# A Multi-Format Floating-Point Multiplier for Power-Efficient Operations

**Nannarelli, Alberto**

*Published in:*
Proceedings of the 30th IEEE International System-on-Chip Conference

*Publication date:*
2017

*Document Version*
Peer reviewed version

# A Multi-Format Floating-Point Multiplier for Power-Efficient Operations

Alberto Nannarelli
Department of Applied Mathematics and Computer Science
Technical University, Denmark
alna@dtu.dk

***Abstract*—In this work, we present a radix-16 multi-format multiplier to multiply 64-bit unsigned integer operands, double-precision and single-precision operands. The multiplier is sectioned in two lanes such that two single-precision multiplications can be computed in parallel. Radix-16 is chosen for the reduced number of partial products and the resulting power savings. The experimental results show that high power efficiency is obtained by issuing two single-precision multiplications per cycle. Moreover, by converting the double-precision numbers which fit to single-precision, further energy can be saved.**

## I. Introduction

Several computing applications require high throughput (parallel) multiplications. Multiplication is a very power demanding operation and increasing its efficiency is highly desirable especially in systems performing several multiplications per cycle in parallel, such as accelerators, multi-lane vector units and GPUs.

Current implementations of parallel binary multiplication X×Y follow the steps of [1]: 1) recoding of the multiplier Y, 2) digit multiplication of each digit by the multiplicand X, resulting in a number of partial products (PPs), 3) reduction of the partial product array to two operands using multioperand addition techniques, and 4) carry-propagate addition of the two operands to obtain the final result.

What makes multiplication power hungry is the large adder tree to reduce the partial products (PPs) to the final product, especially if the operands have a large number of bits [2]. By resorting to a higher radix, the number of PPs can be reduced.

A very popular scheme for parallel multiplication is radix-4, where for a 32-bit multiplier Y, the PPs are 32/2=16 [3]. As the wordlength becomes longer, the radix has been extended to radix-8 [4]–[6] and even radix-16 [7]. The advantage is that the PP reduction tree is shallower (faster and less power demanding) at expenses of a more complicate PP generation. For radix-8, 3×X must be computed and this requires a carry-propagate addition: $3X = 2X + 1$. However, floating-point (FP) multipliers are normally pipelined, and if the latency of the pre-computation can be contained in a single pipeline stage, high radix multipliers become attractive because we can save power.

In this work, we present a radix-16 multiplier supporting both integer and FP multiplication in *binary32* (single precision) and *binary64* (double precision). Moreover, the multiplier can execute two *binary32* multiplications in parallel (dual lane). We chose radix-16 to limit the depth of the PP accumulation tree and save power. The unit was inspired by the implementation in the Intel Itanium FP-unit [7].

The main contributions of this paper are:

- A trade-off analysis of the power dissipation between the radix-4 and radix-16 implementation for a 64×64 multiplier (Sec. II).
- A pipelined implementation of the radix-64 multi-format multiplier with a power analysis for the different formats (Sec. III).
- We propose a simple method to reduce from double to single precision FP numbers to save power (Sec. IV).

## II. Baseline Radix-16 Multiplier

In this section, we briefly describe the architecture of the basic radix-16 multiplier for 64-bit unsigned operands. We denote X the multiplicand operand and Y the multiplier.

The first multiplication step is the recoding of the multiplier operand Y: groups of four bits ($0 \le Y_i \le 15$) are recoded in the minimally redundant radix-16 digit set $\{-8, -7, \ldots, 0, \ldots, 7, 8\}$. This recoding is done carry-free and the transfer-digit (see [1] for detail) is the most-significant bit (MSB) of the four-bit group $Y_i$.

Because of the radix-16 recoding, the number of partial products (PPs) is 16 for a 64-bit Y. However, due to a possible transfer digit from the most significant radix-16 digit, the number of resultant radix-16 recoded digits is $\lceil (n+1)/4 \rceil$ for the general case of a $n$-bit multiplier operand. Therefore, for $n = 64$ the number of PPs is 17. The most significant PP, is either 0 or X shifted 64 bits to the left (16 radix-16, 4 bits, positions). A method to reduce the number of PPs from 17 to 16 is presented in [8].

