

Chapter 1

Description of tutorial unit

This part of the tutorial describes a floating-point multiplier (almost) compliant with the IEEE standard. The floating-point multiplier is part of the floating-point unit which handles the arithmetic operations in modern microprocessors. In the following sections, the IEEE standard, the floating-point multiplication algorithm and other information necessary to design the unit are given.

1.1 The IEEE standard for floating point arithmetic

The IEEE (Institute of Electrical and Electronics Engineers) has produced a standard for floating point arithmetic. This standard specifies how single precision (32 bit) and double precision (64 bit) floating point numbers are to be represented, as well as how arithmetic should be carried out on them.

The IEEE single precision floating point standard representation requires a 32 bit word. The first bit is the sign bit, S , the next eight bits are the exponent bits, E , and the final 23 bits are the fraction, or significand M :

S	EEEEEEEE	MMMMMMMMMMMMMMMMMMMMMMMM
31	30	23 22
		0

The value V represented by the word may be determined as follows:

$$V = (-1)^S \cdot (1.M) \cdot 2^{E-127} \quad \text{with } 0 < E < 255$$

where " $1.M$ " is intended to represent the binary number created by prefixing M with an implicit leading 1 and a binary point.

The special values or the exceptions are the following:

- If $E = 255$ and M is nonzero, then $V = NaN$ ("Not a number")
- If $E = 255$ and M is zero and S is 1, then $V = -\infty$
- If $E = 255$ and M is zero and S is 0, then $V = \infty$
- If $E = 0$ and M is zero and S is 1, then $V = -0$
- If $E = 0$ and M is zero and S is 0, then $V = 0$

Finally, if $E = 0$ and M is nonzero, then $V = (-1)^S \cdot 2^{-126} \cdot (0.M)$ These are "unnormalized" values.

In Table 1.1 are reported some examples. Additional information on floating-point representation can be found in [1] and [2].

S_x	E_x	M_x	V
0	00000000	000000000000000000000000	= 0
1	00000000	000000000000000000000000	= -0
0	11111111	000000000000000000000000	= ∞
1	11111111	000000000000000000000000	= $-\infty$
0	11111111	000001000000000000000000	= NaN
1	11111111	00100010001001010101010	= NaN
0	10000000	000000000000000000000000	= $+1 \cdot 2^{128-127} \cdot (1.0)_2 = 2$
0	10000001	101000000000000000000000	= $+1 \cdot 2^{129-127} \cdot (1.101)_2 = 6.5$
1	10000001	101000000000000000000000	= $-1 \cdot 2^{129-127} \cdot (1.101)_2 = -6.5$
0	00000001	000000000000000000000000	= $+1 \cdot 2^{1-127} \cdot (1.0)_2 = 2^{-126}$
0	00000000	100000000000000000000000	= $+1 \cdot 2^{-126} \cdot (0.1)_2 = 2^{-127}$
0	00000000	000000000000000000000001	= $+1 \cdot 2^{-126} \cdot (0.000000000000000000000001)_2$ = 2^{-149} (Smallest positive value)

Table 1.1: Examples of floating point representation.

1.2 Floating Point Multiplication

It is now illustrated the algorithm to perform the multiplication of two floating-point numbers (x and y) represented according to the IEEE standard.

Both x and y are normalized operands represented by $(S_x; M_x; E_x)$ and $(S_y; M_y; E_y)$. The algorithm can be divided into the following steps:

1. Perform multiplication of significands

$$M_z = M_x \times M_y$$

2. Add exponents:

$$E_z = E_x + E_y - B$$

3. Compute sign: $S_z = S_x \oplus S_y$

4. Normalize result (if needed)

5. Round the result according the specified mode

1.2.1 Multiplication of significands

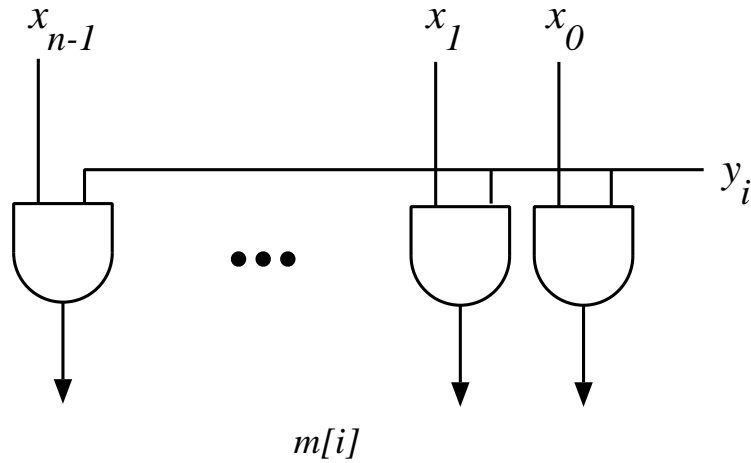
The multiplication of the two significands is usually done as we learned in elementary school. This "*paper & pencil*" method, can be directly implemented in the unsigned binary representation. For example, the multiplication of the two 3-bit operands $M_x = 1.75 = (1.11)_2$ and $M_y = 1.50 = (1.10)_2$ is performed as

$$\begin{array}{r}
 \begin{array}{r}
 1 \ 1 \ 1 \ \times \\
 1 \ 1 \ 0 \ = \\
 \hline
 0 \ 0 \ 0 \\
 1 \ 1 \ 1 \\
 \hline
 1 \ 1 \ 1 \\
 1 \ 0 \ 1 \ 0 \ 1 \ 0
 \end{array}
 \end{array}$$

Or $M_x \times M_y = 1.75 \times 1.50 = 2.625 = (10.1010)_s$.

Based on this example, we observe:

1. If we multiply two n -bit operands the result is a $2n$ -bit number. More in general, the multiplication of a n -bit and a m -bit operand results into a $(n + m)$ -bit product.

Figure 1.1: Generation of multiples of x .

2. If the operands represent normalized fractional numbers, the result might not be normalized.

$$1 \leq M_x < 2 \quad \text{and} \quad 1 \leq M_y < 2 \quad \Rightarrow \quad 1 \leq (M_x \times M_y) < 4$$

The algorithm can be formalized as follows:

$$z = \sum_{i=0}^{n-1} xy_i r^i$$

where y_i are the digits (bits) of the n -bit operand y (M_y), and r is the radix of the multiplication ($r = 2$ in the example). The algorithm can be implemented in two steps:

1. The generation of the multiples of the multiplicand shifted r^i positions to the right

$$(x \times y_i) r^i$$

This task can be accomplished with a simple circuit such as the one in Figure 1.1.

2. The multi-operand addition of the multiples generated in step 1.

The accumulation of those multiples can be done with a linear array of adders or a tree of adders [2].

More details on the implementation of the multiplication algorithm are given in [2].

1.2.2 Addition of exponents and sign computation

In the multiplication of two exponential numbers, the exponent of the product is the sum of the two exponents. However, because the IEEE floating-point format is biased, we need to subtract this bias (B) once:

$$M_x \cdot 2^{E_x} \times M_y \cdot 2^{E_y} = (M_x \times M_y) \cdot 2^{E_x + E_y}$$

where

$$E_x = E_{M_x} + B \quad \text{and} \quad E_y = E_{M_y} + B \quad \Rightarrow \quad E_x + E_y = E_{M_x} + E_{M_y} + 2B$$

and, therefore,

$$E_z = E_x + E_y - B$$

The computation of the sign is straightforward. As we learned in elementary school

S_x	S_y	S_z	
+	+	+	0
+	-	-	1
-	+	-	1
-	-	+	0

the sign of the product can be easily computed by using an exclusive-or gate:

$$S_z = S_x \oplus S_y$$

1.2.3 Normalization and Rounding

For this tutorial, we consider only the IEEE rounding method *round-to-the-nearest*, and ignore the ties (for more details see [2]).

As mentioned above, the result of significand multiplication could be not normalized, and a normalization step is required. Because $1 \leq M_z < 4$ at most a one-position left shift is required when $M_z > 2$. In case normalization is required, the exponent has to be incremented by one unit.

