Tunable Floating-Point for Energy Efficient Accelerators

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

***Abstract*—In this work, we address the design of an on-chip accelerator for Machine Learning and other computation-demanding applications with a Tunable Floating-Point (TFP) precision. The precision can be chosen for a single operation by selecting a specific number of bits for significand and exponent in the floating-point representation. By tuning the precision of a given algorithm to the minimum precision achieving an acceptable target error, we can make the computation more power efficient. We focus on floating-point multiplication, which is the most power demanding arithmetic operation.**

1. Introduction

Nowadays, Machine Learning (ML) is arguably the hottest application field for arithmetic processors. ML algorithms execute a large number of operations, predominantly multiplications, and require dedicated hardware, GPUs or FPGAs, to accelerate the execution.

Due to the huge size of the datasets to process in ML, the processing time and the energy necessary is very large [1]. To increase the power efficiency, the computation is migrated from double-precision (*binary64* in IEEE 754-2008 standard [2]) to single (*binary32*) and half (*binary16*) precision. The precision scaling trend is particularly suitable for FPGA-based accelerators [3]. A big advantage of FPGA-based accelerators is that the hardware can be tailored exactly to match the requirements of the application. For example, the precision required in a given part of the algorithm. In contrast, FPGAs have lower performance and power efficiency than on-chip accelerators.

Previously, we implemented multi-precision multipliers [4] to increase the power efficiency of on-chip accelerators. In this paper, we address the design of an on-chip accelerator with Tunable Floating-Point (TFP) precision. That is, the precision of operands and results can be chosen for a single operation by selecting a specific number of bits for significand and exponent in the floating-point representation.

By tuning the precision of a given algorithm to the minimum precision achieving an acceptable target error, we can make the computation more power efficient. In this work, we present a Tunable Floating-Point multiplier (TFP-mult) which can be used as part of on-chip accelerators. The TFP-mult can handle significand precision from 24 to 4 bits, and exponent from 8 to 5 bits. The maximum precision ($m = 24$, $e = 8$) is that of *binary32* (single-precision), and the tunable range includes *binary16*, or half-precision, ($m = 11$, $e = 5$).

The main contributions of this paper are: 1) The design of the TFP multiplier providing correct rounding when reducing the precision of the significands. 2) The precision of the significand m and the exponent range e can be changed on a cycle basis. 3) To show the power efficiency obtained for sample algorithms and the energy-error trade-offs.

2. Tunable Floating-Point

The floating-point representation of a real number x is

$$x = (-1)^{S_x} \cdot M_x \cdot b^{E_x} \qquad x \in \mathcal{R}$$

where S_x is the sign, M_x is the significand or mantissa (represented by m bits), b is the base ($b = 2$ in the following), and E_x is the exponent (represented by e bits). The representation in the IEEE 745-2008 standard [2] has significand normalized $1.0 \le M_x < 2.0$ and biased exponent: *bias*$= 2^{e-1} - 1$.

The formats for binary floating-point in IEEE 745-2008 are specified in Table 1 [2].

	Binary formats			
Format	16	32	64	128
Storage (bits)	16	32	64	128
Precision m (bits)	11	24	53	113
Total exponent length e (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 1. BINARY FORMATS IN IEEE 754-2008 [2].

The dynamic range[1] for binary floating-point (BFP) is

$$DR_{BFP} = (2^m - 1) \cdot 2^{2^e - 1} .$$

For example, for *binary32* ($m = 24$, $e = 8$)

$$DR_{b32} = (2^{24} - 1) 2^{2^8 - 1} \approx 9.7 \times 10^{83}$$

which is much larger than the dynamic range of the 32-bit fixed-point (FXP) representation: $DR_{FXP} = 2^{32} - 1 \approx 4.3 \times 10^9$ [5].

For the Tunable Floating-Point (TFP) representation, we only consider dynamic ranges from and below the *binary32* representation. We support significand's bit-width from 24 to 4 and exponent's bit-width from 8 to 5. Table 2 shows the dynamic ranges for some TFP cases.

1. The dynamic range is the ratio between the largest and the smallest (non-zero and positive) number [5].

m	e	DR	storage bits*	comment
24	8	9.7×10^{83}	32	*binary32*
11	8	1.2×10^{80}	19	
4	8	8.7×10^{77}	12	
24	5	3.6×10^{16}	29	
11	5	4.4×10^{12}	16	*binary16*
4	5	3.2×10^{10}	9	
16		65,536	16	16-bit FXP

* It includes the sign bit.

TABLE 2. DYNAMIC RANGE FOR SOME TFP FORMATS.

By comparing TFP with FXP ranges, the TFP ones are much larger than the FXP for similar bits of storage. This is advantageous especially for multiplication for which the dynamic range increases quadratically. As for the significand's precision, the optimal bit-width depends on the application.

We assume that the TFP representation is normalized to have the conversions compatible with the IEEE 754-2008 standard. Therefore, the implicit (integer) bit is not stored. Subnormals support is quite expensive for multiplication, and, therefore, we opted to flush-to-zero TFP numbers when the exponent is less than $-(E_{max} - 1)$.
As for the rounding, TFP supports three types:

- **RTZ** Round-to-zero (truncation);
- **RTN** Round-to-the-nearest where half *ulp*[2] is always added before the rounding;
- **RTNE** Round-to-the-nearest-even (on a tie) which is the default mode *roundTiesToEven* in IEEE 754.

