APPENDIX **B**

FIXED-POINT ARITHMETIC AND HDL CODING

This appendix presents an overview of the representation of numbers in a finite precision arithmetic environment. The discussion then focuses on the binary two's complement representations and coding of fixed-point arithmetic in hardware description languages (HDLs)—VHDL and Verilog-HDL.

B.1 ROUNDING OPERATION AND ROUND-OFF ERROR

In a finite precision computing system, a limited number of digits are available to represent any given signal. *Representable numbers*, that is, those numbers having an exact representation in the considered arithmetic system, necessarily form a discrete subset $\tilde{\mathcal{I}}$ of a continuous subset $\mathcal{I}$ of the real axis.

It is first necessary to clarify how a given $c \in \mathcal{I}$ can be approximated by a suitable element $\tilde{c} \in \tilde{\mathcal{I}}$. To this end, recall that for a given integer b strictly greater than one, every nonzero real quantity c can be *uniquely* written as

$$\boxed{c = \pm \mathcal{S}_b(c) \times b^{\mathcal{E}_b(c)}}, \tag{B.1}$$

where

- b is called the *base* or *radix* of the representation.
- $\mathcal{E}_b(c)$ is an integer called the *exponent* of c, which defines its *order of magnitude*.
- $\mathcal{S}_b(c)$ is a real number such that $1 \leq \mathcal{S}_b(c) < b$. It is called the *significand* of c.

For instance,

$$\begin{aligned} \frac{3}{8} &= 3.75 \times 10^{-1} && \text{(Radix-10 representation } (b = 10)), \\ &= 3 \times 8^{-1} && \text{(Radix-8 representation } (b = 8)), \\ &= 1.5 \times 2^{-2} && \text{(Radix-2 representation } (b = 2)). \end{aligned} \tag{B.2}$$

Digital Control of High-Frequency Switched-Mode Power Converters, First Edition.
Luca Corradini, Dragan Maksimović, Paolo Mattavelli, and Regan Zane.

Representation (B.1), known as *exponential notation*, suggests that, in order to define $\tilde{c}$, one can truncate or round-off $\mathcal{S}_b(c)$ to a given number of digits. Consider, for instance, number $\pi = 3.1415926535\ldots$. In a radix-10 notation, $\mathcal{E}_{10}(\pi) = 0$ and $\mathcal{S}_{10}(\pi) = \pi$. Consider then successive approximations of π in which the significand is *rounded* to its first n decimal digits,

$$\begin{aligned}
\tilde{\pi} &= 3 \times 10^0 && \text{(1-digit approx.)},\\
\tilde{\pi} &= 3.1 \times 10^0 && \text{(2-digits approx.)},\\
\tilde{\pi} &= 3.14 \times 10^0 && \text{(3-digits approx.)},\\
\tilde{\pi} &= 3.141 \times 10^0 && \text{(4-digits approx.)},\\
\tilde{\pi} &= 3.1416 \times 10^0 && \text{(5-digits approx.)},\\
\tilde{\pi} &= 3.14159 \times 10^0 && \text{(6-digits approx.)},\\
&\ldots
\end{aligned} \tag{B.3}$$

The operation of rounding c to its first n significant digits is denoted as

$$\boxed{\tilde{c} = \mathcal{Q}_n\,[c]}, \tag{B.4}$$

the radix b being usually clear from the context. The notation can be simplified by eliminating the radix point "." and adjusting the exponent accordingly,

$$\begin{aligned}
\mathcal{Q}_1\,[\pi] &= 3 \times 10^0 && \text{(1-digit approx.)},\\
\mathcal{Q}_2\,[\pi] &= 31 \times 10^{-1} && \text{(2-digits approx.)},\\
\mathcal{Q}_3\,[\pi] &= 314 \times 10^{-2} && \text{(3-digits approx.)},\\
\mathcal{Q}_4\,[\pi] &= 3141 \times 10^{-3} && \text{(4-digits approx.)},\\
\mathcal{Q}_5\,[\pi] &= 31416 \times 10^{-4} && \text{(5-digits approx.)},\\
\mathcal{Q}_6\,[\pi] &= 314159 \times 10^{-5} && \text{(6-digits approx.)}\\
&\ldots
\end{aligned} \tag{B.5}$$

From these preliminary considerations, in an n-digit, radix-b finite precision arithmetic system the representable numbers are of the form

$$\boxed{\tilde{c} = \pm w \times b^q}, \tag{B.6}$$

where

- The *radix* or *base* b is an integer equal or greater than 2.
- The *unsigned significand* w is a nonnegative integer, which is expressed, in positional notation, by an n-digit base-b word.

$$w = \left(d_{n-1}d_{n-2}\ldots d_1 d_0\right)_b \triangleq \sum_{i=0}^{n} d_i \times b^i, \quad d_i \in \{0, 1, \ldots b-1\}. \tag{B.7}$$

- The *exponent* q is an integer that, depending on the arithmetic system, may or may not have an explicit encoding.

Absolute and relative *round-off errors* between c and its approximation $\mathcal{Q}_n[c]$ are denoted with

$$\begin{aligned} d_n c &\triangleq \mathcal{Q}_n[c] - c && \text{(Absolute round-off error),} \\ \delta_n c &\triangleq \frac{d_n c}{c} = \frac{\mathcal{Q}_n[c] - c}{c} && \text{(Relative round-off error).} \end{aligned} \tag{B.8}$$

B.2 FLOATING-POINT VERSUS FIXED-POINT ARITHMETIC SYSTEMS

Encoding a given number in a finite precision system involves the representation of (i) the rounded significand and (ii) the exponent q. Signed numbers can be treated by devoting one additional bit to represent the sign.