The rounding to the nearest is performed by adding a 1 to the bit in position 2^{-n} (after normalization) and by truncating after bit in position $2^{-(n-1)}$ as indicated in Table 1.2. The bit in position 2^{-n} is called **guard** bit. Care must be taken that the rounded $M_z < 2$, otherwise a post-normalization has to be performed.


bit weight:	2^0	.	2^{-1}	2^{-2}	...	$2^{-(n-1)}$	2^{-n}	$2^{-(n+1)}$...
norm. result	1	.	w	w	...	w	w	w	...
rounding							1		
M_z	1	.	z	z	...	z			
									
n -bit									

Table 1.2: Rounding to the nearest scheme.

1.2.4 Examples

We now show two examples of multiplication of significands and rounding. The examples are made for 6-bits operands normalized in $[1.0, 2.0)$, but the method can be easily extended to the single (24 bits) and double (54 bits) precision IEEE formats.

Example	M_x	M_y	M_z	
1	1.3125 (1.01010) ₂	1.40625 (1.01101) ₂	1.84375 (1.11011) ₂	Table 1.3
2	1.3125 (1.01010) ₂	1.75 (1.11000) ₂	1.15625 (1.00101) ₂	Table 1.4

$$\begin{array}{cccccccccccccc}
 & & & & & & & & 1 & 0 & 1 & 0 & 1 & 0 & \times \\
 & & & & & & & & 1 & 0 & 1 & 1 & 0 & 1 & = \\
 \hline
 & & & & & & & & 1 & 0 & 1 & 0 & 1 & 0 & \\
 & & & & & & 0 & & 0 & 0 & 0 & 0 & 0 & & \\
 & & & & 1 & & 0 & & 1 & 0 & 1 & 0 & & & \\
 & & 1 & & 0 & & 1 & & 0 & 1 & 0 & & & & \\
 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & & & & & \\
 1 & 0 & 1 & 0 & 1 & 0 & & & & & & & & & \\
 \hline
 0 & \mathbf{1} & 1 & 1 & 0 & 1 & 1 & \mathbf{0} & 0 & 0 & 1 & 0 & + & & \\
 & & & & & & & 1 & & & & & = & & \\
 \hline
 1 & 1 & 1 & 0 & 1 & 1 & | & \mathbf{1} & & & & & & &
 \end{array}$$

Table 1.3: Example 1.

$$\begin{array}{cccccccccccccccc}
 & & & & & & & & & & 1 & 0 & 1 & 0 & 1 & 0 & \times \\
 & & & & & & & & & & 1 & 1 & 1 & 0 & 0 & 0 & = \\
 \hline
 & & & & & & & & & & 0 & 0 & 0 & 0 & 0 & 0 & \\
 & & & & & & & & & 0 & 0 & 0 & 0 & 0 & 0 & & \\
 & & & & & & & & 0 & 0 & 0 & 0 & 0 & 0 & & & \\
 & & & & & & 1 & 0 & 1 & 0 & 1 & 0 & & & & & \\
 & & & & 1 & 0 & 1 & 0 & 1 & 0 & & & & & & & \\
 & & 1 & 0 & 1 & 0 & 1 & 0 & & & & & & & & & \\
 \hline
 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & + \\
 \hline
 & & & & & & & 1 & & & & & & & & & = \\
 \hline
 1 & 0 & 0 & 1 & 0 & 1 & | & \mathbf{0} & & & & & & & & &
 \end{array}$$

Table 1.4: Example 2.