3. TFP Multiplier

3.1. Baseline FP Multiplier

The starting point is a generic floating-point multiplier. The architecture of the significand computation is sketched in Figure 1. We assume the FP-operands to be normalized and subnormals are flushed to zero. The implicit (integer) bit is already included in Figure 1.

The radix-4 partial products (PPs) are added in an adder tree to form the product in carry-save format: P_S and P_C. Because of the normalization, the product is in the range $1.0 \leq P < 4.0$ and may require normalization (1 position right shift) if $P \geq 2.0$, i.e., condition `OVF=1`.

As normally done for a faster operation, the rounding is performed speculatively by adding $\frac{ulp}{2}$ in the two adjacent position m and $m-1$ as illustrated in Figure 2. The injection of the rounding bit in a fixed position is done by an array of half-adders (HA) in Figure 1. Therefore, this speculative rounding is implemented by the pairs HA-CPA[3] producing P2 and P1. The normalized and rounded result is selected depending on the position of the leading one in P1 and P2:

$$\texttt{OVF} = P1_{2m-1} \text{ AND } P2_{2m-1} \ .$$

2. *ulp* is the *unit* in the *last position*.
3. CPA: carry-propagate adder.

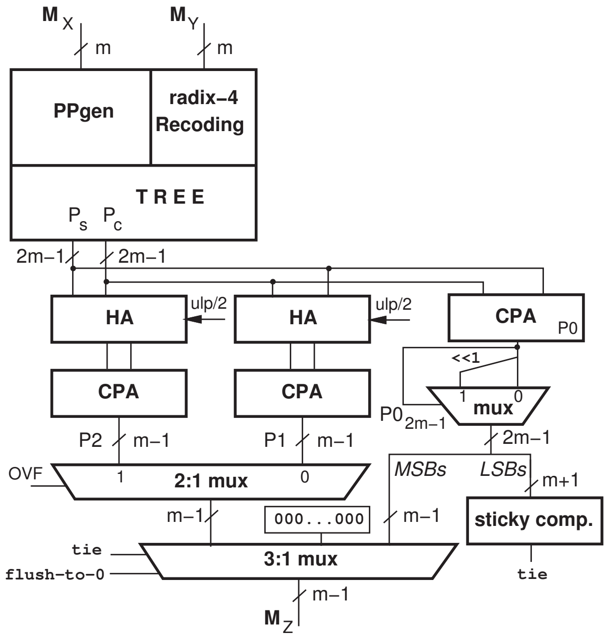

Figure 1. Significand computation in FP multiplier (for *binary32* $m = 24$).

pos.	*2m-1*	*2m-2*	*2m-3*	...	*m+2*	*m+1*	*m*	*m-1*	*m-2*	...	*1*	*0*
P2:	1	b .	b	...	b	L	R	b	b	...	b	b
ulp/2:							**1**					
P1:	0	1 .	b	...	b	b	L	R	b	...	b	b
ulp/2:								**1**				

Figure 2. Speculative rounding for m-bit significands.

To support roundTiesToEven, we need to compute the sticky bit

$$T = \text{OR}_{i=0}^{m-1} b_i$$

For each of the cases in Figure 2, if $L = 0$, $T = 0$ and $R = 1$ the bit to be added for rounding is 0. To include this case of rounding, we need to include a third adder which adds $P_S + P_C$ (CPA P0 in Figure 1).

Moreover, if the biased exponent is 0 (subnormal or zero) or $2^e - 1$ (infinity), we need to flush the significand to zero.

The final selection of the significand is done taking into account the leading one in P0, the `OVF` value, the roundTiesToEven condition `tie` and the exponent flag `flush-to-0`.

The architecture of Figure 1 can be optimized by sharing parts of the datapaths (addition of the $m-1$ least-significant bits, LSBs) and by using compound adders [5], or by many other methods in the literature [6].

The hardware to compute the exponent is sketched in Figure 3. It consists in the exponents addition

$$E_1 = E_x + E_y - \text{Bias}$$

followed by the speculative increment $E_2 = E_1 + 1$ in case the significand needs normalization (`OVF`=1). In parallel

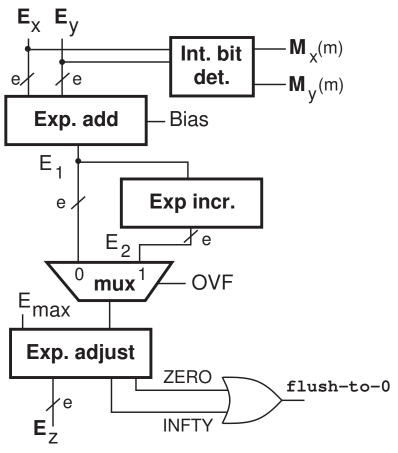

Figure 3. Exponent computation in FP multiplier (for *binary32* $e = 8$).

pos.	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	. . .	25	24	23	22	. . .
P2:	1	b	b	b	b	b	b	b	b	b	L	R	b	b	b	. . .	0	0	0	0	. . .
RW:	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0	0	. . .	0	0			
SB:	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	. . .	-	L	R	b	. . .

