This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2019.2906907, IEEE Transactions on Computers

Tunable Floating-Point Adder

Alberto Nannarelli, *Senior Member, IEEE*

Abstract—In this work, we address the design of an adder in Tunable Floating-Point (TFP) precision. TFP is a variable precision format in which a given 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 an algorithm to the minimum precision achieving an acceptable target error, we can make the computation more power efficient. In previous work, we introduced a unit for TFP multiplication. Here, we focus on floating-point addition.

Index Terms—Floating-point, addition, IEEE rounding, energy efficiency.

1 Introduction

Deep Learning (DL) is currently one of the hottest application fields for arithmetic processors. DL algorithms execute a huge number of operations, predominantly multiplications and additions, and resort to dedicated hardware, GPUs [1] or FPGAs [2], to accelerate the execution.

Due to the size of the datasets in DL, the processing time and the energy necessary is very large [3]. To increase the power efficiency, the computation is migrated from double-precision (*binary64* in IEEE 754-2008 standard [4]) to single (*binary32*) and half (*binary16*) precision. In addition to the IEEE formats, several other formats have been introduced, such as the Google's Brain-FP format [5], or the Intel's "Flexpoint" format [6].

Although floating-point operations are more complex to implement in hardware than fixed-point, the advantage is that the precision scaling is handled by the unit and it is transparent to the programmer (no operand shifting or re-scaling).

In [7], we introduced the Tunable Floating-Point (TFP), a format in which the precision of operands and results can be chosen for a single operation by selecting a specific number of bits for the significand and the 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 [7], we described the implementation of a TFP multiplier. In this work, we present a Tunable Floating-Point adder (TFP-add) to complete, along with the multiplier, a TFP unit which can be used as part of on-chip accelerators. The TFP unit 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* ($m = 11$, $e = 5$) and Google's Brain-FP format ($m = 8$, $e = 8$).

The contributions of this paper are: 1) The revisitation of the "double-path" adder [8] implementing the baseline *binary32* adder. 2) The design of the TFP adder providing correct rounding when reducing the precision of the significands – the main contribution. 3) An example showing the benefits of the Tunable Floating-Point for a simple neural network used to approximate a generic function and the error-energy trade-offs.

• *A. Nannarelli is with the Dept. of Applied Mathematics and Computer Science (DTU Compute), Technical Univ. of Denmark, Lyngby, Denmark. E-mail: alna@dtu.dk*

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 [4] has significand normalized $1.0 \leq M_X < 2.0$ and biased exponent: *bias*$= 2^{e-1} - 1$.

We introduced the Tunable Floating-Point (TFP) representation in [7]. In TFP, we only consider dynamic ranges from and below the *binary32* representation. We support significand's bit-width m from 24 to 4 and exponent's bit-width e from 8 to 5.

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 and, therefore, we opted to flush-to-zero TFP numbers when the exponent is less than the minimum representable in e bits.
TFP supports three types of rounding:

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

For the rounding, we use the terminology of [8] summarized for reference in Fig. 1.

We also developed a TFP simulator [7] consisting in a library of C functions and implementing TFP operations. Each operation is implemented with a standard FP algorithm by limiting the computation of the significand bits to m and applying the specified rounding mode. The simulator

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2019.2906907, IEEE Transactions on Computers

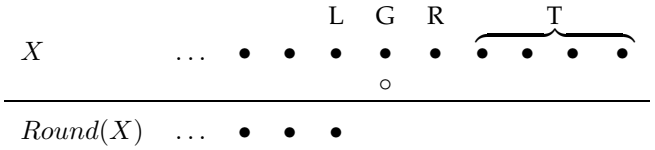

Fig. 1. Rounding. From the left: L last bit; G guard bit; R round bit; T sticky bit; ○ rounding position.

provides the error double-precision vs. TFP in key points of the algorithm and generates test patterns for the hardware verification.

3 Baseline FP Adder

The algorithm for floating-point addition can be summarized in the following steps [8]:

1) Subtract exponents $D = E_X - E_Y$.
2) Align significands M_X, M_Y by shifting right D positions the significand with the smallest exponent.
3) Add, or subtract, the significands and determine the sign of the result.
4) Normalize and round the result.