The arithmetic formats where the exponent q is explicitly encoded are known as *floating-point* arithmetic systems. Consider, for instance, a radix-10 system using two digits for the significand and one digit for a signed exponent. Positive representable quantities would range from $01_{10} \times 10^{-9}$ to $99_{10} \times 10^{9}$, covering 20 decades with a relative round-off error never larger than $\approx 4.7\%$. The single main advantage of floating-point arithmetic, therefore, is the capability to span several orders of magnitude while maintaining a limited relative round-off error throughout the represented range. Standard IEEE Std 754™-2008 [176] defines a number of floating-point formats. For instance, in the IEEE *binary32* format, 32 total bits are available, 1 bit encoding the sign, 8 bits encoding the exponent, and the remaining 23 bits being reserved for the significand. The represented range spans approximately 83 decades.

Implementation of floating-point systems involves a significant computational overhead to carry out even the fundamental arithmetic operations, because of the need to decode and encode operands prior and after every manipulation. Furthermore, *normalization* of the represented quantities is required to make representations unique: In a three-digit system, for instance, $\tilde{c} = 8.2$ can be represented either as $082_{10} \times 10^{-1}$ or as $820_{10} \times 10^{-2}$. The representation can be made unique by requiring that $100 \leq w < 1000$. More generally, in an n-digit radix-b system, one requires that $b^{n-1} \leq w < b^n$.

Because of its complexity, floating-point arithmetic is nowadays implemented in most microprocessors and high-end digital signal processors (DSPs) in dedicated *floating-point units* (FPUs). On the other hand, floating-point arithmetic is typically not supported by low-cost DSPs and microcontrollers, where floating point can be *software-emulated* when absolutely needed. In general, owing to cost and complexity constraints, floating-point arithmetic is avoided in many embedded system applications, including digital controllers considered in this book.

The formats where the exponent is not explicitly encoded are called *fixed-point* arithmetic systems. Any given quantity is represented solely by a *signed significand*, whereas the exponent remains *fixed* once and for all and therefore does not require encoding. Manipulation of fixed-point quantities can be carried out much more rapidly and efficiently. In fact, it is easy to realize that *arithmetic operations in a fixed-point environment are essentially arithmetic operations between integers*. Furthermore, representations are inherently unique, with the possible exception of the zero, which may or may not have a unique encoding depending on the format used. The drawback of such simplicity is a much larger relative error with respect to a floating-point encoding using the same number of bits — or, equivalently, the need for longer word lengths to achieve the same precision. As a comparison with the previous example, suppose two integer digits are available to represent quantities over a scale of 10^2. Positive representable numbers therefore range from $01_{10} \times 10^2$ to $99_{10} \times 10^2$, that is, two decades. The worst-case relative round-off error amounts to 50%.

B.3 BINARY TWO'S COMPLEMENT (B2C) FIXED-POINT REPRESENTATION

In this book, a radix-2 fixed-point system is considered in which a *signed* significand is encoded in two's complement notation. The binary two's complement (B2C) representation is a base-2 positional system capable of encoding both positive and negative numbers, with a unique representation of the zero. It has a number of appealing features that make it easily the most commonly adopted integer arithmetic system in today's microcontrollers, DSPs, and microprocessors.

Representable numbers are of the form

$$\boxed{x = w \times 2^q}, \tag{B.9}$$

where the signed significand w is an n-bit binary word w encoded in B2C,

$$\begin{aligned} w &= \left(b_{n-1}b_{n-2}\ldots b_1b_0\right)_{\bar{2}} \\ &\triangleq -b_{n-1} \times 2^{n-1} + \sum_{i=0}^{n-2} b_i \times 2^i, \quad b_i \in \{0,1\}. \end{aligned} \tag{B.10}$$

The exponent q is fixed once and for all and therefore does not have an explicit hardware encoding.

Bits b_{n-1} and b_0 of the significand are called *most significant bit* (MSB) and *least significant bit* (LSB) of the representation, respectively. The most significant bit b_{n-1} is also called the *sign bit*, as it is equal to 1 if and only if $w < 0$ and equal to 0 otherwise.

The range spanned by an n-bit B2C word is

$$\underbrace{-2^{n-1}}_{w_{min}} \leq w \leq \underbrace{2^{n-1}-1}_{w_{max}}, \tag{B.11}$$

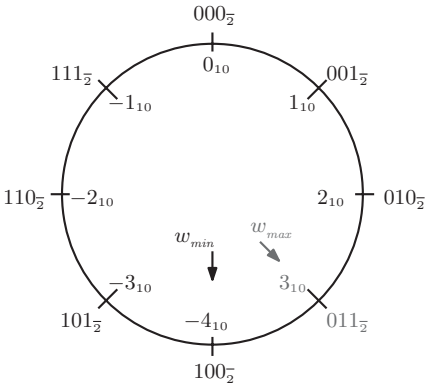


Figure B.1 Circular representation of a 3-bit B2C arithmetic system.

where w_{max} and w_{min} represent the most positive and the most negative representable numbers. For instance, the range of integers represented by a 3-bit B2C word goes from -4 to $+3$. As shown in Fig. B.1, B2C can be thought of as a circular representation: if binary one is added to the most positive number—performing the operation as if dealing with a plain base-2 representation—the most negative number is obtained.