Figure 4. Example of "rounding word" (RW), and selection (SB) for sticky-bit computation for $m = 11$.

to the exponent addition, the implicit bit is computed: $M_i(m) = 0$ if $E_i = 0$.

To ensure the exponent is in the allowed range, E_1, or E_2, is checked against 0 and E_{max}. The final exponent E_z is set depending on the signals `INFTY` (infinity) or `ZERO` (zero and subnormals), and, for the significand selection:

$$\texttt{flush-to-0} = \texttt{ZERO} \;\; \text{OR} \;\; \texttt{INFTY}$$

The sign is computed by a simple XOR-gate:

$$S_z = S_x \oplus S_y.$$

3.2. Hardware Support for Tunable FP

In this section, we explain how to augment the unit of Figure 1 and Figure 3 to support TFP.

For the significand computation, the challenge is to inject the rounding bit in position r so that the m resulting bits of the significand are correctly rounded. For example, as illustrated in Figure 4, for the datapath of Figure 1 (the MSB of P2 is 47 for *binary32*), if the required precision is $m = 11$, the rounding bit is in position $r = 36$ if the leading one in P2 is in position 47.

Therefore, to correctly round the TFP significand with arbitrary m, we need to augment the hardware as follows.
1) Add a **decoder** to generate the "rounding word" (RW) depending on m. By setting $m > 24$, the decoder selects RW=0 to implement RTZ (truncation).
2) Change the array of half-adders (HA) in an array of full-adders, or 3:2 Carry-Save-Adder (CSA), to allow the injection of the rounding bit in an arbitrary position depending on m.

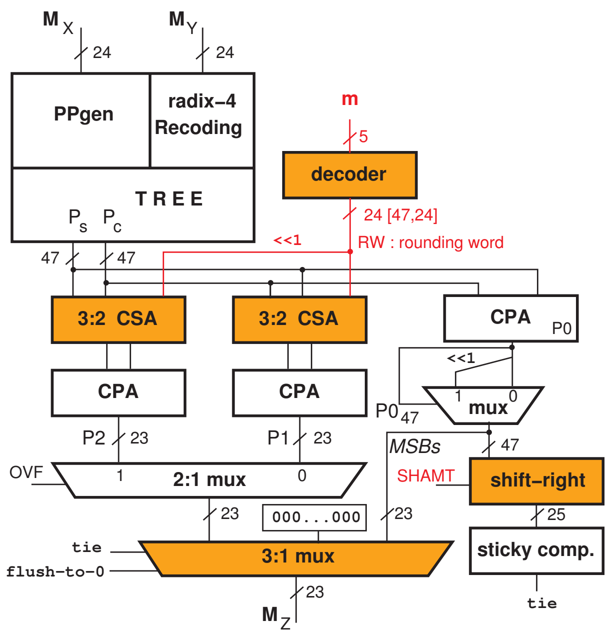

Figure 5. Significand computation to support TFP multiplication.

3) Add a variable **shift-right** unit to select the proper bits for the sticky bit computation. Since the **sticky comp.** unit of Figure 1 reads the $m + 1 = 25$ LSBs for *binary32*, for a different m, we have to shift to the right the significand so that bit L moves to position 24 (MSB) of **sticky comp.**. The shifting amount is `SHAMT`$= 24 - m$. For the example of Figure 4, $m = 11$, L is in position 37, and by shifting right by 24-11=13 positions we obtain the correct selection.
4) Add a mask in the m-MSB of the **3:1 mux** to zero the m LSBs of the product.
This mask can be generated by the decoder. For example for $m = 11$:

```
RW:   0000 0000 0001 0000 0000 0000
MASK: 1111 1111 1110 0000 0000 0000
```

The modified datapath supporting TFP $m = [4, 24]$ for significand computation is depicted in Figure 5. The added or modified parts are in color in the figure.

The exponent computation in TFP is less complicated than the significand one. The range of supported exponents is $e = [5, 8]$ to comprise the *binary16* and *binary32* formats. The bias (B), equal to E_{max}, can be obtained by simple logic as in Table 3 from the 2-LSBs of e:

$$B_6 = \overline{e_1 + e_0}, \quad B_5 = \overline{e_1 \oplus e_0}, \quad B_4 = e_1 + \overline{e_o} \; .$$

The remaining parts of Figure 3 are not changed.

Since the TFP-unit can accommodate up to *binary32*, when $m < 24$ or $e < 8$, the (24-m)-LSBs of the significand are zero and the (8-e)-MSBs of the exponent are meaningless.

The significand and exponent bit-widths m and e can be selected for the single operation by setting a 7-bit value in a control register (not depicted in Figure 5).

	e (bits)				Bias (bits)					
e	e_3	e_2	e_1	e_0	Bias	B_7	B_6	B_5	B_4	$B_3B_2B_1B_0$
5	0	1	0	1	15	0	0	0	0	1111
6	0	1	1	0	31	0	0	0	1	1111
7	0	1	1	1	63	0	0	1	1	1111
8	1	0	0	0	127	0	1	1	1	1111

TABLE 3. TABLE TO GENERATE BIAS.

