Using Fixed-Point Numbers

Prof. Dr. Martin J. W. Schubert

Electronics Laboratory

Regensburg University of Applied Sciences

Regensburg

Abstract. This tutorial is intended to detail the use of integer and fixed point numbers when processing data samples with micro controllers or FPGAs.

1 Introduction

Using integers as fixed point numbers is an essential skill for micro controller and FPGA programming, particularly when digital signal processing (DSP) and A/D - D/A conversion are taken into account.

The organization of this document is as follows:

Chapter 1	introduction,
Chapter 2	introduces different number representations and conversion algorithms between them,
Chapter 3	discusses rounding techniques,
Chapter 4	offers an exercise,
Chapter 5	summarizes the tutorial,
Chapter 6	gives some references and
Chapter 7	the solutions to the exercises.

2 Number Representations (See chapter 6.2 for solutions.)

2.1 Integral Numbers

There are two ways to interpret a bit vector as integral number: *unsigned* and *signed*, corresponding to the *IEEE* VHDL libraries *std_logic_unsigned* and *std_logic_signed*, resp.

- **Unsigned interpretation: A bit vector of w bits represents the integer range $0 \dots 2^w - 1$.**
- **Signed interpretation: A bit vector of w bits represents the int. range $-2^{w-1} \dots +2^{w-1} - 1$.**

2.2 Fixed Point Numerical Representation: The Q Number Format

Unsigned: UQg.f with *g* integral (deutsch: ganze) and *f* fractional bits. Width *w=g+f.*
Signed: Qg.f with 1 sign bit plus *g* integral and *f* fraction bits. Width *w=1+g+f.*

Example: 101.1001 can be interpreted as UQ3.4 format representing $1011001 * 2^{-4} = 89/16 = 5.5625$ or as Q2.4 delivering $-(0100110+1) * 2^{-4} = -(0100111) * 2^{-4} = -39/16 = -2.4375$.

Exercise: The bit string **110.1011** can be interpreted...

... as UQ3.4 format representing

. . . .

... as Q2.4 delivering

. . . .

Unsigned: Range: $0 \le A_U \le \frac{2^{g+f}-1}{2^f}$, Resolution: $r = 2^{-f} = \frac{1}{2^f}$.

Singed Range: $-\frac{2^{g+f}}{2^f} \le A_S \le \frac{2^{g+f}-1}{2^f}$, Resolution: $r = 2^{-f} = \frac{1}{2^f}$.

- ➢ You can append an arbitrary number of zeros after the point.
- ➢ You can precede an arbitrary number of zeros before an unsigned number.
- ➢ You can precede an arbitrary multiple of the sign bit before a signed number.

Summation an subtraction of fixed-point numbers is easy as they can be treated like integer numbers when they are written such that the points are over each other. Example:

Given numbers	Unsigned treatment	Signed treatment
11011011.11011	11011011.11011**000**	11011011.11011**000**
± 101.11101101	± **00000**101.11101101	± **11111**101.11101101

Table 2.2: Q-formats (as typical for micro controllers), *w*: total number of bits, *r*: resolution

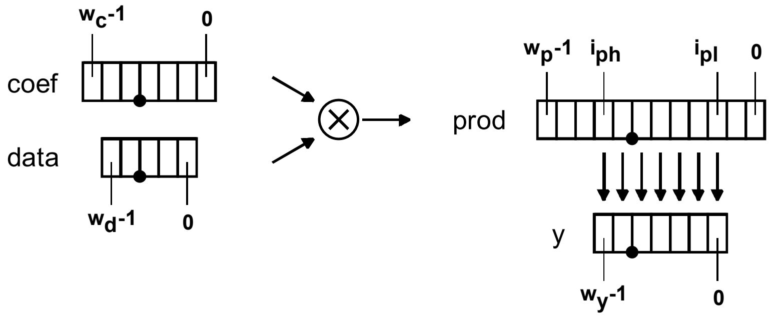
Format	*w*	*g*	*f*	**min**	**max**	***r* (resolution)**
UQ16	16	16	0	0	2^16 - r	1
UQ.16	16	0	16	0	1 - r	2^-16
Q15	16	15	0	-2^15	2^15 - r	1
Q.15	16	0	15	-1	1 – r	2^-15
UQ16.16	32	16	16	0	2^16 - r	2^-16
Q15.16	32	15	16	-2^15	2^15 - r	2^-16

Caution: Sometimes you will find the so-called Qf-Format with Q15 meaning Qg.15, g=?. Then we know about 1 sign bit and 15 fractional bits but an unknown number of integral bits. This causes uncertainty! Avoid it, even in a C program with all *integers* having 32 bits (because there also exist *short int* (16 bits) and *char* (8 bits) types in C).