In parallel with the recoding, the odd multiples of X (3×X, 5×X, and 7×X) are precomputed by three fast carry-propagate adders (CPAs), implementing X+2X=3X, X+4X=5X, and 8X+X=7X, respectively. The values 2X, 4X and 8X are obtained by simple wiring (left shifts) of X. Moreover, 6X is obtained by a simple one bit left shift of 3X.

The PPs are then generated by selecting a multiple of X depending on the value of the radix-16 digit $Y_i$. Fig. 1 illustrates a possible implementation of the PPs generation. An array of XOR gates, complements the bits of the PP when the recoded multiplier digit is negative.

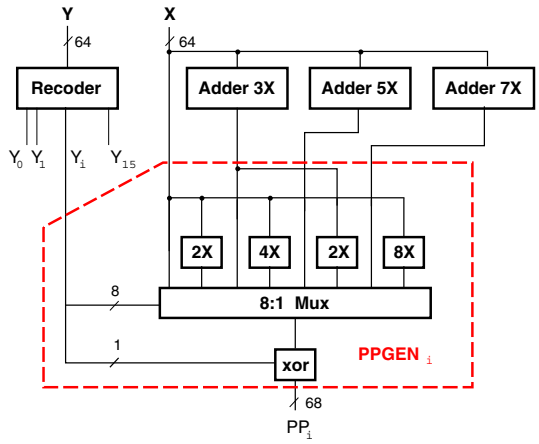


Fig. 1. Partial product generation.

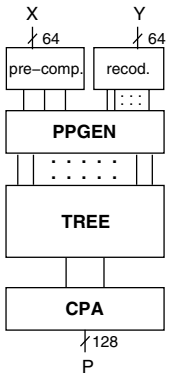


Fig. 2. Radix-16 multiplier.

Each partial product has $n$+4 bits (68 bits) including the sign in two's complement representation. The extra four bits are required for the digit multiplication by up to 8 and the sign bit.

The costly sign extension in the PP array is avoided by the standard method of sign extension reduction and correction [1].

After the generation of the partial product bit array, the reduction (multioperand addition) from a maximum height of 17 (for $n = 64$) to 2 is implemented by 3:2 or 4:2 carry-save adders (CSAs).

After the reduction to two operands, a carry-propagate addition is performed. The complete block diagram for radix-16 multiplication is depicted in Fig. 2.

The baseline 64×64 radix-16 multiplier is implemented in a 45 nm CMOS library of standard cells by using commercial synthesis tools. The FO4 delay for this low power library is 64 $ps$ and the area of the NAND-2 gate is 1.06 $\mu m^2$.

Referring to Fig. 1 and Fig. 2, the delay of the critical path is reported in Table I along with the multiplier's area.

| Critical Path | | | | | |
|---|---|---|---|---|---|
| pre-comput. | PPGEN | TREE | CPA | | |
| 578 | 258 | 571 | 445 | = | 1852 $ps$ |

| Latency | | Area | |
|---|---|---|---|
| $[ns]$ | FO4 | $[\mu m^2]$ | NAND2 |
| 1.852 | 29 | 50,562 | 47.8K |

TABLE I
LATENCY, AREA AND CRITICAL PATH OF 64×64 RADIX-16 MULTIPLIER.

| Critical Path | | | | | |
|---|---|---|---|---|---|
| | PPGEN | TREE | CPA | | |
| | 313 | 739 | 454 | = | 1506 $ps$ |

| Latency | | Area | |
|---|---|---|---|
| $[ns]$ | FO4 | $[\mu m^2]$ | NAND2 |
| 1.506 | 23 | 60,204 | 56.9K |

TABLE II
LATENCY, AREA AND CRITICAL PATH OF 64×64 RADIX-4 MULTIPLIER.

### A. Comparison Radix-4 vs. Radix-16 Multiplication

The main problem with high radices is the computation time of the odd multiples (pre-computation in Table I). However, since FP multipliers are normally pipelined, if the pre-computation can be accommodated in a pipeline stage, its delay is not critical.

To confirm that the radix-16 multiplier consumes less energy than the common radix-4 multiplier, we implement a 64×64 radix-4 Booth multiplier.

