Double Precision Floating-Point Multiplier using Coarse-Grain Units

Rui Duarte
INESC-ID/IST/UTL.
rduarte@prosys.inesc-id.pt

Mário Véstias
INESC-ID/ISEL/IPL.
mvestias@deetc.isel.ipl.pt

Horácio Neto
INESC-ID/IST/UTL
hcn@inesc-id.pt

Abstract

This paper describes the implementation of an algorithm to compute the product of two double precision floating-point operands. The algorithm relies on the optimization of the significands multiplication using coarse-grain multiply-accumulate (MACC) units. This work does not detail floating-point pre-and post-processing because it does not benefit from the new coarse-grain units. The algorithm was implemented using DSP48 multiply-accumulate blocks available in Virtex-4 FPGAs.

1. Introduction

This paper describes an algorithm to compute the multiplication of two significands of double precision floating-point numbers using built-in coarse-grain units.

The main motivation to use built-in coarse-grain units is their higher performance when compared to arithmetic units using fine grain elements. However, since these coarse-grain units have a fixed limited size, a few of them have to be used to support bigger operands. Multiplication of large operands can be accomplished by splitting the operands into smaller blocks, according to the maximum size admitted by the coarse-grain multiply-accumulate units, and then bus joining the partial results in the appropriate way to obtain the final result.

A partial products multiplication optimization method to multiply the significands is presented by Knuth [1], as program M for the MIX processor. Each operand is divided into a group of integers ($a_{m-1}...a_0$ and $b_{n-1}...b_0$) and the result is the accumulation of all partial multiplications. If one of the operands in a partial multiplication is equal to zero then that multiplication and its accumulation are skipped.

The architecture proposed in this paper performs better than the Knuth's approach, and exploits coarse-grain FPGAs to use the built-in multiply-accumulate elements in the partial products multiplication algorithm. It is also designed to perform better than "traditional" implementations, and other commercially available IP cores.

In the following sections, we describe the IEEE-754 standard and the typical architecture for a floating point multiplier. Then, the optimized significands multiplication is described followed by a set of results.

2. IEEE-754 Standard

The intention of the IEEE Standard for Binary Floating-Point Arithmetic [5], also known as IEEE-754, is to specify floating-point formats, arithmetic, conversions and exceptions.

Floating-point numbers have three components: sign (±), significand (s) and exponent (e), generically given by equation 1:

$$x = \pm s \times 2^e \tag{1}$$

The sign is represented with one bit indicating whether the number is positive (0) or negative (1). The significand represents the fractional part of the number, in a fixed-point format. The exponent indicates the magnitude of the number, and has an embedded biased value.

The number of bits of the significand determines its precision. In the IEEE standard, the significand is a real number in the interval [1; 2[. The integer component, value 1 is not included, so the significand contains the fractional part of the floating-point value.

The standard defines two formats: single precision (32 bits) and double precision (64 bits). In this work only the double precision format is considered. This format uses 11 bits to represent the exponent and 52 bits + 1 hidden bit for the significand, as shown in figure 1.

0	1 ... 11	12 ... 63
$\pm$	$e + bias$	s

Figure 1. ANSI/IEEE-754 standard floating-point number representation format.

Table 1 summarizes the features of IEEE-754 double precision floating-point number format.

Width (bits)	:	64
Significand (bits)	:	52 + 1 hidden
Significand Range	:	$[1, 2-2^{-52}]$
Exponent Bits	:	11
Exponent Range	:	$[-1022, 1023]$
Exponent Bias	:	1023
Normal Number	:	$1.f \times 2^e$
Denormal Number	:	$0.f \times 2^{-1022}$
Min Absolute Value	:	2^{-1022}
Max Absolute Value	:	$2^{1024} - 1$

Table 1. Floating-point double precision representation.

Besides normal floating-point representation there can also be represented denormalized numbers. They allow representation for very small numbers that otherwise would be rounded to zero. Since it changes the normal representation, it introduces an additional computational effort and cost to support this optional feature, so it is not implemented in this work.