4. TFP Simulation

In parallel to the hardware implementation of the TFP-unit, it is important to implement a simulator to profile the applications and determine the acceptable precision and dynamic range.

A first version of the simulator consists in a library of C functions implementing TFP operations. The supported operations are: addition/subtraction, multiplication, division and square root.

Each operation is implemented with a standard FP algorithm by limiting the computation of the significand bits to m and by applying the specified rounding mode. However, all operands and results are rendered in double-precision (`double`) in C. Therefore, it is necessary to trap the cases when the unbiased 11-bit exponent exceeds the maximum e-bit exponent generating an infinity exception, and flushing the `double` to zero, when the unbiased 11-bit exponent is less than the minimum e-bit exponent.

TFP multiplication is implemented di the C function serially, one bit at time. TFP division and square-root are implemented in the C functions by a radix-4 digit-recurrence algorithm, two bits per iteration [5].

In the first version of the simulator, the algorithm under test is coded in the C main program by invoking the TFP operations with arbitrary m, e. Each operation may have a different m, e, i.e., precision and dynamic range can be changed in different parts of the algorithm.

The simulator also executes the algorithm under test in `double` and provides the error in key points. Moreover, by setting a debug option, we can display the error for each TFP operation.

Moreover, the simulator generates TFP vectors (either in format *binary32* or *binary16*) to be used to test the hardware implementation.

We briefly describe next the two test algorithms used to verify our TFP multiplier.

4.1. Matrix Multiplication

Matrix multiplication is probably the most common operation in ML algorithms, and common to many other application domains. In consists of multiplications of pairs of elements followed by additions (DOT product). It can be implemented serially or in parallel.

We run experiments on our simulator with fractional elements (input) in $(-1.0,\ 1.0)$ and dynamic range of 24 bits. We used matrices of different sizes. As an example, Figure 6 shows the average error for square 8×8 matrix multiplication under the three rounding modes: RTZ (truncation), RTN (round to the nearest), and RTNE (IEEE 754: roundTiesToEven).

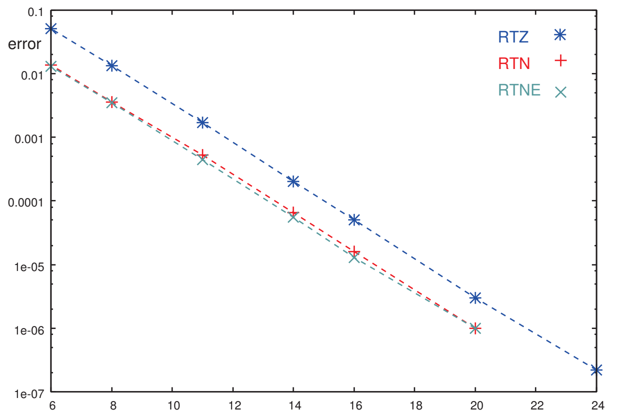

Figure 6. Average error for 8×8 matrix multiplication under TFP rounding modes.

The simulations are run with exponent bit-width $e = 8$ and uniform[4] significands' bit-width $m = \{24, 20, 16, 14, 11, 8, 6\}$. From Figure 6, we notice that RTZ (truncation) leads to a larger error (i.e., factor 3, the scale is logarithmic on y-axis). The error difference for RTN and RTNE is smaller (5–20%). For example, for $m = 11$, $\overline{\epsilon}_{RTN} = 0.532 \cdot 10^{-3} \approx 2^{-11}$ and $\overline{\epsilon}_{RTNE} = 0.442 \cdot 10^{-3} \approx 2^{-12}$. In Sec. 5, we discuss the trade-offs in the hardware implementation of the RTN and the RTNE rounding modes.

Figure 6 shows that for the RTN and RTNE simulations the average error with respect to the double-precision simulation is below 0.1% for $m = 11$. This error is probably acceptable for many applications.

We repeated the simulations for exponent bit-width $e = 5$ and the resulting errors were almost identical. We can conclude that for this range of operands and operations, the error mostly depends on the significand precision.

4.2. Gaussian Elimination

Gaussian elimination is an algorithm used in linear algebra to solve systems of linear equations.
A system with n linear equations in n unknowns x_i

$$\begin{cases} a_{1,1}x_1 + a_{1,2}x_2 + \ldots + a_{1,n}x_n &= b_1 \\ a_{2,1}x_1 + a_{2,2}x_2 + \ldots + a_{2,n}x_n &= b_2 \\ &\vdots \\ a_{n,1}x_1 + a_{n,2}x_2 + \ldots + a_{n,n}x_n &= b_n \end{cases}$$

can be represented by storing $a_{i,j}$ and b_i in a $(n+1) \times n$

4. The value is kept constant for all operations in the algorithm.

Algorithm 1 Gauss elimination.

```
/* rows transformation to upper triangular matrix */
for i=1 to n do
  for j=i+1 to n+1 do
    t=a[j][i]/a[i][i];
    for k=i to n+1 do
      a[j][k]=a[j][k]-t*a[i][k];
    end for
  end for
end for

/* backward substitutions */
x[n]=a[n][n+1]/a[n][n];
for i=n-1 down to 1 do
  s=0;
  for k=n down to i+1 do
    s=s+a[i][k]*x[k];
  end for
  x[i]=(a[i][n+1]-s)/a[i][i];
end for
```

matrix

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,n} & b_1 \\ a_{2,1} & a_{2,2} & \dots & a_{2,n} & b_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{n,1} & a_{n,2} & \dots & a_{n,n} & b_n \end{bmatrix}$$

The Gaussian elimination algorithm consists of two steps:

1) A first pass on the rows of matrix A to transform it in an upper triangular matrix;
2) A backward substitution step in which x_i are computed from the last row and backwards.