Inset B.1 – B2C Round-off Using Matlab®

A simple Matlab® function that implements the n-digit B2C round-off operation $\mathcal{Q}_n[x]$ of a given quantity x is given in this inset:

```
function [wk,dx]    =   Qn(x,n)

x1  =   x;

neg     =   (x<0);
x       =   abs(x);

E   =   floor(log2(x));
F   =   2^(log2(x)-E);
q   =   E-(n-2);
wd  =   round(F*2^(n-2));
if (wd==1)
    wd  =   round(F*2^(n-3));
    q   =   q+1;
end;

if (neg)
    xq  =   -wd*2^q;
    s   =   ['1',dec2bin(-wd+2^(n-1),n-1)];
    wd  =   -wd;
else
    xq  =   wd*2^q;
    s   =   ['0',dec2bin(wd,n-1)];
end;
```

```
wk.xq    =   xq;
wk.w     =   wd;
wk.q     =   q;
wk.n     =   n;
wk.s     =   s;
dx       =   xq-x1;

return;
```

The above-mentioned function accepts quantity `x` to be quantized and the target word length `n`. Its outputs `wk` and `dx` are

- `wk`: A structure encoding the B2C quantity. Its fields are as follows:
  - `wk.xq`: Rounded-off quantity $\mathcal{Q}_n[x]$.
  - `wk.w`: Significand w of the B2C representation of x.
  - `wk.q`:Scale q of the B2C representation of x.
  - `wk.n`: Number of bits.
  - `wk.s`:String representation of the n-bit B2C word w.
- `dx`: Absolute round-off error $d_n x$.

For instance, `[wk,dx]=Qn(pi,5)` produces

```
wk =

    xq: 3.2500
     w: 13
     q: -2
     n: 5
     s: '01101'


dx =

    0.1084
```

or

$$\mathcal{Q}_5[\pi] = 01101_{\bar{2}} \times 2^{-2} = 13_{10} \times 2^{-2} = 3.25_{10}. \tag{B.12}$$

B.4 SIGNAL NOTATION

Referring to (B.9), one can interpret x as a generic *signal* and word w as *representing* x over a scale 2^q. A notation is now introduced that is extensively used in Chapter 6 and that makes the relationship between x and w more explicit, without the need to rewrite (B.9) every time. Define

$$\boxed{[x]_q^n \triangleq w}, \tag{B.13}$$

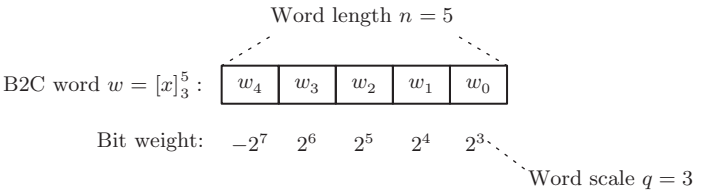


Figure B.2 Signal notation of a B2C word $w = [x]_3^5$.

so that the relationship between x and w becomes

$$x = [x]_q^n \times 2^q. \tag{B.14}$$

In other words, $[x]_q^n$ is the *unique* n-bit B2C word that represents signal x if its least significant bit is given a weight equal to 2^q. In a sense, this notation puts emphasis on the signal x *represented* by a B2C word rather than on the word itself. As an example, Fig. B.2 shows a pictorial representation of the signal notation for a B2C word $w = [x]_3^5$ representing a signal x with 5 bits and over a scale 2^3.

B.5 MANIPULATION OF B2C QUANTITIES AND HDL EXAMPLES

This section summarizes the most common arithmetic and bitwise operations on B2C words, along with the corresponding coding in VHDL or Verilog [173, 174, 177, 178].

As far as VHDL is concerned, data types and packages standardized in the *IEEE Standard VHDL Synthesis Packages* document [172] are employed. The standard provides a description of data types, arithmetic, and logic operators, which are likely supported by any synthesis tool. B2C quantities are represented by means of the `signed` data type as defined in the `NUMERIC_STD` package, which in turn is built upon the IEEE-defined package `STD_LOGIC_1164` [171]. Hence, the following library configuration preamble is assumed for all VHDL examples in this book:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
```

An n-bit word $[x]_q^n$ representing signal x can then be defined as

```
signal x : signed(n-1 downto 0);        --  [n,q]
```

Observe that the scale q of $[x]_q^n$ is not encoded anywhere and remains *implicit*, as a fixed-point arithmetic representation is used. For such reason and for improved code readability, the comment `-- [n,q]` is included, which reports both the size and the scale of `x`.

The reference documentation for the Verilog language definitions and synthesizable constructs is in [175, 178]. For manipulating B2C words, the Verilog data type `signed` is employed, and words $[x]_q^n$ introduced earlier can be defined as

```
wire signed [n-1:0] x;          //  [n,q]
```

All binary and arithmetic operations described in this section implement, in hardware, purely *combinational* functions. Therefore, they are coded as either concurrent statements (if VHDL is used) or continuous assignments (in Verilog examples). Recall that the basic syntax for a VHDL concurrent statement makes use of the <= operator,

```
y   <=  x;                  --  Concurrent statement
```

where x and y are two VHDL signals. A Verilog continuous assignment, on the other hand, has the basic syntax