➢ You cannot mark the Q-format within the bit string. It's a predefined arrangement of your design.

2.3 Multiplication of Fixed-Point Numbers

Fig. 2.3:
Reducing the product length to the length of its factors with indices `iph` and `ipl`.



We compute **`prod = coef`** * **`data`** with **`coef`** and **`data`** having w_c and w_d binary places, respectively, f_c and f_d of them fractional. Then **`prod`** has $w_p=w_c+w_d$ binary places, $f_p=f_c+f_d$ of them fractional.

Mathematical proof: We can write $\mathtt{coef = icoef \cdot 2^{-fc}}$ and $\mathtt{data = idata \cdot 2^{-fd}}$ with **i**xxx integral. Consequently, the product can be written as
$\mathtt{prod = coef \cdot data = icoef \cdot 2^{-fc} \cdot idata \cdot 2^{-fd} = icoef \cdot idata \cdot 2^{-(fc+fd)}}$.

Reducing the length of products:

Proof: We want to reduce the width of **prod** by taking result vector **y** out of it. Result **y** has w_y bits in formatted as [U]Qg_y.f_y.

Considering fractional bits only:
The fractional part of product **prod** consists of bits f_p-1...0.
The fractional part of result **y** will consist of bits f_y-1...0.

Preserving the point we get $\mathbf{y}(\mathbf{f}_y\text{-}1 : 0) = \mathbf{p}(\mathbf{f}_p\text{-}1 : \mathbf{f}_p\text{-}\mathbf{f}_y)$ with lowest index $\mathbf{i}_{pl} = \mathbf{f}_p\text{-}\mathbf{f}_y$.

Considering integral bits also:
As $\mathbf{y} = \mathbf{y}(\mathbf{w}_y\text{-}1 : 0)$ its max. index is $\mathbf{w}_y\text{–}1$ larger than its min. index: $\mathbf{i}_{ph} = \mathbf{i}_{pl} + (\mathbf{w}_y\text{-}1)$.

Consequently (formula to be used in exercise chapter 4):

$$\boxed{\mathbf{y} = \mathbf{prod}(\mathbf{i}_{ph} : \mathbf{i}_{pl}) \quad \textbf{with} \quad \mathbf{i}_{pl} = \mathbf{f}_p\text{-}\mathbf{f}_y, \quad \mathbf{i}_{ph} = \mathbf{i}_{pl} + \mathbf{w}_y\text{-}\mathbf{1}}$$

Exercises (for solutions see → chapter 6) **:**

Let `coef` have $\mathbf{w}_c$ binary places, $\mathbf{f}_c$ of them fractional. Signal `data` has $\mathbf{w}_d$ binary places, $\mathbf{f}_d$ of them fractional. The product has

$\mathbf{w}_p$ = ... binary places, $\mathbf{f}_p$ = ... of them fractional.

Fig. 2.3 illustrates the multiplication of the coefficient `coef` with $\mathbf{w}_c$ = ... , $\mathbf{f}_c$ = ...

and the data sample `data` with $\mathbf{w}_d$ = ... , $\mathbf{f}_d$ = ... The product `prod` has

$\mathbf{w}_p$ = ... binary places,

$\mathbf{f}_p$ = ... of them fractional.

We want to take result vector `y` out of `prod` preserving the point. For all bit vectors the LSB has index 0.

In Fig. 2.3 `y` has $\mathbf{w}_y$ = ... , binary places $\mathbf{f}_y$ = ... of them fractional.

To apply the VHDL command `y<=prod(iph DOWNTO ipl)` we have to compute

$\mathbf{i}_{pl}$ = ...

$\mathbf{i}_{ph}$ = ...