Among the different implementations of the FP addition algorithm, we select as the baseline FP adder a "double-path" architecture derived from the schemes of [8] and [9]. The architecture is sketched for the significand part in Fig. 2 for *binary32*. The significands M_X, M_Y and M_Z include the integer bit ("1" if the biased exponent is larger than zero). The exponent difference (D) is performed at the beginning of the FP operation to determine the alignment of the two operands and to set which path will produce the correct result.

The CLOSE path is taken when the effective operation is subtraction (EOP=1) and the exponent difference is one (D=1) or zero (D=0). Therefore, the select signal CLS in Fig. 2 is set as:

$$\text{CLS} = \text{EOP} \ \text{AND} \ (\ (\text{D=1}) \ \text{OR} \ (\text{D=0})\ )$$

The FAR path is taken for all additions and subtractions when $D > 1$ (CLS=0).

The main advantage of the double-path implementation is that only one variable shifter is in the critical path so that the FP-addition latency is reduced. The unit can be pipelined in two stages as suggested in [8]. In the following, we explain how this is done for the baseline unit. The architectures in the CLOSE and FAR paths are shown in Fig. 3 and Fig. 4. With respect to the implementation of [8], the unit to swap the operands is not shared but moved in the FAR path to reduce the latency in the CLOSE path.

3.1 CLOSE Path

The CLOSE path is taken when the effective operation is subtraction (EOP=1) and the exponent difference is one (D=1) or zero (D=0). The architecture of the CLOSE path is shown in Fig. 3. We explain next how it works.

Since in the CLOSE path the exponent difference is one or zero, operands alignment can be done by shifting one position to the right the significand of the number with the smallest exponent when D=1. This alignment is done by the

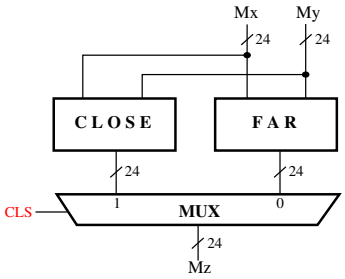

Fig. 2. Double-path architecture (significands only).

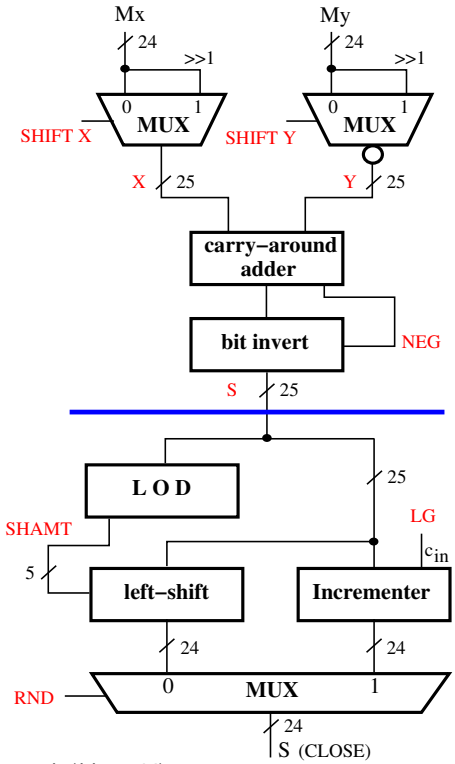

Fig. 3. CLOSE path (*binary32*).

two 2:1 multiplexers "MUX" in Fig. 3. The control signals are set as:

$$\text{SHIFT X} = \overline{sign(D)} \ \text{AND} \ (\text{D=1})$$
$$\text{SHIFT Y} = sign(D) \ \text{AND} \ (\text{D=1})$$

Determining the sign of the result before the actual subtraction is expensive because it requires to compare the two significands when the exponent is the same (D=0) to decide which operand to invert.

An alternative is to complement a fixed operand, for example M_Y, and then complement the subtraction result if negative. Since two's complementation requires a full precision addition, we opted for a one's complement adder, or carry-around adder [8], followed by conditional bit complementation, implemented by an array of XOR gates. These are the blocks indicated as "carry-around adder" and "bit invert" in Fig. 3. The complementation of the bits of M_Y is marked by the circle at the output of the MUX.