3. Architecture of a Floating-Point Multiplier

As described in [3], given two floating-point numbers n_1 and n_2 having signs $sign_1$ and $sign_2$, exponents e_1 and e_2, and significands s_1 and s_2 respectively, their product is computed as follows:

$$\begin{aligned} sign &= sign_1 \; xor \; sign_2 && (2)\\ s &= s_1 \times s_2 && (3)\\ e &= e_1 + e_2 && (4) \end{aligned}$$

The result belongs to the interval:

$$1 \leq s \leq (2 - ulp)^2,$$

where ulp is the value of the least significant position.

0	1 — 11	12 — 63	64
s	Exponent	Significand	R

Figure 2. Result representation format including extra round bit.

Actually, the result of a multiplication of two numbers $n1, n2 \in [1;2[$, belongs to $[1;4[$. It means that the significand of the result may have to be shifted right one bit, for normalization, and therefore information may be lost.

To minimize numerical errors due to truncation of the result an extra bit is included on the right side of the least significant bit, known as the round bit (R) (See figure 2). If this bit is 0 then truncation is performed, else the number is rounded-up.

Round-to-nearest-even cannot be implemented according to the standard because it requires to compute all the least significant bits. It happens because the proposed algorithm optimizations skips the computation of the least significant bits.

Figure 3 represents the data flow for the floating-point multiplication operation. The two operands of floating-point multiplication are unpacked, processed and the final result is packed into the appropriate IEEE format.

Unpacking (or pre-processing) involves:

1. Separating the sign, the exponent, and the significand for each operand and reinstating the hidden 1;

2. Converting the operands to the internal format;

3. Testing for special operands and exceptions (e.g, recognizing NaN and Infinity.

Packing (or post-processing) involves:

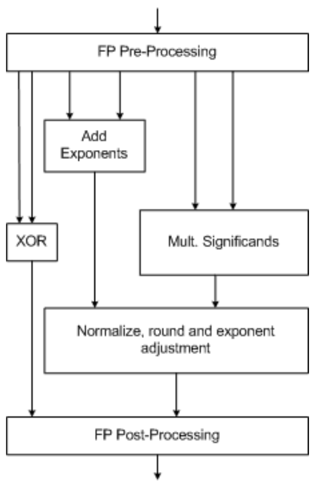

Figure 3. Data flow for floating-point multiplication.

1. Combining the sign, the exponent and the significand and removing the hidden 1;

2. Testing for special values and error conditions.

4. Partial Products Multiplication

DSP48 [2] slices compute a multiplication of 18 bits two's complement operands, or 17 bits unsigned operands, followed by an adder/subtracter. It is possible to combine several DSP48 slices and create a larger arithmetic unit. The unit has internal pipeline registers and it does each multiplication in one clock cycle.

Equation 5 defines operand A as a composition of smaller A_i operands, according to their dimension. N and n are the number of bits for A and A_i respectively. The number of A_i operands, m, is given by equation 6.

$$\begin{aligned} A &= \sum_{i=0}^{m} A_i.2^{n.i} && (5)\\ m &= \left\lceil \frac{N}{n} \right\rceil && (6) \end{aligned}$$

Since floating-point standard operands have 53 bits and DSP48 have 17 bits inputs, A is equal to $A_3.2^{51} + A_2.2^{34} + A_1.2^{17} + A_0$.

Treating each group of 17 bits as a digit, the 53×53 multiplication can be formulated according to its distributive propriety. A 53×53 bits multiplication turns into 16 multiplications of 17 bits each. Multiplication of $A \times B$ is now a multi-word multiplication:

$$A \times B = \left(\sum_{i=0}^{m} A_i.2^{n.i} \right) \times \left(\sum_{j=0}^{m} A_j.2^{n.j} \right) \qquad (7)$$

$$\begin{aligned} A \times B = \; & \left(A_3.2^{51} + A_2.2^{34} + A_1.2^{17} + A_0\right) \times \\ & \times \left(B_3.2^{51} + B_2.2^{34} + B_1.2^{17} + B_0\right) \end{aligned} \qquad (8)$$

The number of required multipliers to produce the result, same as the pipeline depth, is equal to the number of partial multiplications and it is given by equation 9.