2.4 Binary → Hexadecimal → Binary Conversion

Table 2.4: Mapping decimal, hexadecimal and binary numbers

Decimal number	Hexadecimal Digit	Bit vector		Decimal number	Hexadecimal Digit	Bit vector
0	0	0000		8	8	1000
1	1	0001		9	9	1001
2	2	0010		10	A	1010
3	3	0011		11	B	1011
4	4	0100		12	C	1100
5	5	0101		13	D	1101
6	6	0110		14	E	1110
7	7	0111		15	F	1111

Hexadecimal numbers are easier to read and remember than bit vectors. Starting from the point bits are subdivided into packages of 4 bits and replaced by equivalent hex-digits.

Example: $10100101101.011010110101_2 = 101\ \ 0010\ \ 1101\ .\ 0110\ \ 1011\ \ 0101_2 = 52D.6B5_{16}$.

Convert the hex-number back to a bit vector translating every hex-digit to a 4-bit string.

Example: = $52D.6B5_{16}$ => $101\ \ 0010\ \ 1101\ .\ 0110\ \ 1011\ \ 0101_2$.

Exercise: convert to binary:

ABC.DEF$_{16}$ **=** .

. . . .

Exercise: convert to hex:

1111 1110 1101.1100 1011 1010$_2$ **=**

. . . .

2.5 Decimal → Hexadecimal → Decimal Conversion

Decide for the number of fractional hex-digits, $\mathbf{f_h}$, and multiply the decimal number with 16^{fh}. If desired the decimal number can then be rounded or truncated. The resulting integral number is then converted to a hex-number.

Example: We want to have $\mathbf{f_h}$=3 hexadecimal fractional digits.

1234.567_{10} $= 1234.567_{10} * (16^3 * 16^{-3}) = 1234.567_{10} * 16^3 * 16^{-3} = 5\,056\,786.432_{10} * 16^{-3}$

$\approx\ 5\,056\,786_{10} * 16^{-3} = 4D2912_{16} * 16^{-3} = 4D2.912_{16}$

Easier to compute might be the form separating integral and fractional parts:

1234.567_{10} $=1234_{10} + 0.567_{10} = 4D2_{16} + 0.567_{10} * 16^3 * 16^{-3} = 4D2_{16} + 2322.432_{10} * 16^{-3}$

$\approx\ 4D2_{16} + 2322_{10} * 16^{-3} = 4D2_{16} + 912_{16} * 16^{-3} = 4D2.912_{16}$

Remember: $1234_{10} = (77*16) + 2 = ((4*16) + 13)*16 + 4 = 4*16^2 + 13*16^1 + 2*16^0 = 4D2_{16}$.

Back translation to decimal is performed by multiplying hex-digit on position m with 16^m.

Example: $\mathbf{4D2.912}_{16} = \mathbf{4}*16^2 + \mathbf{13}*16^1 + \mathbf{2}*16^0 + \mathbf{9}*16^{-1} + \mathbf{1}*16^{-2} + \mathbf{2}*16^{-3} \approx 1234.56689$.

Exercise: convert to decimal (f=3) **:** **`ABC.DEF`**$_{16}$ **=** . . . .

Exercise: convert to hex (f=3)**:** **`2748.871`**$_{16}$ **=** . . . .

. . . .

2.6 Real → Binary Conversion

Factors – like filter coefficients – are computed as real numbers and have to be converted to bit vectors. Let's assume the number rVal=1.234 has to be converted to a bit string with 8 binary places, 6 of them fractional. The example below shows a possible way to accomplish this.

$\text{rVal} = \text{rVal} \cdot (1) = \text{rVal} \cdot (2^6 \cdot 2^{-6}) = (\text{rVal} \cdot 2^6) \cdot 2^{-6} = (1.234 \cdot 64) \cdot 2^{-6} = 78.976 \cdot 2^{-6}$
$\text{iVal} = \text{round}(\text{rVal}\cdot 2^6) \cdot 2^{-6} = \text{round}(78.976) \cdot 2^{-6} = 79 \cdot 2^{-6} = 01001111_2 \cdot 2^{-6} = 01.001111_2$.