Next, a leading-one-detector (LOD) detects the position of the leading one for subtractions producing leading zeros. The position (SHAMT) is then used in the variable left-shifter to normalize the result.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2019.2906907, IEEE Transactions on Computers

In the CLOSE path, rounding is required when D=1 and the result is normalized. In contrast, variable shifting is required when the result is not normalized. Since rounding and shifting are mutually exclusive, they are implemented in parallel paths in Fig. 3. The selection is done by the signal

$$\text{RND} = \text{MSB}(S) \text{ AND } (\text{D=1})$$

For the rounding, a simple incrementer is used. The result is rounded up when the last (L) and the guard (G) bits are both "1": $LG = 1$.

The blue horizontal line in Fig. 3 shows the optimal placement of the pipeline registers in the CLOSE path.

3.2 FAR Path

All additions and subtractions with exponent difference larger than one are executed in the FAR path (Fig. 4).

Due to many positions shifts that may be necessary to align operands and to the fact that the shifter is expensive, one barrel shifter is used. To shift the significand with the smallest exponent in one shifter, it is necessary to swap the operands. This is done by the "swap" and "right-shift" blocks in Fig. 4. The swap is activated when $E_Y > E_X$: $sign(D) = 1$.

The "right-shift" block does preserve the shifted out bits (22 least-significant bits, or LSBs) which are used to compute the sticky bit[1].

The most-significant part, 26 bits including guard (G) and round (R) bits, is conditionally inverted in case of subtraction (EOP=1).

The rounding is combined with the addition to reduce the latency. The sum of F_X and F_Y is performed in a compound adder depicted by the two carry-propagate adders, or CPAs, followed by the MUX in Fig. 4. Block CPA0 in Fig. 4 computes $F_X + F_Y = S0$, CPA1 computes $F_X + F_Y + 1 = S1$. The selection between the two sums is done according to the rounding.

To explain the rounding, we separate between the case of effective operation addition (EOP=0) and effective operation subtraction (EOP=1).

3.2.1 FAR Path Rounding: Addition

When the effective operation is addition, we obtain for the sum $F_X + F_Y$ a rounding pattern similar to the one in Fig. 1.

For the round-to-the-nearest-even mode (*roundTiesToEven*), we round up (select S1) if

$$\text{MUXC(0)}: \quad G(R \vee T \vee L) = 1 \tag{1}$$

where we use "$\vee$" as a compact way for the logic OR to avoid confusion with the "+" used for addition. However, if the sum result is larger than 2.0, we have an overflow (OVF=1) and the result is not normalized. In this case, we need to shift the sum one position to the right[2] (block "shift-1 L or R" in Fig. 4) and perform the rounding as follows. By considering L^*, the bit to the left of bit L, we round up if:

$$\text{MUXC(1)}: \quad L(L^* \vee G \vee R \vee T) = 1\,. \tag{2}$$

That is, the guard bit is L and the last bit is L^*.

1. The sticky bit T is needed to detect a tie. T is computed by ORing the trailing bits of the result beyond bit R (Fig. 1).
2. The exponent need to be incremented by one when OVF=1.