The pseudo-code to implement the Gaussian elimination is illustrated in Algorithm 1.

Also for this algorithm we run simulations with fractional inputs in $(-1.0,\ 1.0)$ and dynamic range of 24 bits. As an example, we use a order 5 system (5 unknowns). Figure 7 shows the average error under the three rounding modes: RTZ, RTN, and RTNE. The simulations are run with exponent bit-width $e = 8$ and uniform m with bit-width $m = \{24, 20, 16, 14, 11, 9, 7\}$ for the significands.

For the Gaussian elimination the error for the three rounding modes is similar.

5. Hardware Implementation

For the implementation of the multiplier we opted for a 45 nm CMOS library of standard cells by using commercial synthesis and place-and-route tools (Synopsys). The FO4 delay[5] for this low power library is 64 ps and the area of the NAND-2 gate is 1.06 μm^2.

We set as a target clock period a delay of about 15 FO4 which is comparable to the clock cycle of industrial designs: 17 FO4 in [7]. Therefore, since 15 FO4 $\simeq$ 1.0 ns, the target throughput is 1 GFLOPS for a single TFP multiplier.

5. A 1 FO4 delay is the delay of an inverter of minimum size with a load of four minimum sized inverters.

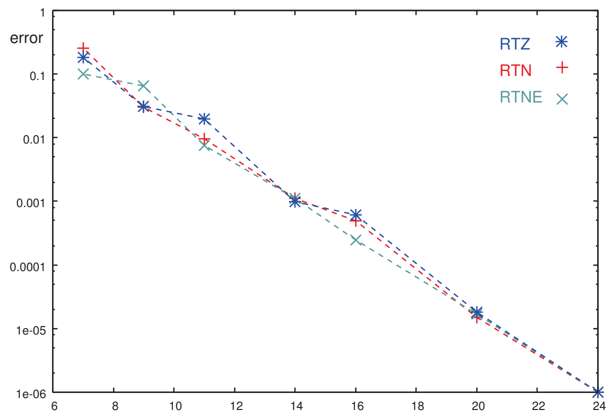

Figure 7. Average error for Gaussian elimination (order 5) under TFP rounding modes.

To reach the target clock period, the architecture of Figure 5 must be pipelined in two stages, with pipeline registers placed after the adder tree (P_S and P_C) for the significand datapath.

We evaluated the implementation of two TFP variants: one with roundTiesToEven (RTNE) rounding mode, the other with round to the nearest (RTN) mode.

The RTN unit, with respect to Figure 5, does not require the blocks CPA (P0), the shifter, and the sticky bit computation. Moreover, the selection logic is simplified. The RTN unit (significand plus exponent datapaths) is shown in Figure 8. The blue horizontal cut indicates the position of the pipeline registers (same placement for both units).

For comparison purposes, we also implemented two plain *binary32* multipliers supporting RTN and RTNE rounding modes. The latter is the unit in Figure 1.

A post-synthesis comparison of the implemented units is reported in Table 4.

For all units, the delay of the critical path is considered the same, since the difference of a few pico-seconds is due to the heuristics of the synthesis. The multiplier array (PPgen+Tree) is the same for all units and sets the minimum cycle time achievable. All units meet the timing constraint of $T_C = 1.0\ ns$.

By comparing the area of the *binary32* and TFP units for the two modes, the overhead is about 16% for RTN and 23% for RTNE. The extra area is due to the additional/augmented blocks represented in color in Figure 5. For example, the shifter contributes to an area of 580 NAND2 (about 4% of the area for the TFP RTNE multiplier), and the CSAs in place of HAs, increase the area of the TFP units by about 250 NAND2 equivalent gates.

The power estimations of the four units stimulated by *binary32* patterns, follow a similar trend than the area, however, the impact of the additional blocks on the power is reduced: about 10% for RTN and 16% for RTNE.

As for the TFP units, the area in the functional blocks of RTNE is about 20% larger than in the RTN unit. This overhead is reduced to about +17% when the input and

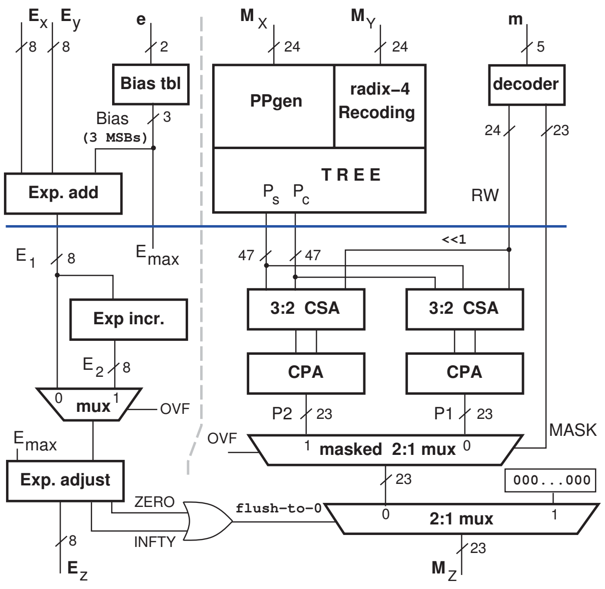

Figure 8. TFP multiplier (RTN variant).

pipeline registers are accounted for. The power overhead for the RTNE unit is about 14%.