				A_3	A_2	A_1	A_0
			$\times$	B_3	B_2	B_1	B_0
				$A_3.B_0$	$A_2.B_0$	$A_1.B_0$	$A_0.B_0$
			$A_3.B_1$	$A_2.B_1$	$A_1.B_1$	$A_0.B_1$	
		$A_3.B_2$	$A_2.B_2$	$A_1.B_2$	$A_0.B_2$		
+	$A_3.B_3$	$A_2.B_3$	$A_1.B_3$	$A_0.B_3$			
	[4]	[17]	[17]	[17]	[17]	[17]	[17]

Figure 4. 53x53 bits Multiplier using 16 DSP48 slices.

$$num\ DSP48 = m^2 \tag{9}$$

The DSP48 blocks have internal registers to reduce the critical path, and they compute one multiplication in each clock cycle. In the particular case of double precision floating-point multiplication (53×53 bits), using a DSP48 (17 bits size operators) it takes 16 clock cycles to compute 16 partial multiplications. In the last partial multiplication, for the most significant bits, the DSP48 computes only 2×2 bits, figure 4. When compared to the total DSP48 available in a Virtex-4 FPGA, the quantity of units needed for the multiplier reaches half of the total available on the smallest FPGA of this family.

Other algorithms to reduce the number of partial multiplications, namely Karatsuba and Cook-Toom, are not favorable to implementation using DSP48 slices built-in the FPGA. They minimize the number of multiplications but increase the number of additions/subtractions. Detailed description of both can be found in Sec. 4.3.3 of [1].

5. Algorithm Optimization

The optimization described in this paper relies on the rearrangement of the operands and on the elimination of some least significant partial multiplications.

In the new scenario the MSb of the operands are placed in the MSb of the last partial multiplication: $A = \left(A_3.2^{36} + A_2.2^{19} + A_1.2^2 + A_0\right)$. A_0 has 2 bits, $A_1..A_3$ have 17 bits.

The product is now computed according to equation 10, $A \times B$ is equal to:

$$\begin{aligned} A \times B &= \left(A_3.2^{36} + A_2.2^{19} + A_1.2^2 + A_0\right) \times \\ &\quad \times \left(B_3.2^{36} + B_2.2^{19} + B_1.2^2 + B_0\right) \end{aligned} \tag{10}$$

Using the distributive propriety to develop the product we have:

$$\begin{aligned} A \times B &= (A_3.B_3)\,2^{72} + (A_2.B_3 + A_3.B_2)\,2^{55} + \\ &+ (A_3.B_1 + A_1.B_3 + A_2.B_2)\,2^{38} + \\ &+ (A_3.B_0 + A_0.B_3)\,2^{36} + \\ &+ (A_1.B_2 + A_2.B_1)\,2^{21} + \\ &+ (A_2.B_0 + A_0.B_2)\,2^{19} + (A_1.B_1)\,2^{4} + \\ &+ (A_1.B_0 + A_0.B_1)\,2^{2} + A_0.B_0 \end{aligned} \tag{11}$$

Multiplying two operands with 53 bits generates a result with 106 bits, where only 53 bits are significant. It means that half of the information is disused, more exactly 52 bits $(106 - 54)$, because only the most significant 54 bits will be kept in the final result.

Ignored information is located at the least significant bits, which leads to the concept of skipping some partial multiplications since they don't take meaning to the final result.

This multiplication unit receives two operands with 53 bits of size and returns a number with 54 bits. Figure 5 shows the organization of partial multiplications, like "pencil and paper", for the 53×53 bit multiplier using DSP48 to produce multiplications. The multiplications in bold are done outside the DSP48 blocks with FPGA LUTs. The stroked out elements are suppressed and don't contribute to the final result. The bottom line has the number of significant bits from each group of partial multiplications.