For the negative rVal2 = -rVal = -1.234 we obtain in the same way rVal2 = -78.976 · 2-6 and $\text{iVal2} = \text{round}(\text{rVal2}\cdot 2^6) \cdot 2^{-6} = \text{round}(-78.976) \cdot 2^{-6} = -79 \cdot 2^{-6} = 10110001_2 \cdot 2^{-6} = 10.110001_2$.

Positive an negative numbers are distinguished by the first bit. Be careful to not set this bit accidentally by a too large positive number. The largest positive number for a signed 8-Bit representation is $\text{iVal}_{max}=2^7–1=127$ and the largest negative number is $\text{iVal}_{min}=–2^7=–128$.

Exercises (for solutions see → chapter 8) **:**
Convert π=3.14159 into a signed bit vector with 8 binary places, 4 of them fractional.

. . . .

. . . .

Convert -π=-3.14159 into a signed bit vector with 8 binary places, 4 of them fractional.

. . . .

. . . .

2.7 Floating-Point Numbers

Fig. 2.7: Floating-point data structure

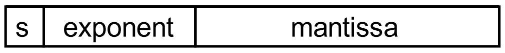


Table 2.7: IEEE 754 binary formats [1]

Format	**Sign**	**Exponent**	**Mantissa**	**Total number of bits**	**Exponent bias**
Half	1	5	10	16	15
Single	1	8	23	32	127
Double	1	11	52	64	1023
Quad	1	15	112	128	16383

The floating point data structure has 1 sign bit *s*, *e* exponent bits and *m* mantissa bits. The number is computed from

$$\texttt{real\_value} = (-1)^{s} \times 2^{\texttt{exponent} - \texttt{exponent\_bias}} \times \texttt{mantissa}$$

The exponent is biased by $(2^{e-1})-1$ to obtain both positive and negative exponents.

If possible, the mantissa is stored normalized i.e. with one bit before the point. Example: the number `101.1101` is stored as $\texttt{1.011101} \times 2^{+2}$.

The number is said to be de-normalized if the MSB of the mantissa is 0 and its fraction ≠0.

Particular situations

- ±0 (depending on the sign bit) : exponent = 0 and mantissa = 0.
- ±∞ (depending on the sign bit) : exponent = 2^e-1 (=all ones) and mantissa fraction =0
- NaN (Not a Number) : exponent = 2^e-1 (=all ones) and mantissa fraction ≠0

Floating point numbers are well suited for multiplication and division, as $2^A \times 2^B = 2^{A+B}$, but not for addition and subtraction, as for this operations it has be brought into a fixed-point like format. Typically, working with floating-point numbers is significantly more time consuming than working with fixed-point numbers. However, the range of floating-point numbers is significantly larger than that of fixed-point numbers.

3 Rounding and Truncation

Truncation

Truncating a number with integral part *g* and fractional part *f* (i.e. *f*<1):
g.f truncates to *g* (, regardless whether *g* is positive or negative):
Example:5.8 truncates to 5, -5.8 truncates to –5.

Rounding Threshold

The threshold for rounding is ½·LSB with LSB being the least significant bit. For integral numbers LSB=1. With Base (or radix) B = 10, 2, 16 we get ½ B = 5, 1, 8, respectively. Consequently the numerical thresholds are $5\cdot10^{-1}=0.5_{10} = 1\cdot2^{-1}=0.1_2 = 8\cdot16^{-1}=0.8_{16}$.

Rounding:

This method corresponds to the C or Matlab expression *round(g.f)* for decimal numbers.
Positive numbers: *g.f* rounds to *g* when $f < 0.5$ and to *g+1* when $f \geq 0.5$.
Negative numbers: *g.f* rounds to *g* when $f < 0.5$ and to *g-1* when $f \geq 0.5$.

Possible realization:
+ For numbers ≥ 0 : *rounded_number* = $g + f_1$, with f_1 being the first fractional bit.
- For numbers < 0 : *rounded_number* = $-(g' + f_1')$ with $g'.f' = -(g.f)$.

Bit-Vector Easy Rounding Scheme:

This method corresponds to the C or Matlab expression *floor(g.f+0.5)* for decimal numbers.

Easy realization: *bver_rounded_number* = $g + f_1$ with f_1 being the first fractional bit.