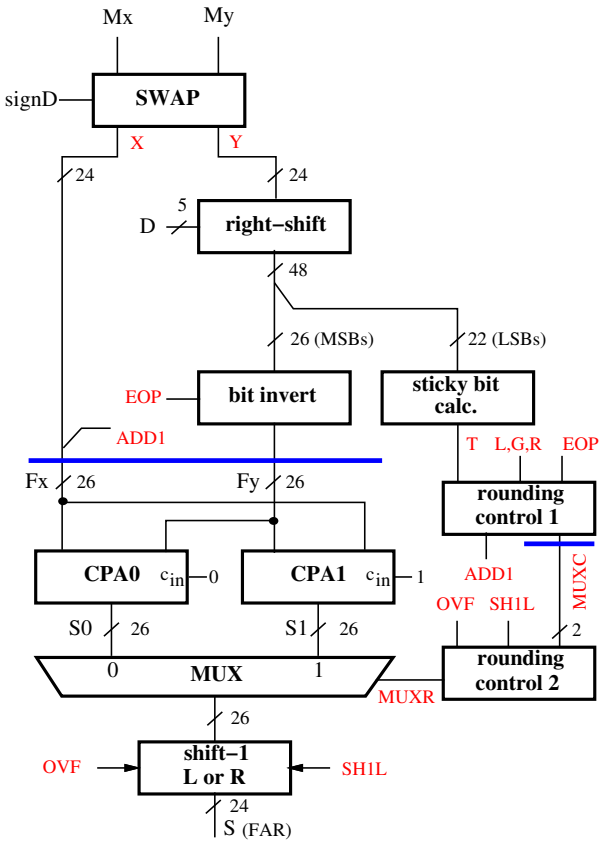

Fig. 4. FAR path (*binary32*).

Expressions (1) and (2) are evaluated in block "rounding control 1" in Fig. 4 from LGR (and L^*) extracted from $F_X + F_Y$ and the sticky bit T. The output is the 2-bit MUXC signal.

By indicating with S_i the bit of weight 2^i of the sum, the overflow detection is done by

$$\text{OVF} = (S1_1 \cdot S0_1) \vee (S1_1 \cdot \text{MUXC(1)} \cdot \text{MUXC(0)})$$

where the second term is used for correction when the round up bit causes overflow.

Based on the overflow, either MUXC(0) or MUXC(1) is selected to determine the result between S0 and S1

$$\text{MUXR} = (\text{OVF} \cdot \text{MUXC(1)}) \vee \overline{\text{OVF}} \cdot \text{MUXC(0)})\,.$$

3.2.2 FAR Path Rounding: Subtraction

When the effective operation is subtraction (EOP=1), the significand with the smallest exponent Y may be shifted and its 26 most-significant bits (MSBs) are one's complemented to form F_Y. To perform two's complement, a "1" must be added in the least-significant position of Y (before the shifting). However the complement's "1" is propagated to the 26 MSBs (F_Y) only if the shifted out bits of Y are all zero. Consequently, we can use the sticky bit T to determine if we need to add the complement's "1" to $-F_Y$. Therefore, the condition to add a "1" to the LSB of $-F_Y$ is

$$\text{ADD1} = \text{EOP AND } \overline{T}\,. \tag{3}$$

Consequently, for EOP=1, ADD1 is the complement of T. Since F_X is a normalized 24-bit significand represented in 26

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2019.2906907, IEEE Transactions on Computers

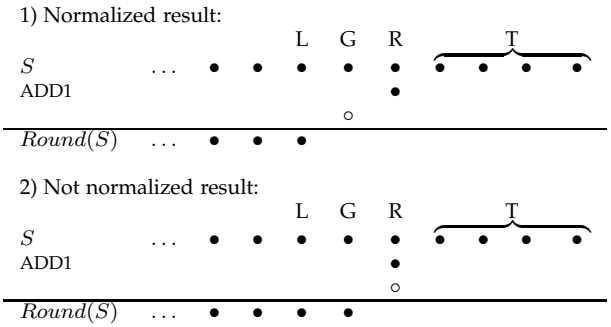

Fig. 5. Rounding for subtraction in the FAR path. ○ rounding position.

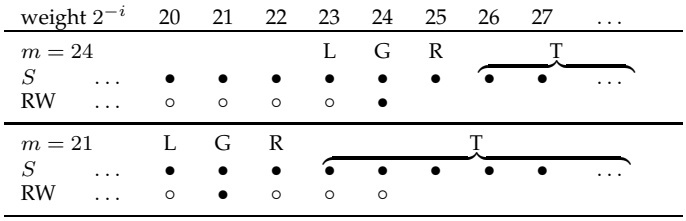

Fig. 6. Rounding in variable position for $m = 24$ and $m = 21$. RW holds the position of the rounding bit, marked as •.

bits, and its G and R bits are zero, the ADD1 bit is appended in the LSB of F_X.