				A_3	A_2	A_1	A_0
			$\times$	B_3	B_2	B_1	B_0
				$A_3.B_0$	~~$A_2.B_0$~~	~~$A_1.B_0$~~	~~$A_0.B_0$~~
			$A_3.B_1$	$A_2.B_1$	~~$A_1.B_1$~~	~~$A_0.B_1$~~	
		$A_3.B_2$	$A_2.B_2$	$A_1.B_2$	~~$A_0.B_2$~~		
+	$A_3.B_3$	$A_2.B_3$	$A_1.B_3$	$A_0.B_3$			
	[17]	[17]	[17]	[2+R]	-	-	-

Figure 5. 53x53 bits Multiplier using 8 DSP48 slices.

The error introduced by the elimination of the least significant partial multiplications is given by E_A, in equation 12. Its maximum value is less than 2^{39}, it means that the maximum representation error is 2^{-65}. This error is negative, or introduced by defect, because it is caused by the lack of information in the carry bits of the least significant partial multiplications.

$$\begin{aligned} E_A &= (A_2.B_0 + A_0.B_2)\,2^{19} + (A_1.B_1)\,2^{4} + \\ &+ (A_1.B_0 + A_0.B_1)\,2^{2} + A_0.B_0 \end{aligned} \tag{12}$$

This is particulary advantageous if other floating-point operations are to be done consecutively in pipeline because the error introduced into the "real result" is less than the error introduced by any standard rounding scheme. The cost for such precision is to have a data path with 68 bits.

If an application or algorithm, as in [4], admits errors larger than 2^{-65} then the value of $A_0 \times B_3 + A_3 \times B_0$, made of FPGA LUTs, as illustrated in figure 6, can be replaced by a smaller multiplier using less most significant bits of A_3 and B_3.

Using only the 4 most significant bits of A_3 and B_3, the error becomes 2^{-54}, which is still lower than the 2^{-53} error implied by the IEEE-754 double precision representation.

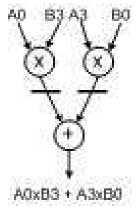

Figure 6. Multiplication of the significand's least significant bits.

The final architecture for the multiplier uses 8 DSP48, as shown in figure 7.

In figures 6 and 7 the small bars lines represent pipeline registers. The registers inside the dashed box represent registers inside the DSP48 block.

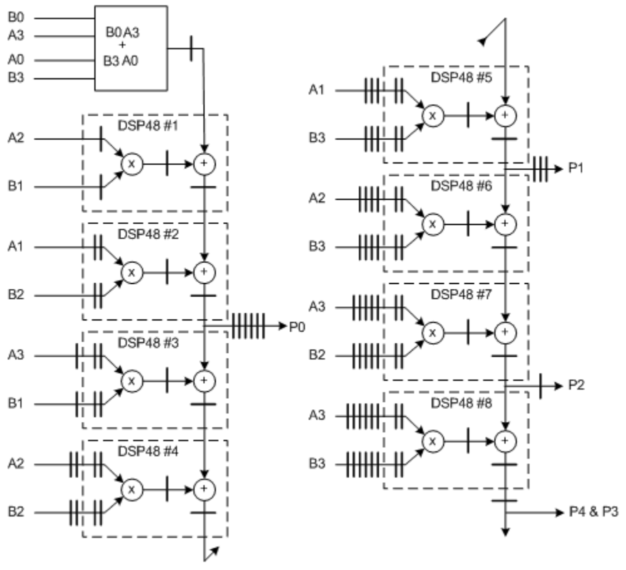

Figure 7. 10 stage pipelined 53×53 bits floating-point significand multiplier.

6. Implementation and Results

The mutliplier was described in VHDL and implemented in a Virtex-4 FPGA from Xilinx, which has built-in DSP48 blocks, each one containing one 18×18 multiplier followed by an adder/subtracter.

The target platform was a Virtex-4 ML402 SX XtremeDSP Evaluation Platform [6]. This platform has one Virtex-4 Xilinx device XC4VSX35-FF668-10C.

The implementation resources for the proposed architecture are summarized in table 2. This unit operates in pipeline, occupies 8 DSP48 blocks and has 10 clock cycles of latency.

Latency	:	10
4 input LUTs	:	190
Logic slices	:	198
DSP48	:	8
Max. frequency [MHz]	:	235.8

Table 2. Resources for the proposed pipelined multiplier unit.