The TFP multiplier is intended to be used in accelerators as an element of a multi-lane vector unit. Although the overheads are not very large, for a given area or power footprint we can fit more RTN units. For example, if the power budget is 150 mW we can fit 8 RTN-mults vs. 7 RTNE-mults. Moreover, the error analysis in Sec. 4 suggests that the difference between the RTN and RTNE rounding modes is not critical. For these reasons, we selected the RTN unit of Figure 8 for the evaluation of the power efficiency by TFP.

To have a more accurate evaluation, we built the layout for a single TFP RTN-mult of Figure 8. The post-layout timing and area resulted to be better than the estimates of Table 4: the delay of the critical path changed to 943 ps (almost identical), but the total area is reduced to 10,930 NAND2 equivalent gates, about 15% less than the post-synthesis estimate. This reduction has a positive effect on the power dissipation as explained next.

For the power dissipation, we created traces from the simulator by extracting the actual operands multiplied in the algorithm, and we run post-layout simulations (Synopsys VCS) for several test cases.

5.1. Power dissipation: *binary32* vs. TFP multiplier

We also built the layout for a *binary32* multiplier supporting RTN, referenced as b32-mult in the following. The actual area of the unit is also reduced with respect to the post-synthesis evaluation, 9,790 NAND2, and the TFP area overhead is reduced to about 11% (16% in Table 4).

Since the TFP RTN-mult is obtained by adding functional blocks to the *binary32* multiplier, clearly, multiplications of the same operands will result in higher power dissipation for the TFP unit. However, the TFP-mult produces results of arbitrary precision (m, e) correctly rounded, while the b32-mult produces *binary32* results. Consequently, to save power when a reduced precision is sufficient, *binary32* results need to be processed (rounding of m-bit significands and exponent adjustment) adding latency to the algorithm execution and increasing the power dissipation.

	RTN		RTNE		TFP
	binary32	TFP	*binary32*	TFP	difference
max. delay [ps]	950	944	945	948	-
AREA$^{(1)}$					
comb.	9,560	11,540		13,760	+20%
regs.	1,180	1,320	1,180	1,320	–
Total	10,740	12,860 +16%	11,690	15,080 +23%	+17%
Total power$^{(2)}$	17.04	18.98 +10%	18.13	21.57 +16%	+14%

$^{(1)}$[NAND2 equiv.], $^{(2)}$[mW] at 1 GHz

TABLE 4. POST SYNTHESIS COMPARISON BETWEEN *binary32* AND TFP MULTIPLIERS FOR RTN AND RTNE MODES.

Next, we illustrate an example to show the TFP unit be more power efficient than the *binary32* one, for multiple operations in reduced significand precision ($m < 24$). The experiment consist in the following two cases:

1) Multiplication $Z = X \times Y$ for X and Y with $e = 8$ and precisions $m = \{24, 20, 16, 14, 11, 8, 6\}$.
2) Multiplications (two) $Z = (X \times Y) \times W$ for X, Y and W with $e = 8$ and precisions $m = \{24, 20, 16, 14, 11, 8, 6\}$. The intermediate product $X \times Y$ is the m-bit significand output for the TFP-mult and a *binary32* output for the *binary32* multiplier.

The results for the total average power dissipation are reported in Table 5 and the trends are shown in Figure 9.

The performance of the TFP-mult is almost the same for the two test cases, the plots are almost completely overlapped in Figure 9 (squares).

Since the TFP-mult is an augmented b32-mult unit, when both operands are in the same format (*binary32* for $m = 24$, or TFP-format for $m < 24$), for case 1, the TFP-mult consumes on the average 9% more than the b32-mult. However, while the TFP-mult produces correctly rounded significands in m-bit TFP-format, the b32-mult produces significands in full *binary32* range ($m = 24$).

The performance of the b32-mult changes drastically for reduced significand precision when only one operand is in TFP-format (case 2). In this, case for precisions lower than $m = 20$, the TFP-mult is more power efficient than the b32-mult. This is illustrated by the cross-over of the red/blue curves in Figure 9 at $m \simeq 18$, and by the ratios < 1.0 in Table 5.

P_{ave}		$Z = X \times Y$			$Z = (X \times Y) \times W$	
m	TFP-mult	b32-mult	ratio	TFP-mult	b32-mult	ratio
24	15.51	14.24	1.09	15.72	14.42	1.09
20	14.19	12.89	1.10	14.26	13.63	1.05
16	11.14	10.26	1.09	11.10	11.86	**0.94**
14	10.17	9.42	1.08	10.09	11.26	**0.90**
11	8.58	7.91	1.08	8.54	10.28	**0.83**
8	6.05	5.67	1.07	5.98	8.52	**0.70**
6	5.08	4.72	1.08	5.07	7.36	**0.69**

P_{ave} [mW] measured at 1 GHz.

