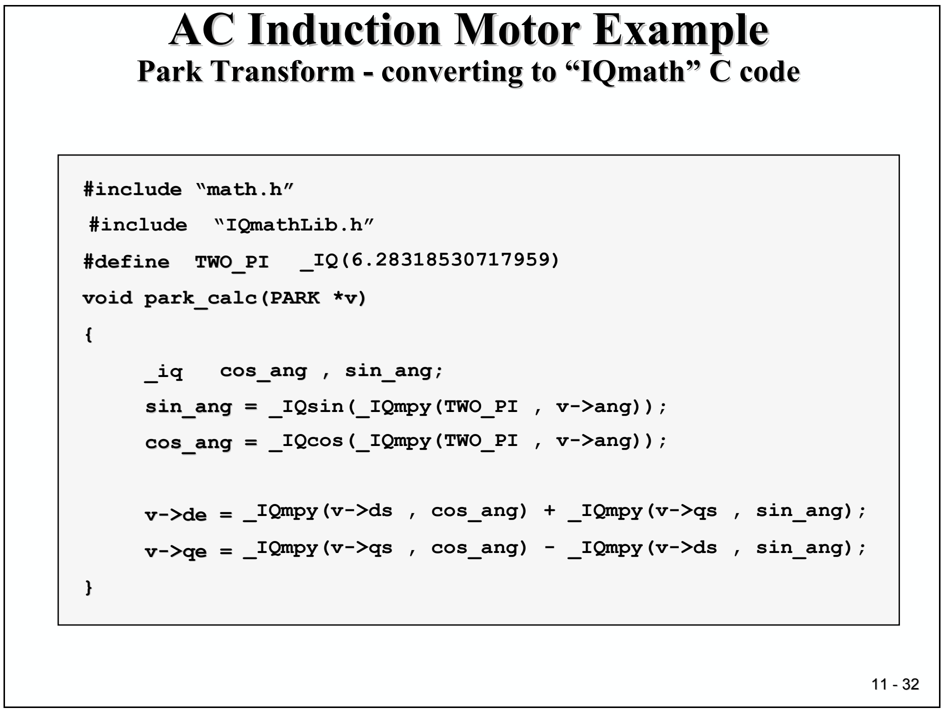
With the “IQ-Math” library we can improve the code for the C28x, as shown at the next slide. Of course, we have to replace all float function calls by “IQ-Math” intrinsics.

All variables are now of data type "_iq", the sine and cosine function calls are replaced by their intrinsic replacements as well as the six multiply operations.

The constant "TWO_PI" will be converted into the standard IQ-format with the conversion function "_IQ( )". This way the number 6.28 will be translated into the correct fixed-point scale before it is used during compilation.

The resulting code will be compiled into a much denser and faster code for the C28x. Of course, a little bit of coding is still needed to convert an existing floating-point code into the "IQ-Math" C-code.

Fortunately, the structure of the two program versions is identical, which helps to keep a development project consistent and maintainable, for both the floating-point and a fixed-point implementations.



