Applications of signal processors

NUMBER SYSTEMS

Author: Grzegorz Szwoch

Gdańsk University of Technology, Department of Multimedia Systems

1 + 1 = 10

- A processor only performs bit operations:
  10011001100110 + 100110011001101 = 111001100110011
- The processor does not interpret what „10011001100110” means.
- Number system – a method of representing numbers with a sequence of 0s and 1s, so that the results are correct.
- The most important number systems:
  - fixed-point
    - – integer
    - – fractional (Q)
  - floating-point

Fixed-point integers

Unsigned integer representation in a binary notation:

- first, most significant bit (MSB): weight 2^{N-1}
- *i*-th bit: weight 2^{i-1}
- last, least significant bit (LSB): weight 1

2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^{9}	2^{8}	2^{7}	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}
32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1
0	0	1	0	0	1	1	0	0	1	1	0	0	1	1	0
		8192			1024	512			64	32			4	2	

= 9830

Fixed-point integers

Signed integer representation:

- MSB – a sign bit, 0: positive, 1: negative
- the remaining bits: an absolute value in two's complement (U2):
  - negate all bits (0 → 1, 1 → 0),
  - add 1.

Example for 8-bit numbers:

(-1) 0000 0001 → 1111 1110 → 1111 1111

(-123) 0111 1011 → 1000 0100 → 1000 0101

Features of fixed-point integers

- Resolution – the smallest difference between numbers, is equal to 2^0 = 1. So we can’t represent fractions!
- Range – the smallest and the largest number, depends on the number of bits *N*:
  - unsigned: 0 to (2^N-1)
  - signed: -2^{N-1} to $(2^{N-1} – 1)$
- 16-bit integers: 0 to 65535 (unsigned), -32768 to 32767 (signed).
- If we only need positive integers (e.g. in a counter), we can double the range by using the unsigned type.

Integer types in C

- *int* – number of bits depends on the processor registers length, in a DSP: usually 16 bits (2B) or 32 bits (4B).
- *char* – 8 bits, a byte (1B)
- *short* – 16 bits (2B), a word
- *long* – 32 bits (4B), a double word
- *long long* – in a DSP, usually equal to the accumulator length, often 40 bits (4.5B) or 64 bits (8B)

Each type has two versions:

- *unsigned*, e.g. *unsigned int*
- *signed* (the default), e.g. *signed long* (= *long*)

Byte order in memory

If a number takes 4 bytes, in what order are they stored in memory?

- From the most to the least significant byte (*big endian*) – some processors, network transmission.
- From the least to the most significant byte (*little endian*) – all Intel and (in default mode) ARM CPUs.
- In a DSP, we can usually choose either one during the code compilation.
- In most cases, DSP programs use *little endian*.

Range overflow

What happens if a number does not fit into the designated type?
A range overflow occurs, and some bits are lost.

Example: adding two 16-bit integers.

- Unsigned integers (max value: 65535)
  (65530 + 10) → [1] 0000 0000 0000 0100 → 4
  - „excessive” bits are removed,
  - the result is a remainder of division by 2^{16}.
- Signed integers (max value: 32767)
  (32760 + 10) → 1000 0000 0000 0010 → -32766 (!!!)
  - “1” overwrites the bit sign,
  - a range wrap occurs, the number becomes negative!.

How to prevent range overflow?

- DSP accumulators have additional “guard bits” (e.g. 40-bit accumulator: 32+8), which reduces the risk of overflow for intermediate results.
- When we write the result to the memory, we must choose a type of a sufficient length (e.g. *long*).
- On DSP, we often scale numbers, e.g. we divide the numbers by 2 before adding them, then we scale the result.

Fractional numbers

- So, is it not possible to represent fractional numbers, such as 0.3?
- Remember: a number system defines how the bits are interpreted.
- So, maybe we can interpret them this way?

16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
sign	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}	2^{-9}	2^{-10}	2^{-11}	2^{-12}	2^{-13}	2^{-14}	2^{-15}
0	0	1	0	0	1	1	0	0	1	1	0	0	1	1	0
+		2^{-2}			2^{-5}	2^{-6}			2^{-9}	2^{-10}			2^{-13}	2^{-14}	

$= 0{,}29998 \approx 0{,}3$

Q15 notation

- Notation of fractional numbers as a signed 16-bit sequence is called Q15 - 1 sign bit, 0 integer bits, 15 fractional bits.
- Resolution (minimum difference between numbers) is 2^{-15} = 0.000030517578125.
- Range: from -1 to $(1-2^{-15})$, to 0.999969482421875.
- A value of +1 is outside the range!
- All rules related to range overflow still apply.

QM.N notation

- We can extend this notation to a general form QM.N:
  1 sign bit, *M* integer bits, *N* fractional bits.
- Resolution: 2^{-N}.
- Range: -2^{M} to $(2^{M} – 2^{-N})$.
- Example: 1 integer bit and 14 fractional bits is Q1.14.
  Range: -2 to “near” 2.
- In a QM.N notation, the position of a decimal point is always the same. Hence the name “fixed point notation”.