TABLE 5. AVERAGE POWER DISSIPATION FOR TFP-MULT AND B32-MULT (RTN).

	Matrix multiplication		Gauss elimination	
m	P_{ave} [mW]	ratio	P_{ave} [mW]	ratio
24	15.51	1.00	13.85	1.00
20	14.19	0.91	12.11	0.87
16	11.14	0.72	9.80	0.71
14	10.17	0.66	8.69	0.63
11	8.58	0.55	7.93	0.57
9			6.24	0.45
8	6.05	0.39		
6	5.08	0.33		

P_{ave} measured at 1 GHz.

TABLE 6. AVERAGE POWER DISSIPATION FOR TFP-MULT (RTN) AS m SCALES ($e = 8$).

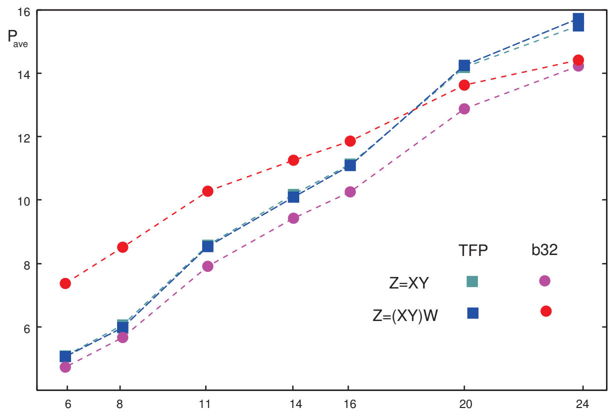

Figure 9. Trends of average power dissipation for comparison TFP-mult vs. b32-mult.

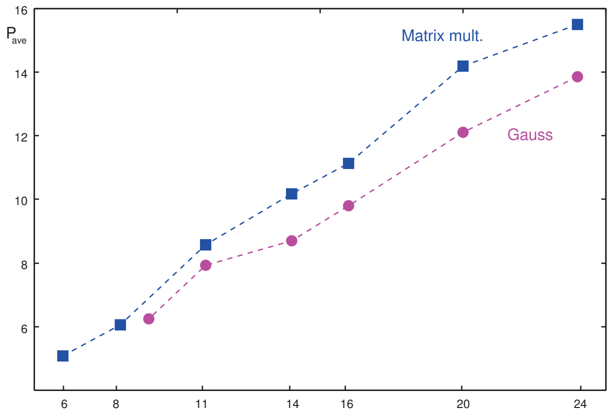

Figure 10. Trends of average power dissipation for TFP-mult (RTN) as m scales ($e = 8$).

5.2. TFP Multiplier Experiments

We sampled the same m and e values used in Sec. 4 for the matrix multiplication and the Gauss elimination algorithm.

By comparing power figures for the same m, but different exponent bit-width e, we noticed a small difference. The reason is that most of the power in a FP multiplier is dissipated in the significand computation. For example by comparing $e = 8$ and $e = 5$ for $m = 11$, the difference in power dissipation in the exponent path is less than 30%. However, the exponent path impacts less than 3% the total power of the TFP-mult with $m = 11$. Therefore, from a power dissipation point of view, going from $e = 8$ to $e = 5$ results in a overall power reduction of less than 1%.

Since the power savings in reducing the exponent bit-width are not significant, and to simplify the presentation of the experimental results, we opted for test patterns with fixed exponent $e = 8$.

For the two algorithms of Sec. 4, we run post-layout power estimation for the following significand bit-widths:

- Matrix multiplication, square 8×8 matrices: $m = \{24, 20, 16, 14, 11, 8, 6\}$.
- Gaussian elimination, systems of order 5: $m = \{24, 20, 16, 14, 11, 9\}$.

With respect to the error plots in Sec. 4, we omitted the case for Gaussian elimination $m = 7$ because the error is too large.

The results for the total average power dissipation are reported in Table 6 and plotted (together with the trends) in Figure 10.

The average power dissipation, estimated post-layout, for $m = 24$, i.e., *binary32* operands, is 15.51 mW at 1 GHz. The smaller value with respect to the post-synthesis estimation, 18.98 mW in Table 4, is due to the denser actual layout than the estimated one. Consequently, the switched capacitance is reduced and so it is the power dissipation.

The trends in Figure 10 show a linear decrease of power dissipation for the matrix multiplication algorithm as m is scaled. Therefore, when the error (lower precision) is acceptable, we can save power by reducing the precision of the algorithm.

For the Gauss elimination case, the curve in Figure 10 is placed below the matrix multiplication's curve because one of the multiplications' operands is the result of a division (`t` in Algorithm 1) and it is multiplied for all the elements of the row. This results in reduced switching activity in the multiplier array, which contributes to the largest part of the power dissipated in the unit. Also in this case, the power dissipation drops linearly as m is scaled.

In summary, the results of Table 6 show that by using TFP multipliers we can achieve a good power efficiency and maintain correct rounding to achieve the target error.