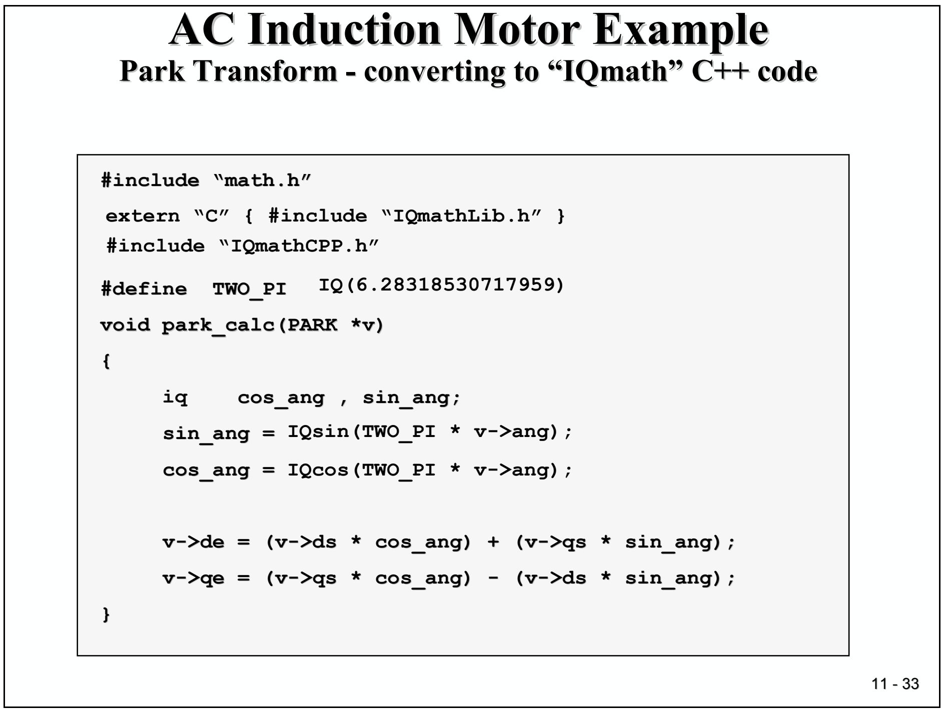
AC Induction Motor Example

Park Transform - converting to "IQmath" C code

```
#include "math.h"
#include "IQmathLib.h"
#define  TWO_PI   _IQ(6.28318530717959)
void park_calc(PARK *v)
{
     _iq   cos_ang , sin_ang;
     sin_ang = _IQsin(_IQmpy(TWO_PI , v->ang));
     cos_ang = _IQcos(_IQmpy(TWO_PI , v->ang));

     v->de = _IQmpy(v->ds , cos_ang) + _IQmpy(v->qs , sin_ang);
     v->qe = _IQmpy(v->qs , cos_ang) - _IQmpy(v->ds , sin_ang);
}
```

11 - 32

If we go further on and use a C++ compiler to translate the "IQ-Math" code, we can take advantage of the overload technique of C++. The result for this PARK-transform is shown at the next slide.



AC Induction Motor Example

Park Transform - converting to “IQmath” C++ code

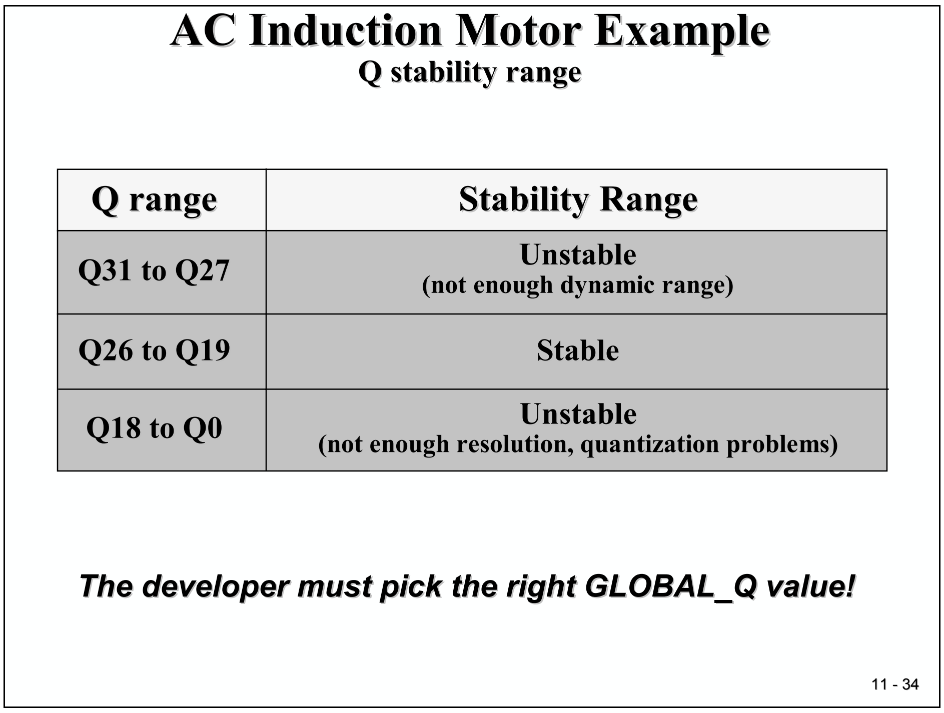
```
#include "math.h"
extern "C" { #include "IQmathLib.h" }
#include "IQmathCPP.h"
#define  TWO_PI  IQ(6.28318530717959)
void park_calc(PARK *v)
{
     iq    cos_ang , sin_ang;
     sin_ang = IQsin(TWO_PI * v->ang);
     cos_ang = IQcos(TWO_PI * v->ang);

     v->de = (v->ds * cos_ang) + (v->qs * sin_ang);
     v->qe = (v->qs * cos_ang) - (v->ds * sin_ang);
}
```