Q notation with integers

- C language does not have a special type for QM.N notation.
- This notation is only a bit interpretation convention.
- In order to write a Q number, we must interpret the bits as if they represented a standard integer, e.g.:
  $0.3 \rightarrow 10011001100110 \rightarrow 9830$
- Which gives us a conversion rule (for QM.N):
  - from a fractional number *x* to an integer *q*:
    $q = x \cdot 2^N$ (with rounding)
  - from an integer *q* to a fractional number *x*:
    $x = q / 2^N$

Q notation with integers

Example conversion for Q15 ($2^N = 2^{15} = 32768$):

0.3 → 0.3 * 32768 → 9830.4 → 9830

0.6 → 0.6 * 32768 → 19660.8 → 19661

Computing an expression (0.3 + 0.6):

9830 + 19661 = 29491

Conversion to a fractional number:

29491 → 29491 / 32768 → 0.89999

(remember: the resolution is 0.00003)

Quantization

- After the conversion, the result must be rounded to the nearest value that has a representation:

  9830.4 → 9830   19660.8 → 19661

- This is called a value quantization.
- Quantization error: a difference between the quantized and the original values (-0.4, 0.2). It has a form of a noise
- If we use integers to store fractional numbers, the quantization error affects the accuracy of calculations.
- For example, in IIR digital filters, quantization noise may cause filter instability, despite a correct filter design.

Multiplication of QM.N numbers

To simplify, we will consider Q15 numbers.

How do we compute $(0.3 \cdot 0.6)$?

- Conversion as before.
- $9830 \cdot 19661 = 193267630$
- Multiplication of two Q15 numbers gives us a Q30 value!
- Multiplication of two QM.N numbers results in Q(2M).(2N) value.
- Conversion of a result:
  $193267630 \rightarrow 193267630 / 2^{30} \rightarrow 0.1799945$

Division of QM.N numbers

- Division is a very slooooow operation, especially in the fixed-point notation. It cannot be performed as directly as a multiplication.
- The most often approach is finding a reciprocal using a numeric algorithm, then performing a multiplication.
- There is an exception: dividing by a power of two (2^k) may be performed very quickly by bit shift right by *k* positions.
  In C: operator >>.
- Similarly, we can multiply by 2^k, with bit shift left (<<) by *k* positions (zeros enter from the right).
- It should be used in the code, e.g. “x >> 1” instead of “x / 2”.

More about multiplication

We have a result of multiplication of two Q15 numbers as a Q30 value. How can we get a Q15 number back?

- First, divide by 2^{15}, shifting right by 15 b.
- Next, discard higher bits, leaving only lower 16 b.
- This way, rounding down is performed.
- To round to the nearest number, before shifting, add 2^{14} (1 on the highest bit that will be removed), then shift and truncate the result.

193267630 (Q30) → (193267630 + (1<<14)) → 193284014 →

→ (>> 15) → 5898 (Q15) → 5898 /32768 → 0.17999267

Underflow

- What happens if, after bit shifting, we get only 0s?
- (0.003 × 0.002) → 98 × 66 = 6468 (Q30)
- 6468 >> 15 = 0 (Q15) !!!
- The result (0.003 × 0,002) = 0.00006 is too small.
- Underflow occurs, resulting in the result equal to zero.
- All the following multiplications (e.g. in a filter) will also give the result of zero!
- We try to prevent the underflow by using longer types and by reorganizing the order of calculations.

Q15 multiplication in C

How do we write a Q15 multiplication in C?

```
short a = 9830;
short b = 19661;
/* short y = ??? */
```

Not this way – the result won’t fit in *short* type (16 bits), higher bits will be lost:

```
short y = a * b;  // 1966
```

Maybe this way? Type *long* has 32 bits, so the result should fit. Unfortunately, nothing has changed. Why?

```
long y = a * b;  // 1966
```

Q15 multiplication in C

How does the C compiler interpret this code?

```
long y = a * b;  // a, b: type short
```

- First, it computes the right-side expression: (a * b).
- The result type is equal to the “largest” type of arguments. Both arguments are *short*, so the result is also *short*.
- The highest bits are truncated (they don’t fit in *short*).
- The (incorrect) result is written to the variable on the left side (type *long*).

Type casting in C

- To obtain the correct result, we must “promote” an argument to a longer type.
- Type cast in C requires specifying the new type in parentheses before an expression or a variable.

```
long y = (long)a * b;  // 193267630
```

- Now, one of the arguments is *long*, so the result will also be *long.*
- We can also cast a numeric constant by adding “L” (for *long*) after the value:

```
long y = a * 19661L;
```

Q15 multiplication in C

To multiply (0,3 · 0,6) and write the result into *short* variable in Q15, we must do this very long instruction:

```
short y = (short)(((long)a * b) >> 15);
```

or, with rounding, even longer:

```
short y = (short)((((long)a * b) + (1<<14)) >> 15);
```

Fortunately, on DSPs we can often use “shortcut” instructions.