Algorithm 2 *binary32* to *binary16* reduction.

```
/* range checking (exponent) */
$E_{b16} = E_{b32} - B_{b32} + B_{b16} = E_{b32} - 112$   // must be positive

/* check lower bound (exponent) */
$E_{b16} - E_{max(5b)} < 0 \rightarrow E_{b32} - 112 - 31 = E_{b32} - 143 < 0$

/* check significand for non-zero bits */
zero = 0;
for i=0 to 12 do
    zero = zero OR $M_{b32}(i)$;
end for

if $((E_{b16} > 0)\text{AND}(E_{b32} - 143 < 0)\text{AND}(zero = 0))$ then
    reduce to binary16
else
    keep binary32
end if
```

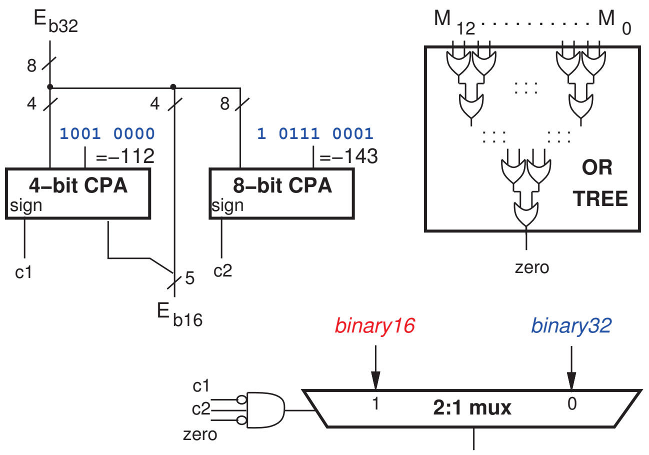

Figure 11. Hardware for *binary32* to *binary16* reduction.

6. Accelerator Interface

The TFP multiplier is intended to be part of an accelerator. When data are moved out to the memory, or some other unit, we can save power in buses by reducing the storage to one of the standard IEEE 754 formats.

Next, we propose a simple method to convert error-free a *binary32* FP number in a *binary16* FP number, when the non-zero bits of the *binary32* significand can be represented by a *binary16* significand (i.e., number of non-zero bits ≤ 10), and the range is representable (i.e., unbiased exponent $[-15, 15]$). This conversion can be useful when the final result of the TFP accelerator can be represented in a smaller format.

The algorithm is shown in Algorithm 2 and its simple hardware architecture is sketched in Figure 11.

The reduction of the exponent (first statement in Algorithm 2) can be implemented by a 4-bit adder since the 4 LSBs of -112=$(1001\ 0000)_2$ are zero. In contrast, to check the lower bound (second statement in Algorithm 2) a 8-bit adder is needed since -143=$(1\ 0011\ 0001)_2$ is odd.

To check whether the 13 LSBs of the significand (M in Figure 11) are zero ("for" loop in Algorithm 2), we can use a tree of OR gates. The selection is done with a multiplexer based on the MSB (sign bit) of E_{b16} and $E_{b32} - 143$, and on the output of the OR-tree. Clearly, the sign bit of the *binary32* is transferred to the *binary16* FP number.

The small hardware of Figure 11 can either be included in the single TFP unit (such as the TFP-mult), or at the accelerator's interface when results are committed to the register file.

7. Conclusions and Future Work

In this paper, we presented a Tunable Floating-Point representation to reduce precision and dynamic range of floating-point numbers when the rounding error produces still acceptable results. The main purpose of TFP is to reduce the energy footprint of accelerators by introducing flexibility in the choice of significand and exponent bit-widths.

The main contribution is the design of a TFP multiplier providing correct rounding for the selected precision, and reduced power dissipation. The TFP-mult can be deployed in an array to form a multi-lane multiplier.

We also designed a TFP simulator for the dual purpose of profiling applications and providing actual traces for the hardware verification and to estimate the power savings.

In presenting the results of the two simple algorithms implemented, we opted for a simple round-to-the-nearest (RTN) unit. However, in future work, we plan to implement a TFP-mult supporting IEEE 754' *roundTiesToEven* (RTNE) and make programmable the rounding mode by disabling the blocks not used in a specific mode. In this way, we can increase the flexibility of the accelerator and keep a good power efficiency.

We also plan to extend TFP to the other operations: addition, division and square root.

References

[1] B. Catanzaro, "Computer Arithmetic in Deep Learning," in *Keynote Talk at the 23rd IEEE Symposium in Computer Arithmetic*, July 2016. [Online]. Available: http://arith23.gforge.inria.fr/slides/Catanzaro.pdf

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

[3] E. Nurvitadhi *et al.*, "Can FPGAs Beat GPUs in Accelerating Next-Generation Deep Neural Networks?" in *Proc. of ACM FPGA'17*, Feb. 2017.

[4] A. Nannarelli, "A multi-format floating-point multiplier for power-efficient operations," in *30th IEEE International System-on-Chip Conference (SOCC)*, Sep. 2017, pp. 351–356.

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

[6] 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.

[7] N. Burgess and C. N. Hinds, "Design of the ARM VFP11 Divide and Square Root Synthesisable Macrocell," *Proc. of 18th IEEE Symposium on Computer Arithmetic*, pp. 87–96, July 2007.