We do not implement a radix-8 multiplier because it also needs the pre-computation of 3X, but its reduction tree is larger than the radix-16 tree.

The result of the radix-4 implementation are reported in Table II.

By comparing Table I and Table II, as expected, the radix-4 unit is about 20% faster than the radix-16 unit, but, due to the larger tree the radix-4 unit area is about 18% larger.

The power dissipation for the two units is reported in Table III. The first row refers to the combinational implementation, the second row refers to a two-stage pipelined implementation. All power estimates are done at clock frequency 100 MHz. Note that for the pipelined version of the multiplier, both implementations have the same latency of 2 clock cycles.

For both implementations the radix-16 consumes less power than the radix-4. The radix-16 has a longer delay, therefore, in the combinational implementation the power due to glitches is probably larger than in the radix-4 unit. In the pipelined implementation, the paths in the two stages are shallower and the glitching power is reduced. As a result, the power savings in the radix-16 multiplier increase from 6% to 11%.

| | Power $[mW]$ radix-4 | radix-16 | Ratio |
|---|---|---|---|
| Combinational | 12.3 | 11.5 | 0.94 |
| two-stage pipelined | 8.7 | 7.7 | 0.89 |

TABLE III
POWER DISSIPATION AT 100 MHZ FOR RADIX-4 AND RADIX-16 MULTIPLIERS.

| | binary | | | |
|---|---|---|---|---|
| Format | 16 | 32 | 64 | 128 |
| Storage (bits) | 16 | 32 | 64 | 128 |
| Precision $f = 1$ (bits) | 11 | 24 | 53 | 113 |
| Total exponent length (bits) | 5 | 8 | 11 | 15 |
| $E_{max}$ | 15 | 127 | 1023 | 16383 |
| bias | 15 | 127 | 1023 | 16383 |
| Trailing significand $f$ (bits) | 10 | 23 | 52 | 112 |

TABLE IV
BINARY FORMATS IN IEEE 754-2008 [9].

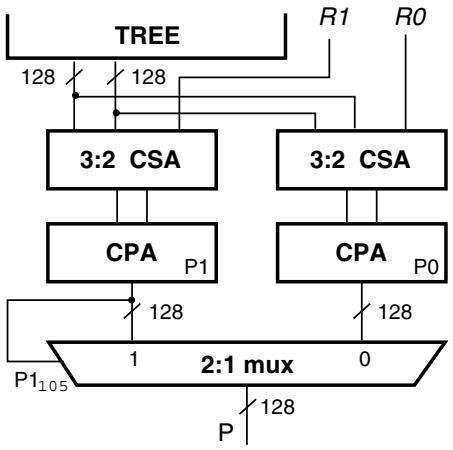


Fig. 3. Detail of the normalize and round operations.

## III. MULTI-FORMAT FP-MULTIPLIER

The starting point to build the Multi-Format FP-multiplier (MFmult) is the radix-16 64×64 multiplier of Sec. II. The MFmult should support the following formats:

- *int64*: multiplication of two 64-bit unsigned integers producing a 128-bit product. This is the operation implemented by the unit of Sec. II.
- *fp64*: multiplication of two floating-point *binary64* numbers (formerly, double-precision), producing a *binary64* result.
- *fp32*: two multiplications of two floating-point *binary32* numbers (single-precision), producing two *binary32* results.

The *int64* format provides a 128 product that can be used for ad-hoc operations in extended precision.

To accommodate the FP multiplications, the unit of Fig. 2 must be augmented by normalization and rounding hardware and sign and exponent handling. The formats for *binary32* and *binary64* are specified in Table IV [9].

### A. Multiplication binary64

The *binary64* FP format requires 64 bits of storage: 1 bit for the sign, 11 bits for the exponent, and 52 bits for the fractional part of the significand. If the biased exponent is larger than zero, the integer bit is '1' and the resulting significand is 53 bits. Therefore, the 53×53 significand multiplication can easily be accommodated in the 64×64 multiplier of Fig. 2 by aligning the operands X and Y to the least-significant bit of 64×64 multiplication.