In the following sections are presented the results for other implementations of the same multiplier with other architectures. Their purpose is to show the benefits of DSP48 blocks and the new architecture. For a comparison of devices using different technologies, there should be only considered latency and the number of multipliers/DSP48 blocks.

Since floating-point pre-and-post-processing is the same for any floating-pointer multiplier, the implementation results only differ in the significand manipulation. For this reason, the results presented are only for the significands multiplication.

Implementations of the standard multiplier unit were done, with and without DSP48 blocks. These implementations were done to see how DSP48 influences the resources of the combinational multiplier circuit. The multiplier description was done in VHDL using the "$*$" operator.

From tables 3, 4 and 5 it is possible to see the impact on the number of occupied LUTs when DSP48 blocks are used. The multiplier implementation using DSP48 blocks consumes 8.5% of LUTs and 46% of logic slices of an implementation without DSPs.

Latency	:	1
4 input LUTs	:	1,654
Logic slices	:	1,021
DSP48	:	0
Max. frequency [MHz]	:	47.2

Table 3. Resources for the combinatorial multiplier unit, without DSP48 blocks.

Latency	:	11
4 input LUTs	:	3,223
Logic slices	:	3,556
DSP48	:	0
Max. frequency [MHz]	:	132.0

Table 4. Resources for the pipelined multiplier unit, without DSP48 blocks.

Latency	:	11
4 input LUTs	:	142
Logic slices	:	470
DSP48	:	16
Max. frequency [MHz]	:	279.1

Table 5. Resources for the pipelined multiplier unit, with DSP48 blocks.

Resources occupied by the IP cores from Nallatech [8] and Xilinx Core Generator [7] implementing pipelined multipliers are in tables 6 and 7, respectively.

Nallatech IP core is for Xilinx Virtex-2 FPGAs, which have built-in 18×18 bit multipliers. It requires one more multiplication than our implementation and takes the same number of clock cycles. The proposed multiplier is slower than the Core generator IP core, but takes less cycles and half of the DSP48 units.

7. Conclusions and Future Developments

The main goal of this work was to demonstrate the advantage of built-in MACC units to develop a new optimized floating-point multiplier. High data throughput requirements have been considered by exploiting pipelining.

Latency	:	10
4 input LUTs	:	-
Logic slices	:	693
MACC	:	9
Max. frequency [MHz]	:	160

Table 6. Resources for the pipelined multiplier IP core from Nallatech.

Latency	:	18
4 input LUTs	:	589
Logic slices	:	430
DSP48	:	16
Max. frequency [MHz]	:	350.8

Table 7. Resources for the pipelined multiplier IP core from Core Generator.

Virtex-4 FPGAs family was considered because of their built-in DSP48 MACC blocks.

Compared to the standard multiplier generated using DSP48, our saves 50% of the DSP48 blocks. The number of LUTs are about the same and the number of logic slices are approximately 25%. Even though, it uses only 11% of the LUTs and 19% of the slices used by the multiplier without DSPs.

As for future developments, the architecture will be mapped to the new DSP48E MACC blocks existent in Virtex-5 FPGAs. These DSP48E slices have multipliers capable of computing 25x18 bits.

References

[1] Donald Knuth, *The art of computer programming*, vol. 2, sec. 4.3.1 and 4.3.3.

[2] DSP48 User Guide, Xilinx inc.

[3] Behrooz Parhami, *Computer Arithmetic : Algorithms and Hardware Designs*, Oxford University Press, 2000, New-York.

[4] Rui Duarte, Vitor Silva, Mário Véstias and Horácio Neto, *Double precision floating-point divider using iterative multiplications*, REC - IV Jornadas sobre Sistemas Reconfigurveis, January 2008, Braga.

[5] American National Standards Institute/Institute of Electrical and Electronic Engineers, *IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Std 754-1985, New York, 1985.

[6] *ML402 Evaluation Platform User Manual*, Xilinx, May 2006.

[7] *Logic Core - Multiplier 10.1 - Product Specification*, Xilinx, April 2008.

[8] *Double Precision Floating Point Core*, Nallatech, 2004, `http://www.nallatech.com/`.