The result of subtraction can be not normalized if $F_X \simeq 1.0$, for example, $1.0 - 0.125 = 0.875 < 1.0$. In this case, the result must be shifted one position to the left, and the exponent decremented by one. This condition is signaled by SH1L=1. Therefore, also for subtraction we need to consider two rounding positions, as in Fig. 5.

1) *The result is normalized (SH1L=0).*
Comparing expression (1) with the bit positions in Fig. 5 (top), if $G = 1$ either there are non-zero bits shifted out (T=1), or we have to add ADD1 (T=0). Therefore, we have to check for G only. However, there is an extra case when round up is needed: L=1 (odd LSB), G=0, R=1 and T=0. In this case, $R + \overline{T} = 2$ produces G=1 and the tie condition. Consequently, the result must be rounded up.
Summarizing, when SH1L=0 the round up condition is

$$\text{MUXC(0)}: \qquad G \vee L\,\overline{G}\,R\,\overline{T} = 1\,. \tag{4}$$

2) *The result is not normalized (SH1L=1).*
In this case, the rounding position is in R as in Fig. 5 (bottom) and the last bit is G. The condition for the round up is

$$\text{MUXC(1)}: \qquad G(R \vee \overline{T}) \vee RT = 1\,. \tag{5}$$

Also expressions (4) and (5) are evaluated in block "rounding control 1" in Fig. 4 and multiplexed with (1) and (2) (selection by EOP) in the 2-bit MUXC signal. The shifting condition is determined by

$$\text{SH1L} = (\overline{S1_0} \cdot S1_{-1}) \vee (\overline{S0_0} \cdot S0_{-1} \cdot \overline{\text{MUXC(1)}} \cdot \text{MUXC(0)})$$

Combining overflow and left-shifting, mutually exclusive conditions, the select bit MUXR is

$$\begin{aligned} \text{SHRL} &= \text{OVF} \vee \text{SH1L} \\ \text{MUXR} &= (\text{SHRL} \cdot \text{MUXC(1)}) \vee (\overline{\text{SHRL}} \cdot \text{MUXC(0)}) \end{aligned}$$

In Fig. 4 the rounding control is split in two blocks because the OVF and SH1L signals are available at a later time.

3.3 The Common Path

The double-path adder is completed by functional units which are common to both the the CLOSE and FAR paths. Beside the exponent handling (selection and update) and the sign computation, we need extra hardware to check for subnormals at the input and in the result. These are the functional units.

- A unit to detect if the exponent of the operands (E_X and E_Y) is zero and set the integer bit in $M_{X/Y}$ accordingly.
- A unit to check is the sum M_Z is a subnormal and in case it is, sets $M_Z = 0$ and $E_Z = 0$. The unit also checks for $\pm\infty$ and sets M_Z, E_Z, and S_Z accordingly.

4 TFP Adder

We now consider how to augment the *binary32* FP adder of Section 3 to support TFP for significant ranges from m=24 (*binary32*) to m=4. The precision of the significand m can be changed on a cycle basis by setting m in a control register.

The TFP adder can also be used to convert a standard format (*binary32*, *binary16*) to TFP, or vice-versa, by setting the output precision m and adding "0" to the number to be converted.

The main challenge to implement the adder in TFP is that the rounding must be done in a variable position. This is different from the case of the fused multiply-add where the operands are pre-shifted, depending of the exponent, and the rounding is done in a fixed position [8].

4.1 FAR Path

One way to round in a variable position is to shift both operands by 24-m bits and perform the rounding in fixed position in the least-significant bits of the result. However, the significand of the smallest number in the FAR path needs to be further shifted for alignment. The two shift-right operations can be combined by right-shifting the significand of the smallest number by (24-m)+D position. However, while m is known at the beginning of the operation, D (exponent difference) computation is in the critical path, and adding $D+m$ to form the control of the shifter increases the delay of the critical path in the first stage of the adder.

One alternative way is to use a reference bit-vector to hold the rounding position. We refer to this bit-vector as "Rounding Word", or RW. The RW is a bit-vector in which all bits are zero except the one in the rounding position. A simple decoder is used to generate RW depending on m.