Once the product is computed, the result may not be normalized [1]. For the multiplication of normalized numbers the leading '1' could be either in bit 105 or in bit 104 (bit 0 is the LSB). Consequently, a 2:1 multiplexer is required to shift the product one position to the left if the leading '1' is in bit 104.

Afterwards, we need to perform the rounding by adding '1' in position 52. By truncating the result in position 53, we obtain the 53 bits $(P_{105} \dots P_{53})$ of the normalized and rounded significand.

Since the rounding step requires a carry-propagate addition, expensive in terms of latency, we can speculatively compute the rounded product for both cases in which the leading '1' is in position 105 or 104, and use the normalization mux to select the normalized product. This is illustrated in Fig. 3. In the figure, $R1$ injects a '1' in position 53 $(0 \dots 001_{53}00 \dots 0_0)$, $R0$ injects a '1' in position 52 $(0 \dots 0001_{52}0 \dots 0_0)$.

This combined normalization and rounding requires an extra fast CPA and extra gates to implement the CSAs[1].

Clearly, for *int64* multiplications, only one CPA is used, $R1$ and $R0$ are set to zero, and the mux-select is set to '0' regardless of bit 105 of P1 (the control for this case is not depicted in Fig. 3).

Currently, the *binary64* multiplier does not support rounding to the nearest in case of a tie (no sticky bit computation) and rounding of *subnormals*.

### B. Two Multiplications binary32

The *binary32* FP format requires 32 bits of storage: 1 bit for the sign, 8 bits for the exponent, and 23 bits for the fractional significand (24 bits for the normalized significand).

We can accommodate two *binary32* multiplications ($XY$ and $WZ$) in the 64×64 multiplier. The operands of the multiplication $XY$ are aligned to bit in position 0 (LSB) and the operands of $WZ$ to bit 32. The arrangement in the PP array is sketched in Fig. 4. The lower part of the array computes $XY$, the upper part $WZ$.

The PPGEN block must be modified to *"blank"* bits of the PP and allow a correct carry-propagation and sign-extension correction for the two independent multiplications. The two CPAs and the mux/shifter of Fig. 3 are divided in an upper and lower part, and the shifter control is separated for the two parts. For example, the upper significand is not shifted while the lower significand is shifted one position to the left.

[1]Since $R1$ and $R0$ contains only one non-zero bit, the CSA is made of one full-adder and 74 half-adders.

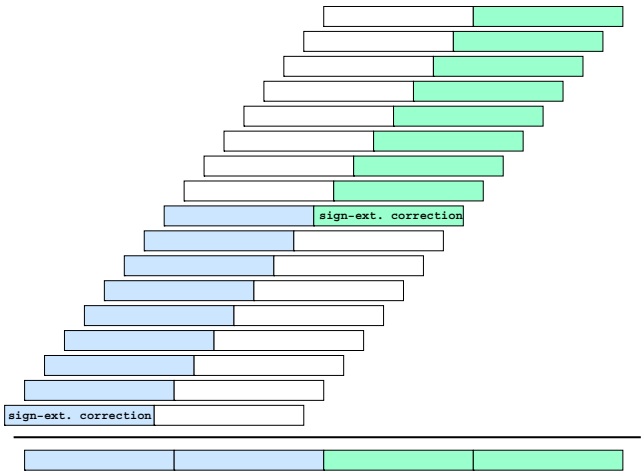


Fig. 4. Array arrangement for two *binary32* multiplications.

Moreover, the injection vectors used for rounding, $R1$ and $R0$, are modified to round in the upper and lower parts of the array:
$R1 = 0\ldots01_{87}00\ldots0_{64}\ldots01_{23}00\ldots0_0$
$R0 = 0\ldots001_{86}0\ldots0_{64}\ldots001_{22}0\ldots0_0$.

### C. Sign and Exponent Handling

Sign and exponent handling (S&EH) is quite straightforward for FP multiplication. For a single multiplication, the sign of the product is the XOR of the sign of the two operands, and the exponent is computed as follows:

1) sum of the two exponents $E_X$, $E_Y$ diminished by the bias $B$: $E_P = E_X + E_Y - B$;
2) $E_P$ is incremented by 1, if the leading '1' of the significand is in bit position 105: $E_P = E_P + 1$.