Exercise:

Fill the empty fields in Table 3-1 to understand the differences between truncation, mathematical rounding and the bit-vector easy rounding presented above. The bit-strings are assumed ot be 5-bit signed numbers.

Table 3-1: Truncation, rounding and bit-vector easy rounding: (complete empty fields):

binary	binary	decimal	decimal	truncated		rounded		$+0.1_2$ truncated	
	rational	rational		bin	=dec	bin	=dec	bin	=dec
01.001	0 1001 / 2^3	09 / 8	+1.125	01	+1	01	+1	01	+1
01.011	0 1011 / 2^3	11 / 8	+1.375						
01.100	0 1100 / 2^3	12 / 8	+1.500						
01.101	0 1101 / 2^3	13 / 8	+1.625						
01.111	0 1111 / 2^3	15 / 8	+1.875						
10.111	1 0111 / 2^3	-09 / 8	-1.125	10	+2	11	-1	11	-1
10.101	1 0101 / 2^3	-11 / 8	-1.375						
10.100	1 0100 / 2^3	-12 / 8	-1.500						
10.011	1 0011 / 2^3	-13 / 8	-1.6250						
10.001	1 0001 / 2^3	-15 / 8	-1.8750						

Check with table 3-2 when rounding and bit-vector easy rounding obtains same or different results:

Table 3-2: Truncation, rounding and bit-vector easy rounding: (complete empty fields):

binary	**decimal**	**decimal**	**rounded**		**+0.1$_2$ truncated**		**iden-**
	rational	fixed point	**bin**	=dec	**bin**	=dec	**tical**
001.01111111	+383 / 2^8	1.49609375	001	+1	001	+1	yes
001.10000000	+384 / 2^8	1.5					
001.10000001	+385 / 2^8	1.50390625					
110.10000001	-384 / 2^8	-1.49609375	111	-1	111	-1	yes
110.10000000	-384 / 2^8	-1.5					
110.01111111	+385 / 2^8	-1.50390625					

What is correct?: The difference between rounding and bit-vector easy rounding **increases / decreases** with the number of fractional bits.