```
assign y = x;           //  Continuous assignment
```

where x and y are declared as `wire signed` nets.

B.5.1 Sign Extension

When extending the number of binary digits from n to $n+k$, k replicas of the sign bit are to be written in the most significant portion of the word. The reason why this works is that the contribution of the sign bit can always be written as

$$-b_{n-1} \times 2^{n-1} = -b_{n-1} \times 2^{n} + b_{n-1} \times 2^{n-1}, \tag{B.15}$$

which allows one to arbitrarily replicate the sign without altering the represented value.

For instance, if

$$\begin{aligned} w = \quad & [x]_q^5 = 10010_{\bar{2}} = -14_{10}, \\ r = \quad & [y]_q^5 = 00111_{\bar{2}} = \quad 7_{10} \end{aligned} \tag{B.16}$$

are two 5-bit B2C words, their extensions to $5+3=8$ bits are

$$\begin{aligned} w' = \quad & [x]_q^8 = 11110010_{\bar{2}} = -14_{10}, \\ r' = \quad & [y]_q^8 = 00000111_{\bar{2}} = \quad 7_{10}, \end{aligned} \tag{B.17}$$

as one can easily verify.

In signal notation, a 1-bit sign extension is simply denoted as

$$[x]_q^{n+1} \leftarrow [x]_q^n \,, \tag{B.18}$$

and, more generally,

$$[x]_q^{n+k} \leftarrow [x]_q^n \,, \quad (k \geq 0) \tag{B.19}$$

for a k-bit sign extension. Observe that sign extension does not modify the signal represented by the word.

As an example, Fig. B.3 shows a pictorial representation of $[x]_3^6 \leftarrow [x]_3^5$.

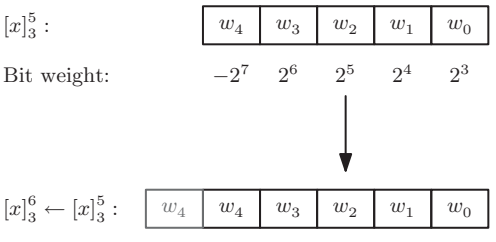


Figure B.3 Sign extension $[x]_3^6 \leftarrow [x]_3^5$.

B.5.2 Alignment

Two B2C words $[x]_q^n$ and $[y]_l^p$ are *aligned* if $q = l$, that is, if they express signals x and y over the same scale. Alignment is often necessary before arithmetic operations such as addition, subtraction, or comparisons. For instance, if

$$\begin{aligned} x = \quad & 01001_{\bar{2}} \times 2^2 \quad = \quad 36_{10}, \\ y = \quad & 10_{\bar{2}} \times 2^0 \quad = \quad -2_{10} \end{aligned} \tag{B.20}$$

are two 5-bit and 2-bit signals represented in different scales, their aligned representation is

$$\begin{aligned} x = \quad & 0100100_{\bar{2}} \times 2^0 \quad = \quad 36_{10}, \\ y = \quad & 10_{\bar{2}} \times 2^0 \quad = \quad -2_{10}, \end{aligned} \tag{B.21}$$

where the represented values are obviously unaltered, but the significand of x is now represented on the same scale as y's and, consequently, on a larger number of bits.Therefore, alignment consists of an LSB extension of the word represented on the largest scale.

The 2-bit LSB extension of $[x]_2^5 = 01001_{\bar{2}}$ to $[x]_0^7 = 0100100_{\bar{2}}$ is indicated, in signal notation, as

$$[x]_0^7 \leftarrow [x]_2^5 . \tag{B.22}$$

In general, given $[x]_q^n$, one can increase the word length and correspondingly decrease the scale without altering the represented signal,

$$[x]_{q-k}^{n+k} \leftarrow [x]_q^n , \quad (k \geq 0), \tag{B.23}$$

an operation that corresponds to the LSB extension mentioned earlier.

On the basis of such observation, if $[x]_q^n$ and $[y]_l^p$are two words of different lengths and different weights, with $q > l$, alignment of $[x]_q^n$ to $[y]_l^p$ is achieved by

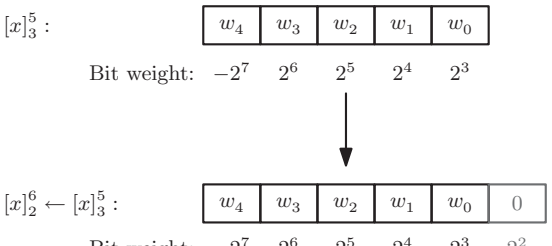


Figure B.4 One-bit LSB extension $[x]_2^6 \leftarrow [x]_3^5$.

adding and subtracting $(q - l)$ from n and q, respectively,

$$[x]_l^{n+q-l} \leftarrow [x]_q^n\,, \quad (q > l). \tag{B.24}$$

Figure B.4 shows a pictorial representation of a $[x]_2^6 \leftarrow [x]_3^5$ LSB extension.

Inset B.2 – VHDL Sign Extension and Alignment

Sign extension of a word $[x]_q^n$ makes use of the VHDL concatenation operator &.Let

```
signal x     : signed(n-1 downto 0);
signal x_ext : signed(n downto 0);
```

A 1-bit sign extension of x is coded as

```
x_ext <= x(n-1)&x;
```

Similarly, for alignment, a 1-bit LSB extension of x is coded as

```
x_ext <= x&'0';
```

Inset B.3 – Verilog Sign Extension and Alignment

In a similar manner, sign extension of an n-bit wire net x to an $(n + 1)$-bit wire net x_ext, both declared as signed, is accomplished in Verilog using the concatenation operator {}:

```
wire signed [n-1:0] x;
wire signed [n:0] x_ext;
assign x_ext = {x[n-1],x};
```

For alignment, the 1-bit LSB extension of x is coded as

```
wire signed [n-1:0] x;
wire signed [n:0] x_ext;
assign x_ext = {x,1'b0};
```

Care must be taken, in general, when concatenating `signed` quantities, as *concatenate results are unsigned, regardless of the operands* [178]. The above-mentioned statements work correctly because no sign extension of {`x[n-1],x`} or {`x,1'b0`} is required during the assignment, but just an implicit—and irrelevant—typecasting occurs. If desired, however, the typecasting operator `$signed` can be explicitly invoked. For instance, the above-mentioned 1-bit LSB extension would become