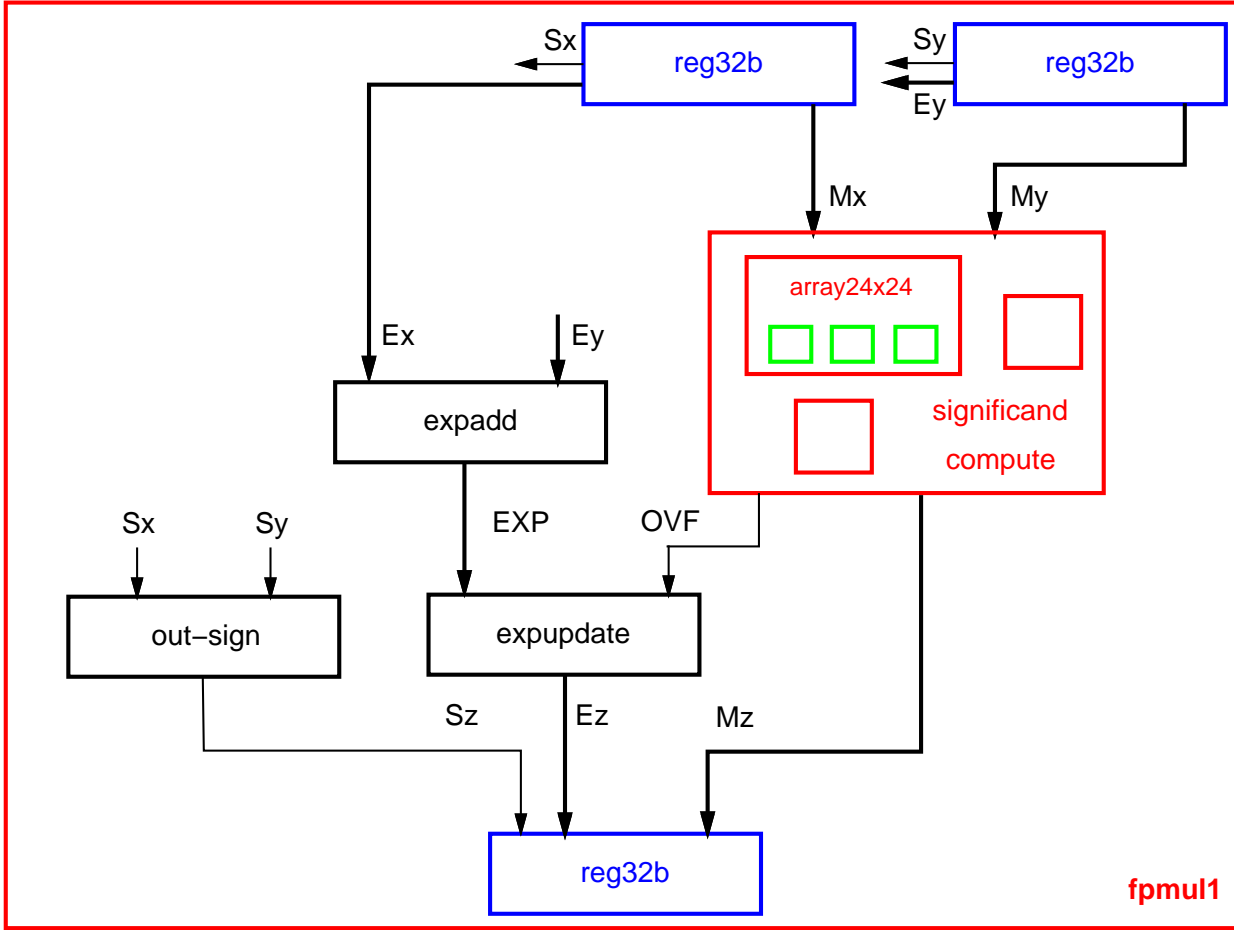


Figure 1.2: Block schematic of FP-mult.

1.3 FP-mult block diagram

The block diagram in Figure 1.2 depicts the structure of the unit. The top-level structural RTL-level VHDL netlist of the unit is in file `fpmul1.vhd`. The unit consists of the main blocks:

expadd computes the sum $E_x + E_y$ and subtracts the bias B . The output of the block is the exponent of the product, if normalized.

significand compute computes the multiplication of the two significands including normalization and rounding. The outputs are the normalized and rounded significand M_z and the signal OVF which indicates if the exponent has to be updated due to normalization.

expupdate updates the value of E_z if the product is not normalized (increment exponent by 1).

out_sign computes S_z . This is simply the exclusive-or of S_x and S_y .

The unit is completed by input and output registers which holds the operands and the result of the floating-point operation.

The VHDL test bench file `tb_fpmul1.vhd` and a set of test vectors to verify the unit functionality and to estimate the power dissipation (`testvecs.in`) are also provided.