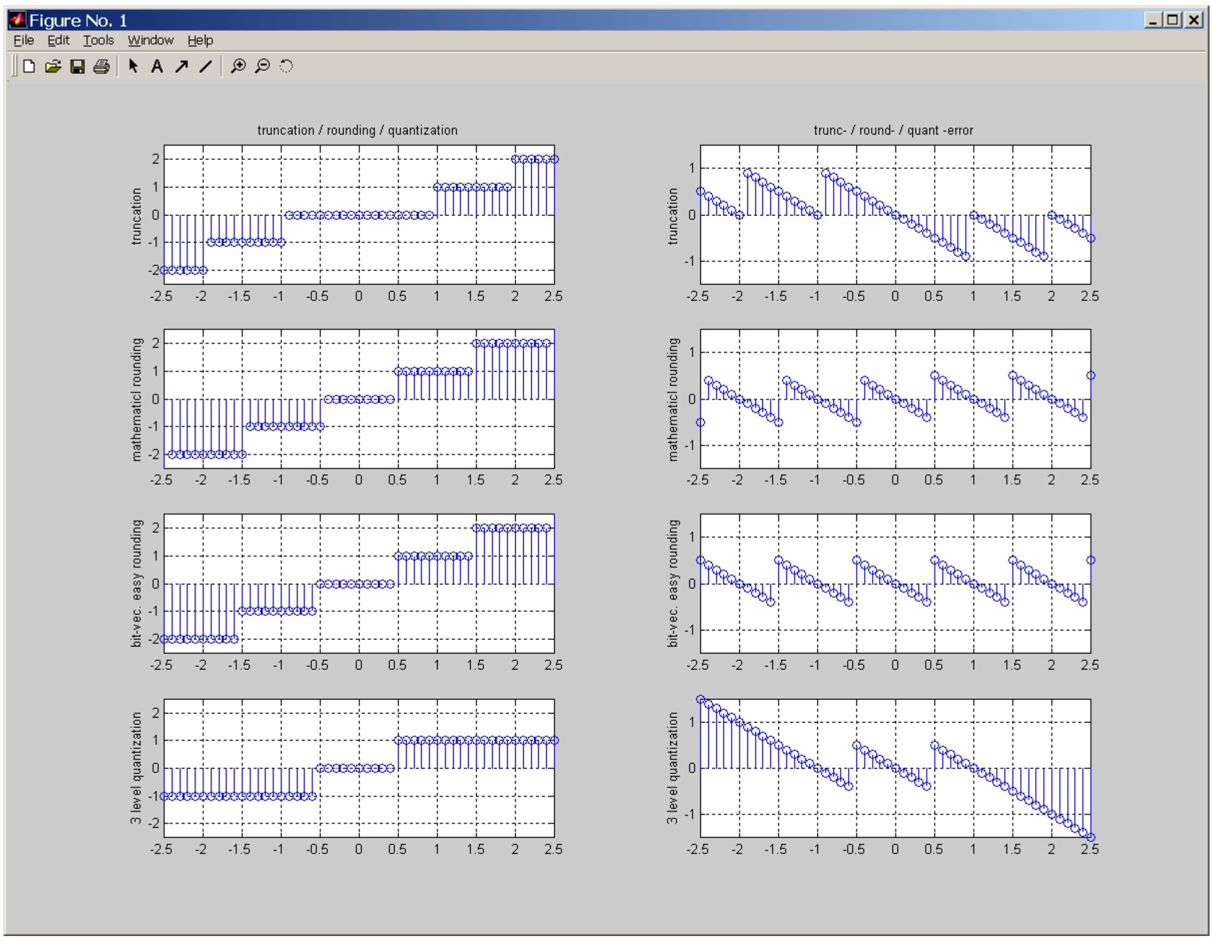


Fig. 3: Matlab plot. Top down: truncation, rounding, bit-vector easy rounding, 3-level quantization. Differences between the second and third line are in -*n*.5 only.

4 Exercise Based on Executable VHDL

Listing 4: Code with gaps

```
(1)  LIBRARY ieee; USE ieee.std_logic_1164.ALL;
(2)  PACKAGE pk_filter IS
(3)    CONSTANT cDataInWidth:POSITIVE:=4;    -- Input-Data BitWidth
(4)    CONSTANT cDataInFract:POSITIVE:=2;    -- No of Input-Data fract. Bits
(5)    CONSTANT cDataOutWidth:POSITIVE:=5; -- Output-Data BitWidth
(6)    CONSTANT cDataOutFract:POSITIVE:=3; -- No of Output-Data fract Bits
(7)    CONSTANT cCoefWidth:POSITIVE:=4;     -- Coefficient's BitWidth
(8)    CONSTANT cCoefFract:POSITIVE:=2;     -- No of Coef's fractional Bits
(9)    SUBTYPE  t_DataIn  IS std_logic_vector(cDataInWidth-1  DOWNTO 0);
(10)   SUBTYPE  t_DataOut IS std_logic_vector(cDataOutWidth-1 DOWNTO 0);
(11)   SUBTYPE  t_coef IS std_logic_vector(cCoefWidth-1 DOWNTO 0);
(12) END PACKAGE pk_filter;
(13)
(14) LIBRARY ieee; USE ieee.std_logic_1164.ALL,
(15)                  ieee.std_logic_signed."+", ieee.std_logic_signed."*";
(16) USE WORK.pk_filter.ALL;
(17) ENTITY TestBitslice IS
(18) END ENTITY TestBitslice;
(19)
(20) ARCHITECTURE rtl_TestBitslice OF TestBitslice IS
(21)   SIGNAL DataIn :t_DataIn;
(22)   SIGNAL coef   :t_coef;
(23)   SIGNAL DataOut:t_DataOut;


(24)   SIGNAL product:std_logic_vector(...................................

     ......................................................................


(25)   CONSTANT iPl:NATURAL:= ............................................

     ......................................................................


(26)   CONSTANT iPh:NATURAL:= ............................................

     ......................................................................
(27) BEGIN
(28)   DataIn  <= "0101", "0100" AFTER 10 ns; -- 1.25,   1.00 AFTER 10 ns
(29)   coef    <= "0101";                      -- 1.25
(30)   product <= coef * DataIn;               -- 1.5625, 1.25 AFTER 10 ns


(31)   DataOut <= product(iPh DOWNTO iPl) ................................


     ......................................................................
(32) END ARCHITECTURE rtl_TestBitslice;
```

Correspondences with chapter 2.3: f_c=cCoefFract, f_d=cDataInFract, f_y=cDataOutFract, w_c, w_d, w_p, w_y: cCoefWidth, cDataInWidth, cProdWidht, cDataOutWidth, respectively.