For dual *binary32* multiplication, the S&EH must be duplicated. The exponent handling of the upper (in the array) multiplication is shared by the *binary64* and the *binary32* operations (11-bit adders), while for the lower *binary32* multiplication a 8-bit datapath is used.

### D. Pipelined Unit

The different components to support the three formats are put together next.

We chose as a target clock period the delay of 16 FO4, corresponding to about 1 ns in our library, to get a throughput of about 1 GFLOPS for *binary64*. With this choice, we pipeline the unit in three stages by placing pipeline stages as indicated in Fig. 5. The figure does not show the replicated hardware for the S&EH of the lower *binary32* multiplication to simplify the drawing. Moreover, control signals to differentiate among the three formats are omitted as well in Fig. 5.

The block **input formatter** in Fig. 5 transfers the operands' bits to the functional units in stage-1 depending on the format specified in *frmt*.

Similarly, the block **output formatter** transfers the result bits to the output. The unit output is split in two 64-bit

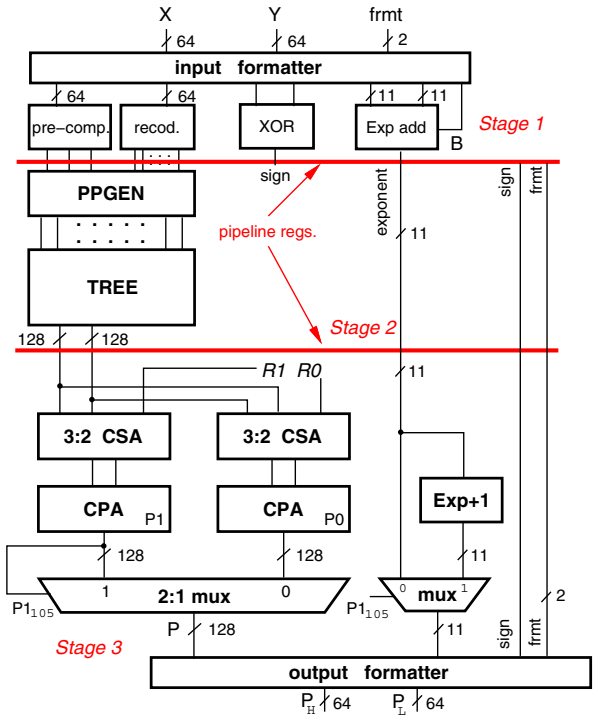


Fig. 5. Pipelined implementation of the multi-format multiplier.

parts $P_H$ and $P_L$. For *int64* multiplications, the 128-bit product is output on both ports. For *binary64* and *binary32* multiplications, the output $P_L$ is not used. For dual *binary32* multiplication, one product is placed in the 32 MSBs of $P_H$, and the other product in the 32 LSBs of $P_H$.

To save time, the exponent is incremented speculatively in stage-3, and then the right exponent is selected once $P1_{105}$ is determined (*binary64*).

For *binary32*, the normalization and the exponent selection is done according to $P1_{111}$ for the upper multiplication, and to $P1_{47}$ for the lower multiplication. This is not depicted in Fig. 5 to keep the drawing as simple as possible.

The pipelined unit is synthesized in the 45 nm library. The resulting critical path of 1120 $ps$ is in stage-2. This corresponds to about 17.5 FO4 delay, 1.5 FO4 in excess of the target cycle time. The pipeline overhead (pipeline register delay and set-up time) is about 3 FO4 (depending on the register loading). According to Table I, the combinational delay in stage-2 (PPGEN+TREE) is about 830 $ps$ or 13 FO4, however, the insertion of the pipeline registers altered the slacks and increased the stage delay.