11 - 33

The multiply operation looks identical in floating-point and in fixed-point implementation. It is quite a simple and fast procedure to take any floating-point algorithm and convert it to an "IQ-Math" algorithm.

The complete system was coded using "IQ-Math". Based on analysis of coefficients in the system, the largest coefficient had a value of 33.3333. This indicated that a minimum dynamic range of 7bits (+/-64 range) was required. Therefore, this translated to a GLOBAL_Q value of 32-7 = 25(Q25). Just to be safe, the initial simulation runs were conducted with GLOBAL_Q = 24 (Q24) value.

Next, the whole AC induction motor solution was investigated for stability and dynamic behavior by changing the global Q value. With a 32-bit fixed-point data type we can modify the fractional part between 0 bit (“Q0”) and 31 bits (“Q31”). The results are shown below. As you can see, there is an area, in which all tests led to a stable operating mode of the motor. The two other areas showed an increasing degree of instability, caused by either not enough dynamic range in the integer part or not enough fractional resolution of the numbering system.



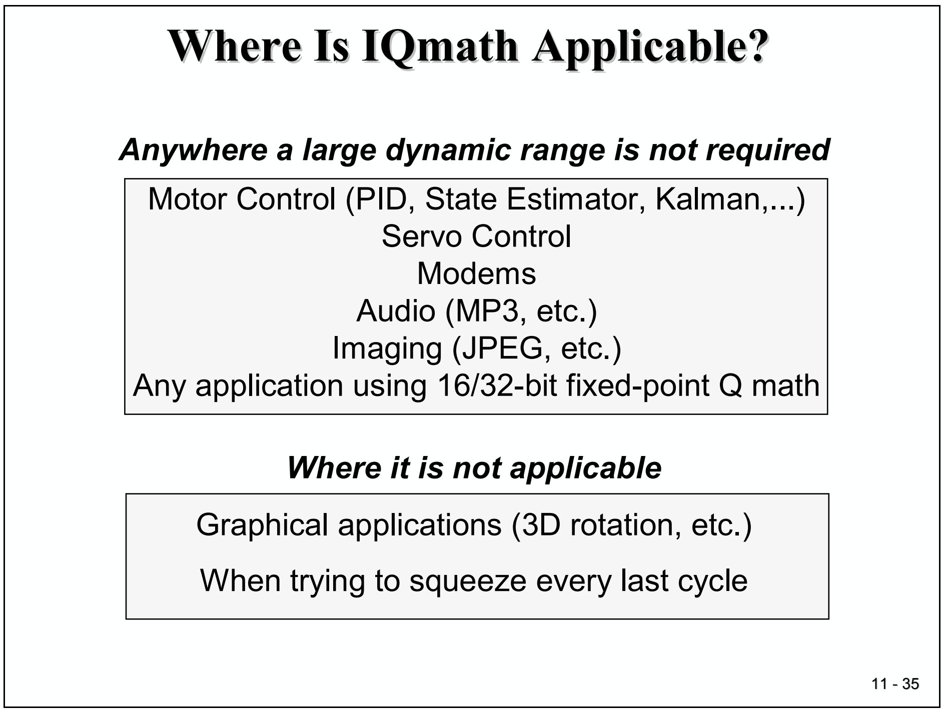
AC Induction Motor Example

Q stability range

Q range	Stability Range
Q31 to Q27	Unstable (not enough dynamic range)
Q26 to Q19	Stable
Q18 to Q0	Unstable (not enough resolution, quantization problems)

The developer must pick the right GLOBAL_Q value!