Exercises:

- Complete line (24) to get a *product* signal that fits to the multiplication of line (30).
- Compute *iPl* und *iPh* in lines (25), (26) to fit the bit-slice operation of line (31).
- Extend line (31) to get the bit-slice by bit-vector easy rounding.
- Verify the product, bit-slice and rounding operation of lines (39), (49) by hand.

5 Summary

Binary, decimal and hexadecimal coding were presented as well as conversion techniques between them, particularly when these number representations appearing fixed-point formats. After a short glance on floating-point numbers rounding was considered and an easy way to round bit vectors was presented. The tutorial finished with an example based on VHDL.

6 References

[1] IEEE standard 754, available: http://www.ieee.org/publications_standards/publications/subscriptions/prod/standards_overview.html.

[2] Available: http://de.wikipedia.org -> fixed-point

7 Appendix: Solutions to the Exercises

7.1 Introduction

7.2 Number Representations

7.2.1 Integral Numbers

7.2.2 Fixed Point Numerical Representation: The Q Number Format

Exercise: The bit string **110.1011** can be interpreted...
... as UQ3.4 format representing
`1101011*2`$^{-4}$ `= 107/16 = 6.6875`
... as Q2.4 delivering
`-(0010100+1)*2`$^{-4}$ `= -(0010101)*2`$^{-4}$ `= -21/16 = -1.3125.`

7.2.3 Multiplication of Fixed-Point Numbers

Exercises (for solutions see → chapter 8) **:**
Let `coef` have w_c binary places, f_c of them fractional. Signal `data` has w_d binary places, f_d of them fractional. The product has
w_p = `....`$w_c + w_d$`.....` binary places, f_p = `....`$f_c + f_d$`.....`of them fractional.
Fig. 2.3 illustrates the multiplication of the coefficient `coef` with w_c = `..7...`, f_c = `..4..`
and the data sample `data` with w_d = `...5...`, f_d = `...3....` The product `prod` has
w_p = `...` $w_c + w_d$`.= 7 + 5 = 12 .................` binary places,
f_p = `..` $f_c + f_d$ `= 4 + 3 = 7 ...................` of them fractional.
We want to take `y` out of `prod` preserving the point. For all bit vectors the LSB has index 0.
In Fig. 2.3 `y` has w_y = `...7...`, binary places f_y = `...5...` of them fractional.
To apply the VHDL command `y<=prod(iph DOWNTO ipl)` we have to compute
i_{pl} = `...` $f_p - w_y$ `= 7 - 5 = 2 ................................`
i_{ph} = `...` $i_{pl} + w_y$ `-1 = 2 + 7 - 1 = 8 ........................`

7.2.4 Binary to Hexadecimal to Binary Conversion

Exercise: convert to binary:
$\texttt{ABC.DEF}_{16}$ = $\texttt{1010 1011 1100 . 1101 1110 1111}_2$.
Exercise: convert to hex:

$1111\ 1110\ 1101.1100\ 1011\ 1010_2$ = FED.CBA$_{16}$

7.2.5 Decimal to Hexadecimal to Decimal Conversion

Exercise: convert to decimal (f=3) **: ABC.DEF$_{16}$ =** ......2748.8708496......
Exercise: convert to hex (f=3)**: 2748.871$_{16}$ =** ...ABC + $0.871\cdot16^{-3}$ = ...
= ABC + 3566.79 ≈ = ABC + $3567\cdot16^{-3}$ = ABC + DEF$\cdot16^{-3}$ = ABC.DEF

7.2.6 Real-to-Binary Conversion

Exercises (for solutions see → chapter 8) **:**
Convert π=3.14159 into a signed bit vector with 8 binary places, 4 of them fractional.
$3.14159 \cdot (2^4 \cdot 2^{-4}) = (3.14159\cdot16) \cdot 2^{-4} = 50.26... \cdot 2^{-4}$ => $50 \cdot 2^{-4}$
$50_{10} \cdot 2^{-4} = 00110010_2 \cdot 2^{-4} = 0011.0010_2$