```
assign  x_ext = $signed({x,1'b0});
```

B.5.3 Sign Reversal

In an n-bit B2C system, every number has its additive inverse except for the most negative number. Therefore, an $(n+1)$-bit word is required when changing the sign of an n-bit B2C quantity.

Operatively, the sign of an n-bit word w can be changed by first extending its representation to $n+1$ bits, then bit-wise negating all the bits, and finally adding one. For instance, if $w = 0110_{\bar{2}} = 6_{10}$, then -6_{10} is calculated as

$$\begin{aligned} w &= 0110_{\bar{2}} && \text{(original word)} \\ &\rightarrow 00110_{\bar{2}} && \text{(sign extension)} \\ &\rightarrow 11001_{\bar{2}} && \text{(bit-wise negation)} \\ &\rightarrow 11010_{\bar{2}} = -6_{10} && \text{(add } 00001_{\bar{2}}\text{).} \end{aligned} \tag{B.25}$$

In signal notation, sign reversal is indicated as

$$[-x]_q^{n+1} \leftarrow -[x]_q^n \,. \tag{B.26}$$

Inset B.4 – VHDL Sign Reversal

Define

```
signal x     : signed(n-1 downto 0);     --  [n,q]
signal x_ext : signed(n downto 0);       --  [n+1,q]
signal z     : signed(n downto 0);       --  [n+1,q]
```

VHDL sign reversal of `x` is accomplished by first sign-extending `x`, then employing the "-" unary operator defined in package `NUMERIC_STD`:

```
x_ext <= x(n-1)&x;
z     <= -x_ext;
```

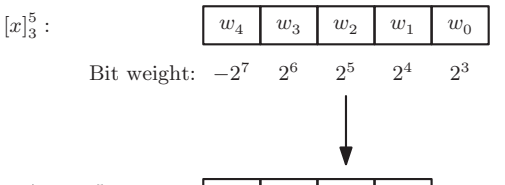


Figure B.5 One-bit LSB truncation $[y]_4^4 \leftarrow [x]_3^5$.

Inset B.5 – Verilog Sign Reversal

Verilog coding of sign reversal of an n-bit signal `x` is accomplished by first sign-extending `x`, then employing the "-" unary operator:

```
wire signed [n-1:0] x;              //  [n,q]
wire signed [n:0]   x_ext;          //  [n+1,q]
wire signed [n:0] z;                //  [n+1,q]
assign  x_ext = {x[n-1],x};
assign  z     = -x_ext;
```

where `z` is a $(n+1)$-bit signal.

B.5.4 LSB and MSB Truncation

Truncation—that is, removal—one or more LSBs or MSBs from an n-bit B2C word destroys, in general, the represented number. This operation is nonetheless discussed as it is frequently employed during bit manipulation. In signal notation, a 1-bit LSB truncation is denoted as

$$[y]_{q+1}^{n-1} \leftarrow [x]_q^n \,. \tag{B.27}$$

In general, $y = x$ if and only if the least significant bit of $[x]_q^n$ is zero, that is, if and only if x is a multiple of 2^q. Otherwise, $y = x - 2^q$.

More generally, truncation of the first k least significant bits of a word $[x]_q^n$ is denoted as

$$[y]_{q+k}^{n-k} \leftarrow [x]_q^n \,, \quad (0 \le k \le n-1) \,, \tag{B.28}$$

and $y = x$ if and only if x is a multiple of 2^{q+k}. Figure B.5 illustrates a pictorial representation of a 1-bit LSB truncation $[y]_4^4 \leftarrow [x]_3^5$.

A 1-bit MSB truncation, on the other hand, is denoted as

$$[y]_q^{n-1} \leftarrow [x]_q^n \,, \tag{B.29}$$

and, more generally for a k-bit MSB truncation, one has

$$[y]_q^{n-k} \leftarrow [x]_q^n \,, \quad (0 \le k \le n-1) \,. \tag{B.30}$$

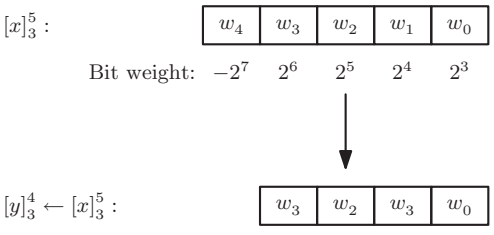


Figure B.6 One-bit MSB truncation $[y]_3^4 \leftarrow [x]_3^5$.

Figure B.6 illustrates a pictorial representation of a 1-bit MSB truncation $[y]_3^4 \leftarrow [x]_3^5$.

Inset B.6 – VHDL Truncation of a B2C Quantity

Truncation of a signal `x` is simply coded, in VHDL, by assigning the proper portion of `x` to a signal of smaller length. Focusing, for definiteness, on a 1-bit LSB truncation, define `x` and `y` as

```
signal  x : signed(n-1 downto 0);   --  [n,q]
signal  y : signed(n-2 downto 0);   --  [n-1,q+1]
```

Truncation of `x` is then coded as

```
y <= x(n-1 downto 1);
```

Inset B.7 – Verilog Truncation of a B2C Quantity

The Verilog construct for a 1-bit LSB truncation is by all means analogous to the VHDL one:

```
wire signed [n-1:0] x;      //  [n,q]
wire signed [n-2:0] y;      //  [n-1,q+1]
assign  y = x[n-1:1];
```

where `y` is a $(n-1)$-bit Verilog signal. Note that, as with the concatenation operator, *part-select results are unsigned, regardless of the operands even if the part-select specifies the entire vector* [175]. In other words, `x[n-1:1]` on the assignment right-hand side is of `unsigned` type. Nonetheless, the above-mentioned example works correctly, as no sign extension of `x[n-1:1]` is required during the assignment and just a conversion back to `signed` occurs. If desired, however, the `$signed` typecasting operator can be explicitly used:

```
assign  y = $signed(x[n-1:1]);
```

B.5.5 Addition and Subtraction

Representing the sum of two n-bit B2C words requires an $(n+1)$-bit word. For instance, in a 3-bit B2C system in which numbers range from -4_{10} to $+3_{10}$, possible sums of its elements range between -8_{10} and $+6_{10}$.

The simplest way to handle the need for an extended range is a preliminary sign extension of the addends. Addition between two B2C words is then accomplished with the same rules of plain base-2 addition. For instance, let

$$\begin{aligned} w = \quad [x]_q^3 = \quad 011_{\bar{2}} = \quad 3_{10}, \\ r = \quad [y]_q^3 = \quad 111_{\bar{2}} = \quad -1_{10} \end{aligned} \tag{B.31}$$

be the two 3-bit addends. Their sum is then accomplished over 4 bits as

$$\begin{array}{lcl} 0011_{\bar{2}} & + & \qquad\qquad 3_{10} \quad + \\ 1111_{\bar{2}} & = & \text{i.e.,} \qquad -1_{10} \quad = \\ 0010_{\bar{2}} & & \qquad\qquad 2_{10}. \end{array} \tag{B.32}$$

In a similar manner, subtraction of two n-bit B2C quantities can be exactly represented over $n+1$ bits. Difference between words w and r defined earlier can be accomplished as the B2C sum of w with $-r$. The latter is obtained, according to the discussion in the previous section, with a preliminary sign extension of r, followed by its bit-wise negation and a unit increment,

$$-r = 0000_{\bar{2}} + 0001_{\bar{2}} = 0001_{\bar{2}}. \tag{B.33}$$

Hence, $w - r$ becomes

$$\begin{array}{lcl} 0011_{\bar{2}} & + & \qquad\qquad 3_{10} \quad + \\ 0000_{\bar{2}} & + & \qquad\qquad 0_{10} \quad + \\ 0001_{\bar{2}} & = & \text{i.e.,} \qquad 1_{10} \quad = \\ 0100_{\bar{2}} & & \qquad\qquad 4_{10}. \end{array} \tag{B.34}$$

In signal notation, addition or subtraction between two n-bit words and storage into a $(n+1)$-bit word is denoted with

$$[x \pm y]_q^{n+1} \leftarrow [x]_q^n \pm [y]_q^n \,. \tag{B.35}$$

Observe that the operation only makes sense *as long as the two addends are aligned*. If not, a preliminary alignment is required.

When the two addends are aligned but have different lengths n and p, their sum or difference can always be correctly represented by a $(\max(n,p)+1)$-bit word,

$$[x \pm y]_q^{\max(n,p)+1} \leftarrow [x]_q^n \pm [y]_q^p \,. \tag{B.36}$$

Inset B.8 – VHDL Addition of B2C Quantities

Addition of two *aligned* words $[x]_q^n$ and $[y]_q^p$ can be VHDL-coded as follows. Consider the signal declarations

```
signal x : signed(n-1 downto 0);          --  [n,q]
signal y : signed(p-1 downto 0);          --  [p,q]
signal z : signed(max(n,p) downto 0);     --  [max(n,p)+1,q]
```

In the above-mentioned code, it is assumed that a function `max` is defined that returns the largest of its two arguments. Addition between `x` and `y` can be accomplished by first sign extending both signals, then using the `+` operator defined on `signed` data types to store the result into `z`:

```
z <= (x(n-1)&x)+(y(p-1)&y);
```

Note that operator +, defined in the `NUMERIC_STD` package, evaluates the result to a vector whose length is the largest between the lengths of the operands [172]. Therefore, both sides of the foregoing concurrent statement have the same length.

Subtraction of two B2C quantities is accomplished with the "-" operator, which automatically implements a signed difference as discussed earlier. All considerations and constructs examined earlier apply with no other modifications.

Inset B.9 – Verilog Addition of B2C Quantities

Addition of two aligned words $[x]_q^n$ and $[y]_q^p$ can be Verilog-coded as follows. Consider the signal declarations

```
wire    signed  [n-1:0] x;        //  [n,q]
wire    signed  [p-1:0] y;        //  [p,q]
wire    signed  [max(n,p):0] z;   //  [max(n+p)+1,q]
```

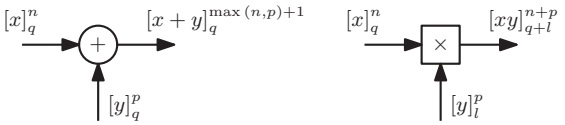
As in VHDL, addition between `x` and `y` can be accomplished by first sign-extending both signals, then using the `+` operator defined on `signed` data types to store the result into `z`:

```
wire signed [n:0] x_ext = {x[n-1],x};
wire signed [p:0] y_ext = {y[p-1],y};
assign z                = x_ext + y_ext;
```

Subtraction is accomplished in a similar manner with the use of Verilog operator "-".

B.5.6 Multiplication

The product of an n-bit B2C word with a p-bit B2C word can be exactly represented by a $(n+p)$-bit B2C word. In particular, multiplication of two n-bit words requires a $(2n)$-bit word to store the product. For instance, in a 3-bit B2C system, possible products between its elements range between -12_{10} and $+16_{10}$.



$$[x+y]_q^{\max(n,p)+1} \leftarrow [x]_q^n + [y]_q^p \qquad [xy]_{q+l}^{n+p} \leftarrow [x]_q^n \times [y]_l^p$$

Figure B.7 Block diagram symbols for addition and multiplication.

In signal notation, multiplication between two words and storage into an output word is denoted as

$$[xy]_{q+l}^{n+p} \leftarrow [x]_q^n \times [y]_l^p \,. \tag{B.37}$$

Block diagram symbols for addition and multiplication are depicted in Fig. B.7. It should be noted that *multiplication changes the scale of the represented quantity*.

Inset B.10 – VHDL Multiplication of B2C Quantities

VHDL multiplication of two B2C quantities is simply accomplished via the `*` operator defined for `signed` data types. Let