Fig. 6 shows two example scenarios for m=24 (*binary32*) and m=21. In the figure, we assume the result S be normalized and the rounding position (bit of weight 2^{-24} for m=24) is marked by "•" in RW.

The RW is used to extract one bit from S by a simple AND-OR network. Fig. 7.a illustrates how the extraction of bit G is done for a 4-bit example. Clearly, for a full RW extraction, 22 bits for $m \in [4, 24]$, 22 AND gates and a tree of OR gates are required.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2019.2906907, IEEE Transactions on Computers

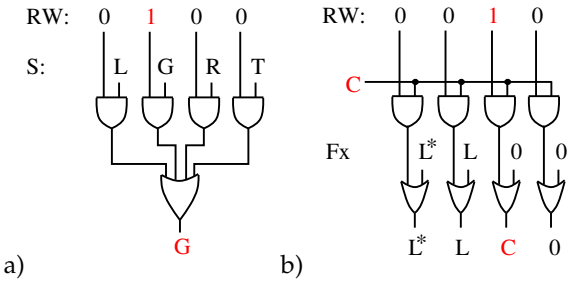

Fig. 7. AND-OR networks (4-bit examples): a) to extract bit G. b) to append bit C after bit L.

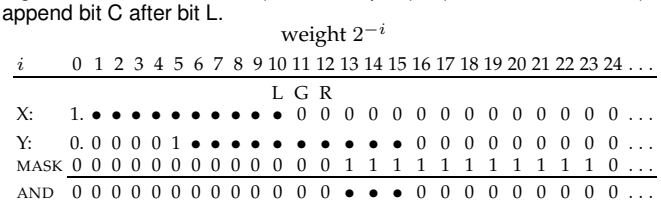

Fig. 8. Example of sticky bit calculation for $m = 11$ and D=5.

By shifting (hard-wired) RW, we can also extract the L*, L, and R bits by using the AND-OR network of Fig. 7.a.

The sticky bit (T) calculation is trickier. Suppose that m=11 and Y (significand with the smallest exponent) has to be shifted 5 positions to the right for alignment. The example is sketched in Fig. 8. The first two rows display X and the shifted Y significands, respectively. To extract the bits necessary for the sticky bit computation (bits of weight $2^{-13}, 2^{-14}, 2^{-15}$ in the figure), we can use a 24-bit vector with m-MSBs set to "1" and the remaining bits set to zero. This bit-vector, called MASK, is generated by another decoder depending on m. By shifting MASK to the right by m+2 positions[3] we obtain the pattern displayed in the third row of Fig. 8. As a final step, by applying the AND-OR network of Fig. 7.a to Y and MASK we obtain the sticky bit T. The bottom row in Fig. 8 displays the bits used to compute T (input to the OR tree).

The sticky bit calculation requires an additional shifter and a 48-bit AND-OR network. However, since MASK depends only on the precision m, the shifting as no impact on the latency and little impact on the power dissipation if m is not changed frequently.

Fig. 9 shows where this second shifter is placed in the augmented FAR path to support TFP. The AND-OR network is labeled "extract bit" in the figure. The extraction of bits L*, L, G, and R is done by combining the AND-OR networks for X and Y and followed by the computation of the L*, L, G, R bits of the result (block "extract LGR" in Fig. 9).

So far, we have described how to obtain the relevant bits for the rounding. Next, we describe how to perform the actual rounding for the FAR path.

The architecture is similar to the one of Fig. 4 with a few modifications. The round up bit (in CPA1) and ADD1 bit must be placed in a variable position. This is done in two steps (refer to Fig. 9):

1) Both X and Y are masked (logic AND) with MASK to preserve only the m-MSBs:

$$F_X = X \text{ AND MASK}\ , \quad F_Y = Y \text{ AND MASK}$$

3. The two extra positions are necessary to exclude the G and R bits.

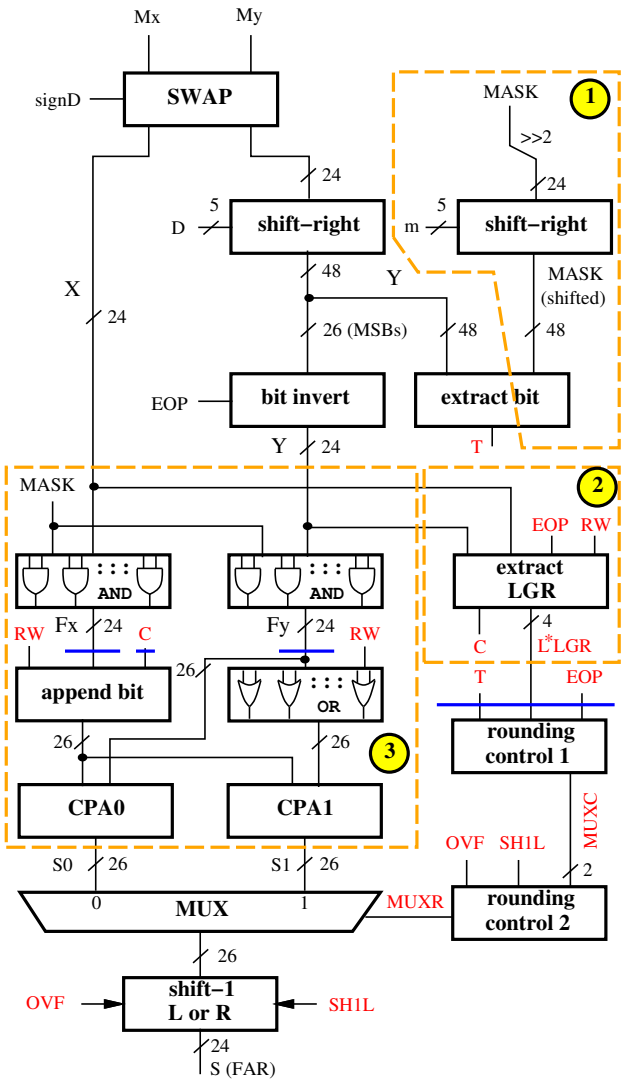

Fig. 9. Architecture of the TFP adder FAR path. Dashed boxes highlight augmentations with respect to Fig. 4.

This operation removes G, R and trailing bits in Y and has no effect on X, unless the significand has higher precision (conversion from longer formats).

2) The round up bit (U=1) and ADD1 (renamed C) are appended after the L bit of F_X and F_Y, as shown in Fig. 10.

Bit C, or ADD1, is computed in block "rounding control 1" in a different way from Section 3. The objective is to inject a carry in position L in CPA1:

$$\text{C} = \overline{\text{EOP}} \vee \text{EOP}(G \vee R\overline{T})$$

For addition (EOP=0), C is always set to "1". For subtraction (EOP=1), C can become the LSBs when the result needs to be left-shifted. To append the round up bit (U) in the correct position we simply OR the bits of F_Y to RW (rounding word). To append C, we need first to mark RW with the value carried by C (AND gates), then, the resulting bit-vector is ORed with the bits of F_Y. The two level network to append C is sketched in Fig. 7.b (for 4 bits), and the actual 26-bit network is labeled as "append bit" in Fig. 9.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2019.2906907, IEEE Transactions on Computers

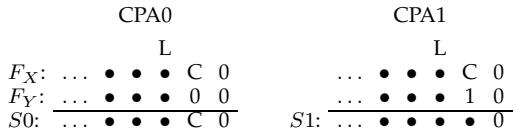

Fig. 10. Position where bits U and C are appendend to F_X and F_Y.

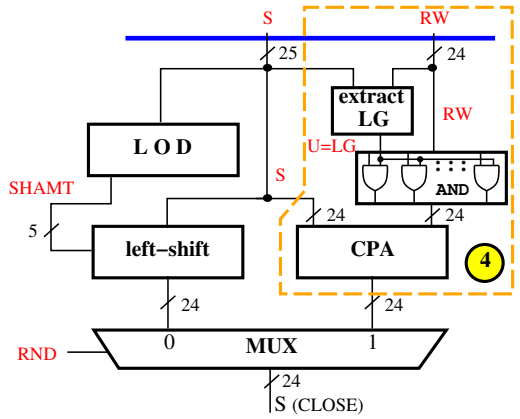

Fig. 11. Additions in the second stage of the CLOSE path in TFP-add.