11 - 34



Where Is IQmath Applicable?

Anywhere a large dynamic range is not required

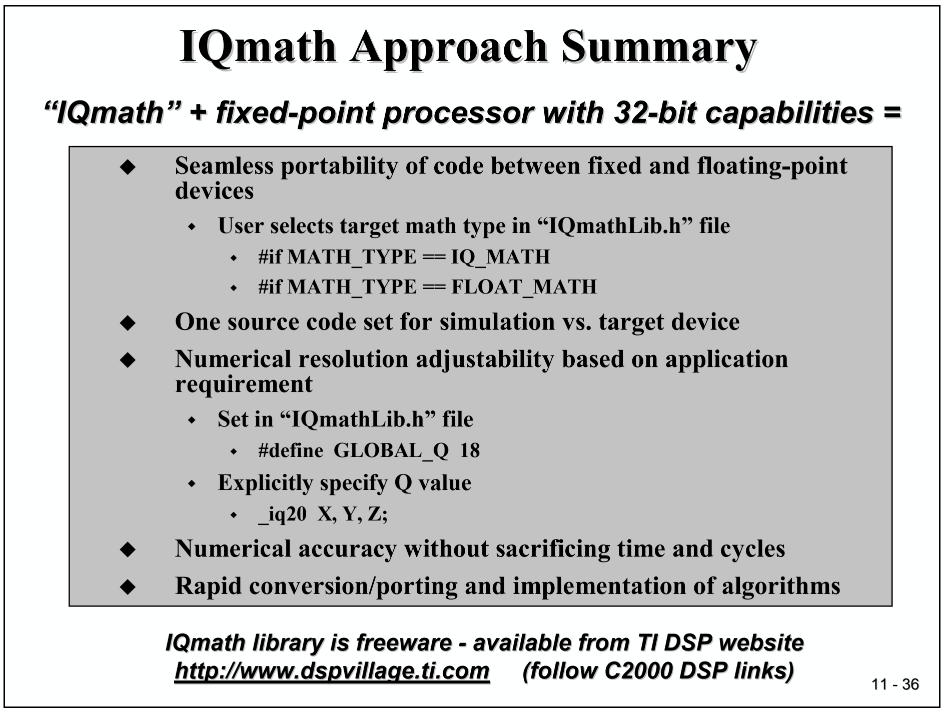
Motor Control (PID, State Estimator, Kalman,...)
Servo Control
Modems
Audio (MP3, etc.)
Imaging (JPEG, etc.)
Any application using 16/32-bit fixed-point Q math

Where it is not applicable

Graphical applications (3D rotation, etc.)

When trying to squeeze every last cycle

11 - 35



IQmath Approach Summary

"IQmath" + fixed-point processor with 32-bit capabilities =

- **Seamless portability of code between fixed and floating-point devices**
  - **User selects target math type in "IQmathLib.h" file**
    - **#if MATH_TYPE == IQ_MATH**
    - **#if MATH_TYPE == FLOAT_MATH**
- **One source code set for simulation vs. target device**
- **Numerical resolution adjustability based on application requirement**
  - **Set in "IQmathLib.h" file**
    - **#define GLOBAL_Q 18**
  - **Explicitly specify Q value**
    - **_iq20 X, Y, Z;**
- **Numerical accuracy without sacrificing time and cycles**
- **Rapid conversion/porting and implementation of algorithms**

IQmath library is freeware - available from TI DSP website
http://www.dspvillage.ti.com (follow C2000 DSP links)

11 - 36