For example, on C5535 DSP we can achieve the same result with:

```
short y = _smpy(a, b);
```

_smpy – multiply with saturation

Saturation mode

- If an overflow occurs, the result may be very wrong, e.g.:
  32760 + 10 → -32766.
- DSPs can use saturation arithmetic. Saturation works by clipping the values to the range:
  32760 + 10 → 32767
- The result is still wrong, but the error is smaller.
- On C5535 DSP, we can use instructions working in the saturation mode. Their names start with _s:
  _sadd (+), _ssub (–), _smpy (×), _sround (rounding), etc.

Floating point numbers

- Floating point notation increases the accuracy of number representation significantly, compared with fixed point.
- The processor must have a special unit for floating point processing – FPU (*floating point unit*). Such a processor is a floating-point processor.
- C5535 DSP used in the course project, as well as many other DSPs, does not have a FPU – it is a fixed-point processor.
- Fixed-point DSPs are still used, they are not “obsolete”.

Floating point numbers

Each number is represented with:

- S – sign (0 or 1, positive or negative),
- M – mantissa,
- E – exponent,
- b – base (usually b = 2).

$$(-1)^{s} \cdot 1.M \cdot b^{(E-127)}$$

Example:

$3.14159265359 = 1.570796326795 \cdot 2^{1}$

S = 0, M = 570796326795, E = 128, b = 2

Floating point types

Floating point types defined in IEEE 754 standard:

- *float* – single precision
  - 32 bits: 1 b sign, 23 b mantissa, 8 b exponent
  - 7 significant digits after the decimal point
  - range: $\pm 3.4 \cdot 10^{-38}$ to $\pm 3.4 \cdot 10^{38}$
- *double* – double precision
  - 64 bits: 1 b sign, 52 b mantissa, 11 b exponent
  - 15 significant digits after the decimal point
  - range: $\pm 1.7 \cdot 10^{-308}$ to $\pm 1.7 \cdot 10^{308}$

Resolution is variable, it depends on the value.

Floating point types

According to IEEE 754, float and double variables may have the following special variables:

- *Inf* – positive infinity, e.g. (1.0 / 0.0)
- *-Inf* – negative infinity, e.g. (-1.0 / 0.0)
- *NaN* – undefined value (not a number), e.g. (0.0 / 0.0)
- -0.0 (negative zero) – should be treated as the normal zero.

Features of the floating-point notation

- The risk of overflow is very low, because of very wide range.
- The risk of underflow is low (lower for *double*).
- No special notation is needed in C:

  ```
  double y = 0.3 * 0.6;
  ```

  We get 0.18, not 0.1799945 as for the fixed-point notation.
- Floating-point operations require significantly more processor cycles and more memory.

Comparing floating-point numbers

One caveat: remember that floating-point numbers have finite precision.

This probably won’t work:

```
if (a / 2 == 2.5) {   // double a
```

The result of division may be e.g. 2.5000000000001 instead of 2.5.

We must check whether the difference is below the limit, e.g. lower than 10^{-8}:

```
if (abs(a / 2 - 2.5) < 1e-8) {
```

Casting floating-point values in C

Remember: a C compiler computes the right-hand side first.

```
short a = 5;
double b = a / 2;
```

The result will be *b* = 2, because integer division (*a* / 2) will be performed first, only then the result will be written to a *double* variable.

The correct casting (result 2.5):

```
short a = 5;
double b = (double)a / 2;
double c = a / 2.0;
float d = (float)a / 2;
float e = a / 2.0f;
```

Advantages of floating-point processors

Floating-point processors compared with fixed-point ones

- Higher accuracy of number representation.
- Larger dynamic range – low quantization noise.
- Easier to code – no special conversion to Q format needed.
- Easier and more accurate mathematic operations (square root, logarithm, trigonometric functions, etc.).

For example, in digital IIR filters, quantization noise is much lower, risk of filter instability is reduced, the results are more accurate.

Disadvantages of floating-point processors

Floating-point processors compared with fixed-point ones

- Longer computations, more cycles used for performing operations (even a simple multiplication).
- More memory used, especially with the *double* type.
- More energy consumption (= higher cost of usage).
- Much higher cost of a processor.

Practical recommendations

When should we use a floating-point DSP?

- when we need high precision of computations,
- when large dynamic range is needed, e.g. processing sound samples with 16-bit or 32-bit resolution – lower noise,
- when we can afford a fixed-point DSP.

When should we use a fixed-point DSP?

- when low cost of a DSP and its usage is important,
- when only simple DSP algorithms (FFT, filters) are used,
- when the processed signal has low dynamic range (e.g. a signal from A/D converter with 12-bit resolution).