Convert -π=-3.14159 into a signed bit vector with 8 binary places, 4 of them fractional.
$-3.14159 \cdot (2^4 \cdot 2^{-4}) = (-3.14159\cdot16) \cdot 2^{-4} = -50.26... \cdot 2^{-4}$ => $-50 \cdot 2^{-4}$
$(-50_{10}) \cdot 2^{-4} = ((\sim0011.0010_2)+1) \cdot 2^{-4} = 11001110_2 \cdot 2^{-4} = 1100.1110_2$

7.3 Rounding and Truncation

Table 3-1: Truncation, rounding and bit-vector easy rounding: (complete empty fields):

binary	**bin rat.**	**dec. rat.**	**decimal**	**truncated**		**rounded**		**+0.1$_2$ truncated**	
				bin	=dec	**bin**	=dec	**bin**	=dec
01.001	0 1001 / 2^3	09 / 8	+1.125	01	+1	01	+1	01	+1
01.011	0 1011 / 2^3	11 / 8	+1.375	01	+1	01	+1	01	+1
01.100	0 1100 / 2^3	12 / 8	+1.500	01	+1	10	+2	10	+2
01.101	0 1101 / 2^3	13 / 8	+1.625	01	+1	10	+2	10	+2
01.111	0 1111 / 2^3	15 / 8	+1.875	01	+1	10	+2	10	+2
10.111	1 0111 / 2^3	-09 / 8	-1.125	10	+2	11	-1	11	-1
10.101	1 0101 / 2^3	-11 / 8	-1.375	10	+2	01	-1	11	-1
10.100	1 0100 / 2^3	-12 / 8	-1.500	10	-2	10	-2	11	-1
10.011	1 0011 / 2^3	-13 / 8	-1.6250	10	-2	-2	-2	10	-2
10.001	1 0001 / 2^3	-15 / 8	-1.8750	10	-2	-2	-2	10	-2

Table 3-2: Truncation, rounding and bit-vector easy rounding: (complete empty fields):

binary	**decimal**	**decimal**	**rounded**		**+0.1$_2$ truncated**		**iden-**
	rational	fixed point	**bin**	=dec	**bin**	=dec	**tical**
001.01111111	+383 / 2^8	1.49609375	001	+1	001	+1	yes
001.10000000	+384 / 2^8	1.5	010	+2	010	+2	yes
001.10000001	+385 / 2^8	1.50390625	010	+2	010	+2	yes
110.10000001	-384 / 2^8	-1.49609375	111	-1	111	-1	yes
110.10000000	-384 / 2^8	-1.5	110	-2	111	-1	no
110.01111111	+385 / 2^8	-1.50390625	110	-2	110	-2	yes

Correct: The difference between rounding and bit-vector easy rounding **decreases** with the number of fractional bits.

7.4 Exercise Based on Executable VHDL

Solutions:

```
(24)   SIGNAL product:std_logic_vector(cDataInWidth+cCoefWidth-1 DOWNTO 0);
(25)   CONSTANT iPl:NATURAL:=cCoefFract+cDataInFract-cDataOutFract;
(26)   CONSTANT iPh:NATURAL:=iPl+cDataOutWidth-1;
(31)   DataOut <= product(iPh DOWNTO iPl) + product(iPl-1);
```

Verification of product and rounding by hand:

Factors:

```
DataIn  =   "01.01"   ,  "01.00"    AFTER 10 ns; -- = 1.5625 → 1.25
coef    =   "01.01";                             -- = 1.5
```

No rounding:

```
product = "0001.1001", "0001.0100" AFTER 10 ns; -- = 1.5625 → 1.25
DataOut =   "01.100" , "  01.010"  AFTER 10 ns; -- = 1.5    → 1.25
```

With bit-vector easy rounding:

```
product = "0001.1001", "0001.0100" AFTER 10 ns; -- = 1.5625 → 1.25
DataOut =   "01.101" , "  01.010"  AFTER 10 ns; -- = 1.625  → 1.25
```