We tried to move the pipeline registers after the PPGEN, but then the critical path moved in stage-1. We also tried to move the pipeline registers between stage-2 and stage-3, by placing registers inside TREE, but then stage-3 became critical. Since the improvements in the timing are marginal, we settle with the pipeline placement of Fig. 5 which is the one with the lowest number of pipeline registers among the tried placements.

| Format | Power $[mW]$ 100 MHz | 880 MHz | throughput [GFLOPS] | Power eff. [GFLOPS/W] |
|---|---|---|---|---|
| *int64* | 8.90 | 78.32 | 0.88 | 11.24 |
| *binary64* | 7.20 | 63.36 | 0.88 | 13.89 |
| *binary32* (dual) | 5.17 | 45.50 | 1.76 | 38.68 |
| *binary32* (single) | 3.77 | 33.18 | 0.88 | 26.53 |

TABLE V
POWER DISSIPATION AND POWER EFFICIENCY FOR THE DIFFERENT FORMATS.

### E. Power Efficiency

Next we estimate the power dissipation for the different multiplication formats. We perform a Monte Carlo simulation by generating pseudo-random input patterns and estimate the power at a reference frequency 100 MHz to have an easily scalable value to any frequency: our design can be clocked at 880 MHz maximum.

Table V reports the results of the power estimation for the different formats. The throughput is computed as one multiplication per clock cycle for *int64* and *binary64* and two multiplications per cycle for dual *binary32*. Moreover, we report the power dissipation when only one *binary32* operation is issued in the multiplier.

The rightmost column in Table V lists the power efficiency expressed as FLOPS per watt.

The different values of power dissipation in the table are due to different activity in the multiplier. With respect to the *int64*, when a *binary64* multiplication is executed only $\frac{53\times53}{64\times64} = 68\%$ of the bits in the significand datapath are meaningful. The power dissipation ratio *binary64*/*int64* in Table V is about 80%. There is clearly some 10% overhead due to the activity in the S&EH that is inactive for *int64* operations.

As for the power efficiency, the extra efficiency in *binary64* over the *int64* is due to the lower dissipation. For *binary32* we obtain the best efficiency because we complete two multiplications per clock cycle and spend less power than *binary64*. By issuing only one *binary32* multiplication per clock cycle, we double the power efficiency with respect to *binary64* at expenses of a lower precision.

## IV. IMPROVED MULTI-FORMAT FP-MULTIPLIER

The results for power efficiency of Table V suggest to use *binary32* FP format, if the application allows for a reduced precision. This is probably the case for many applications, for example, multiplication of small integers or small fractions.

Since even a single *binary32* is more power efficient than *binary64*, by reducing the precision of operands, we can always save power.

Next, we propose a simple method to convert error-free a *binary64* FP number in a *binary32* FP number, when the non-zero bits of the *binary64* significand can be represented by a *binary32* significand (i.e., number of non-zero bits $\leq 23$), and the range is representable (i.e., unbiased exponent $[-127, 127]$).

The algorithm is shown in Algorithm 1 and its hardware architecture is sketched in Fig. 6.

**Algorithm 1** *binary64* to *binary32* reduction.

```
/* range checking (exponent) */
E_b32 = E_b64 − B_b64 + B_b32 = E_b64 − 896   // must be positive

/* check lower bound (exponent) */
E_b32 − E_max < 0   →   E_b64 − 896 − 255 = E_b64 − 1151 < 0

/* check significand for non-zero bits */
zero = 0;
for i=0 to 28 do
   zero = zero OR significand(i);
end for

if ((E_b32 > 0)AND(E_b64 − 1151 < 0)AND(zero = 0)) then
   reduce to binary32
else
   keep binary64
end if
```

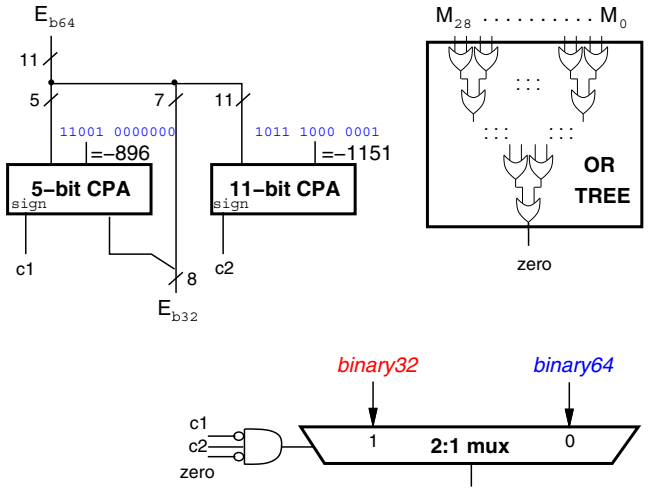


Fig. 6. Hardware for *binary64* to *binary32* reduction.