```
signal x : signed(n-1 downto 0);      --  [n,q]
signal y : signed(p-1 downto 0);      --  [p,l]
signal z : signed(n+p-1 downto 0);    --  [n+p,q+l]
```

be signals representing B2C words of lengths n, p, and $n+p$, respectively. B2C multiplication between `x` and `y` is simply coded as

```
z <= x*y;
```

Inset B.11 – Verilog Multiplication of B2C Quantities

Verilog multiplication of two B2C quantities is simply accomplished via the `*` operator defined for `signed` data types. Let

```
wire    signed  [n-1:0] x;        //  [n,q]
wire    signed  [p-1:0] y;        //  [p,l]
wire    signed  [n+p-1:0] z;      //  [n+p,q+l]
```

be signals representing B2C words of lengths n, p, and $n+p$, respectively. B2C multiplication between `x` and `y` is coded as

```
assign z = x*y;
```

B.5.7 Overflow Detection and Saturated Arithmetic

If result (B.34) is to be stored in a 3-bit word, an *overflow* would occur, as 4_{10} is not representable in a 3-bit B2C system. Simply dropping the most significant bit from the $(3+1)$-bit result would yield $100_{\bar{2}} = -4_{10}$.

Given an $(n+1)$-bit B2C word $w = [x]_q^{n+1}$, it is therefore of interest to determine whether x could be represented in n bits, a check often referred to as *overflow detection*. Overflow detection can be accomplished in a variety of ways. If

$$w = [x]_q^{n+1} = (w_n \dots w_0)_{\bar{2}}, \tag{B.38}$$

is a generic $(n+1)$-bit B2C word, the range overflow occurs if and only if the two most significant bits of w differ,

$$\text{OV} = w_n \text{ XOR } w_{n-1}. \tag{B.39}$$

Whenever OV = 0, truncation of the MSB does not alter the represented quantity. It can be verified that the above-mentioned criterion would correctly detect an overflow condition in the case of sum (B.34).

The above-mentioned result can be generalized to an $(n+l)$-bit word

$$w = [x]_q^{n+l} = (w_{n+l-1} \dots w_0)_{\bar{2}}. \tag{B.40}$$

Word w can be exactly stored in an n-bit word if and only the $l+1$ most significant bits of w are equal,

$$\text{OV} = \text{NOT}\left(w_{n+l-1} = w_{n+l-2} = \dots = w_{n-1}\right). \tag{B.41}$$

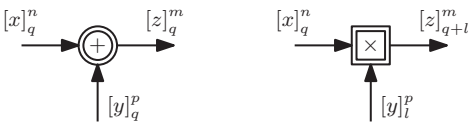
The action to be undertaken when an overflow occurs depends on the system in which the B2C arithmetic is implemented. A frequent provision *saturates* an overflowed result to the most positive or the most negative representable number. In such *saturated arithmetic*, the result of (B.34) would be $011_{\bar{2}} = 3_{10}$, the most positive representable number in a 3-bit B2C system.

In general, a saturated assignment involving truncation of l MSBs is indicated as

$$[y]_q^n \Leftarrow [x]_q^{n+l}, \tag{B.42}$$

The above-mentioned assignment is to be interpreted as follows: whenever the result of the operation on the right-hand side of the assignment overflows, the left-hand side is set to the most positive or the most negative values representable on n bits, depending on the overflow direction. If no overflow occurs, a simple MSB truncation of the result is accomplished. For instance, a saturated sign reversal is denoted as

$$[z]_q^n \Leftarrow -[x]_q^n, \tag{B.43}$$



$[z]_q^m \Leftarrow [x]_q^n + [y]_q^p$
$m \leq \max(n, p)$

$[z]_{q+l}^m \Leftarrow [x]_q^n \times [y]_l^p$
$m < n + p$

Figure B.8 Block diagram symbols for saturated addition and saturated multiplication.

whereas signal notation for saturated addition/subtraction and multiplication becomes

$$[z]_q^m \Leftarrow [x]_q^n \pm [y]_q^p \quad (m \leq \max(n, p)) \tag{B.44}$$

and

$$[z]_{q+l}^m \Leftarrow [x]_q^n \times [y]_l^p \quad (m < n + p). \tag{B.45}$$

Block diagram symbols used for saturated addition and multiplication are shown in Fig. B.8.

Inset B.12 – VHDL Saturated Addition and Multiplication

A combinational saturated adder can be VHDL-coded as follows. Consider the entity declaration first:

```
entity saturated_adder is
    generic (
        n, p, m : integer                    --  m <= max(n,p)+1
        );

    port (
        x       : in signed(n-1 downto 0);  --  [n,q]
        y       : in signed(p-1 downto 0);  --  [p,q]
        z       : out signed(m-1 downto 0); --  [m,q]
        OV      : out std_logic;
        op      : in std_logic
        );
end saturated_adder;
```

Entity `saturated_adder` operates on two inputs `x` and `y`, n-bit and p-bit long, respectively, and outputs their saturated sum as a m-bit word `z`, with $m \leq \max(n, p) + 1$. Input flag `op` specifies whether the sum is actually an addition (if `op='1'`), or a subtraction (if `op='0'`). Output flag `OV` is asserted in the presence of a range overflow with respect to the target word length m.
A possible implementation of the saturated adder is as follows:

```
1  architecture saturated_adder_arch of saturated_adder is
2
3      function MAX(LEFT, RIGHT: INTEGER) return INTEGER is
4      begin
```