The reduction of the exponent (first statement in Algorithm 1) can be implemented by a 5-bit adder since the 7 LSBs of -896 are zero. In contrast, to check the lower bound (second statement in Algorithm 1) a 12-bit adder is needed since -1151 is odd.

To check whether the 29 LSBs of the significand (M in Fig. 6) are zero ("for" loop in Algorithm 1), we can use a tree of OR gates. The selection is done with a multiplexer based on the MSB (sign bit) of $E_{b32}$ and $E_{b64} - 1151$, and on the output of the OR-tree. Clearly, the sign bit of the *binary64* is transferred to the *binary32* FP number.

The small hardware of Fig. 6 can be easily included in the multi-format multiplier of Fig. 5. The two short additions can be done in parallel with the speculative exponent computation. Part of the OR-tree can be shared with the sticky-bit computation (not yet implemented in Fig. 5). The selection between *binary32* (reduced) or *binary64* can be easily accommodated in the **output formatter** block.

## V. Conclusions and Future Work

Radix-16 multiplication is an attractive alternative to implement the product of operands with large wordlength because of the reduced PP accumulation tree, which results is smaller silicon area and lower power dissipation.

Another advantage of a reduced number of PPs is that it makes easier the sectioning of the PP array to perform multi-lane operations on operands of reduced wordlength.

We took advantage of the radix-16 scheme to implement a multi-format multiplier. The power analysis on the implemented unit shows that we can increase the power efficiency with respect to *binary64* operations by executing one or two *binary32* multiplications.

Therefore, the proposed unit provides the flexibility of three formats, depending on the application, and power efficiency when the dynamic range of the operands (precision) can be traded-off for power dissipation.

In future work, we plan to extend the reduction *binary64* to *binary32* to periodic numbers as well, and incorporate the reduction in the multi-format unit.

## References


[1] M. Ercegovac and T. Lang, *Digital Arithmetic*. Morgan Kaufmann Publishers, 2004.

[2] S.Galal *et al.*, "FPU Generator for Design Space Exploration," in *Proc. 21st IEEE Symposium on Computer Arithmetic (ARITH)*, Apr. 2013, pp. 25–34.

[3] K. Tsoumanis *et al.*, "An Optimized Modified Booth Recoder for Efficient Design of the Add-Multiply Operator," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 61, no. 4, pp. 1133–1143, Apr. 2014.

[4] E. M. Schwarz, R. M. A. III, and L. J. Sigal, "A radix-8 CMOS S/390 multiplier," in *Proc. 13th IEEE Symposium on Computer Arithmetic (ARITH)*, July 1997, pp. 2–9.

[5] S. Oberman, "Floating point division and square root algorithms and implementation in the AMD-K7 microprocessor," in *Proc. 14th IEEE Symposium on Computer Arithmetic (ARITH)*, Apr. 1999, pp. 106–115.

[6] G. Colon-Bonet and P. Winterrowd, "Multiplier Evolution: A Family of Multiplier VLSI Implementations," *The Computer Journal*, vol. 51, no. 5, pp. 585–594, 2008.

[7] R. Riedlinger *et al.*, "A 32 nm, 3.1 Billion Transistor, 12 Wide Issue Itanium Processor for Mission-Critical Servers," *IEEE Journal of Solid-State Circuits*, vol. 47, no. 1, pp. 177–193, Jan. 2012.

[8] E. Antelo, P. Montuschi, and A. Nannarelli, "Improved 64-bit Radix-16 Booth Multiplier Based on Partial Product Array Height Reduction," *IEEE Transactions on Circuits and Systems I*, vol. 64, no. 2, pp. 409–418, Feb. 2017.

[9] *IEEE Standard for Floating-Point Arithmetic*, IEEE Computer Society Std. 754, 2008.