```
 5        if LEFT > RIGHT then return LEFT;
 6        else return RIGHT;
 7           end if;
 8      end;
 9
10      signal zx       :   signed(max(n,p) downto 0);
11      signal OVi      :   std_logic;
12      constant wmax   :   signed(m-2 downto 0)    :=  (others=>'1');
13      constant wmin   :   signed(m-2 downto 0)    :=  (others=>'0');
14
15  begin
16
17      zx      <=  (x(n-1)&x) + (y(p-1)&y) when op='1' else
18                  (x(n-1)&x) - (y(p-1)&y);
19
20      overflow_detect :   process(zx)
21          variable temp   :   std_logic;
22          begin
23              temp    :=  '0';
24              for I in m to max(n,p) loop
25                  if ((zx(I) xor zx(m-1))='1') then
26                      temp    :=  '1';
27                  end if;
28              end loop;
29              OVi <=  temp;
30          end process;
31
32      z   <=  ('0'&wmax)  when    OVi='1' AND zx(max(n,p))='0'   else
33              ('1'&wmin)  when    OVi='1' AND zx(max(n,p))='1'   else
34              zx(m-1 downto 0);
35
36      OV  <=OVi;
37
38  end saturated_adder_arch;
```

In the preceding example, an overflow check is accomplished by process `overflow_detect` defined in line 20.
Entity declaration for a combinational saturated multiplier is

```
entity saturated_multiplier is
    generic (
        n, p, m :   integer     --  m<=n+p
        );

    port (
        x       :   in signed(n-1 downto 0);        --  [n,q]
        y       :   in signed(p-1 downto 0);        --  [p,l]
        z       :   out signed(m-1 downto 0);       --  [m,q+l]
        OV      :   out std_logic
        );
end saturated_multiplier;
```

In this case, the word length of the saturated product is $m \leq n + p$. An implementation example of the foregoing entity is

```
architecture saturated_multiplier_arch of saturated_multiplier is

    signal zx       :   signed(n+p-1 downto 0);
    signal OVi      :   std_logic;
    constant wmax   :   signed(m-2 downto 0)  :=  (others=>'1');
    constant wmin   :   signed(m-2 downto 0)  :=  (others=>'0');

begin

    zx      <=  x*y;

    overflow_detect :   process(zx)
        variable temp   :   std_logic;
        begin
            temp    :=  '0';
            for I in m to n+p-1 loop
                if ((zx(I) xor zx(m-1))='1') then
                    temp    :=  '1';
                end if;
            end loop;
            OVi <=  temp;
        end process;

    z   <=  ('0'&wmax)  when    OVi='1' AND zx(n+p-1)='0'   else
            ('1'&wmin)  when    OVi='1' AND zx(n+p-1)='1'   else
            zx(m-1 downto 0);

    OV  <=OVi;

end saturated_multiplier_arch;
```

Inset B.13 – Verilog Saturated Addition and Multiplication

Following the previous VHDL example, a combinational saturated adder can be Verilog-coded as follows:

```
module saturated_adder(x,y,z,OV,op);

    function integer max;
        input integer left, right;
        if (left>right)
            max = left;
        else
            max = right;
    endfunction

    parameter n;
    parameter p;
```

```
13      parameter m;                //  Assuming m <= max(n,p)+1
14      parameter mx = max(n,p)+1;
15
16      input   signed [n-1:0]  x;
17      input signed [p-1:0]    y;
18      output reg signed   [m-1:0]     z;
19      input op;
20
21      output reg OV;
22
23      wire signed [n:0]   xx = {x[n-1],x};
24      wire signed [p:0]   yx = {y[p-1],y};
25      wire signed [mx-1:0]    zx;
26
27      assign zx = (op==1'b1) ? xx+yx : xx-yx;
28
29      reg temp;
30      integer I;
31      always @(zx)
32          begin
33              temp = 1'b0;
34              for (I=m;I<=mx-1;I=I+1)
35                  begin
36                      if ((zx[I]^zx[m-1])==1'b1)
37                          temp = 1'b1;
38                  end
39              OV = temp;
40          end
41
42      always @(OV,zx)
43          case (OV)
44              1'b0:   z = zx[m-1:0];
45              1'b1:
46                  begin
47                      if (zx[mx-1]==1'b0)
48                          z = {1'b0,{(m-1){1'b1}}};
49                      else
50                          z = {1'b1,{(m-1){1'b0}}};
51                  end
52          endcase
53
54  endmodule
```

Overflow check is implemented by the `for` loop contained in the `always` statement in line 42.

Similarly, Verilog code for a combinational saturated multiplier is

```
1   module saturated_multiplier(x,y,z,OV);
2
3       parameter n;
4       parameter p;
```

```
 5      parameter m;                //  Assuming m <= n+p
 6
 7      input   signed [n-1:0]  x;          //  [n,q]
 8      input   signed [p-1:0]  y;          //  [p,l]
 9      output  reg [m-1:0] z;              //  [m,q+l]
10
11      output reg OV;
12
13      wire signed [n+p-1:0]   zx;
14      assign zx = x*y;
15
16      reg temp;
17      integer I;
18      always @(zx)
19          begin
20              temp = 1'b0;
21              for (I=m;I<=n+p-1;I=I+1)
22                  begin
23                      if ((zx[I]^zx[m-1])==1'b1)
24                          temp = 1'b1;
25                  end
26              OV = temp;
27          end
28
29      always @(OV,zx)
30          case (OV)
31              1'b0:   z = zx[m-1:0];
32              1'b1:
33                  begin
34                      if (zx[n+p-1]==1'b0)
35                          z = {1'b0,{(m-1){1'b1}}};
36                      else
37                          z = {1'b1,{(m-1){1'b0}}};
38                  end
39          endcase
40
41  endmodule
```