4.2 CLOSE Path

As for the CLOSE path, the modifications are limited to the extraction of bits L and G, and appending the round up bit in the second stage of the CLOSE path. The result needs to be rounded up if LG=U=1, and U is ANDed with RW to form a bit-vector with a "1" in the rounding position, or a all-zero bit-vector. The rounding in the CLOSE path is illustrated in Fig. 11 (blocks "extract LG" and "AND").

4.3 Rounding in Variable Position: Summary

Table 1 summarizes how the rounding is done once the right bits L, G, R, and T are extracted (Fig. 6).

FAR path				CLOSE path	
EOP=0		EOP=1		leading 0's	
$\overline{\text{OVF}}$	G(R∨T∨L)=1	$\overline{\text{SHRL}}$	G∨L$\overline{G}$R$\overline{T}$=1	no round	LG=1
OVF	L(L*∨G∨R∨T)=1	SHRL	G(R∨$\overline{T}$)∨RT=1		

TABLE 1
Round-up conditions for TFP-adder.

4.4 The Common Path

The TFP adder is completed by the exponent and sign handling blocks. The top-level view of the TFP adder is shown in Fig. 12. A decoder provides the rounding word (RW) and the mask (MASK) to implement the TFP operation to the CLOSE and FAR paths. The figure does not include the sign computation and the units to determine the integer bit of M_X and M_Y.

The exponent is set to the one of the number with the largest exponent and later adjusted depending on the necessary normalization of the result. The unit "condition detect" ("cond. det.") in Fig. 12, determines if the result is a subnormal or plus/minus infinity. In this cases, M_Z is flushed to zero and the exponent E_Z is set accordingly.

Subnormals are flushed to zero. If any of the inputs is a subnormal, its significand is flushed to zero while the exponent difference is computed. If the result of the addition is a subnormal, M_Z and E_Z are flushed by the multiplexers at the bottom of Fig. 12.

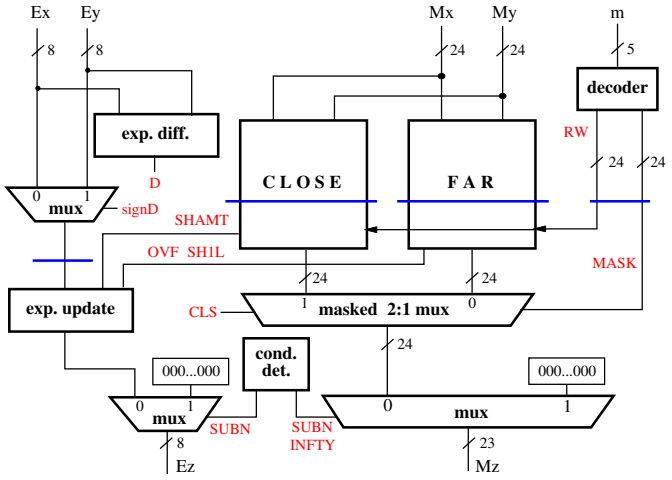

Fig. 12. Top-level view of the TFP adder (except sign computation).

	binary32	TFP	difference
max. delay [ps]	950	980	< 1 FO4
AREA(1)	6,200	8,200	+32%
Total power(2)	9.45	10.76	+14%

(1)[NAND2 equiv.], (2)[mW] at 1 GHz

TABLE 2
Post-synthesis comparison between *binary32* and TFP adders (RTNE).

5 Hardware Implementation

For the implementation of the TFP adder we opted for a 45 nm CMOS library of standard cells by using commercial synthesis and place-and-route tools (Synopsys). The FO4 delay[4] for this low power library is 64 ps and the area of the NAND-2 gate is 1.06 μm^2.

We set as the target clock period a delay of 15 FO4 which is reasonable for a pipelined FP-unit. Since 15 FO4 $\simeq$ 1.0 ns, the target throughput is 1 GFLOPS for a single adder.

The architecture of the TFP adder (TFP-add) of Fig. 12, supporting the IEEE 754 roundTiesToEven (RTNE) rounding mode, is pipelined in two stages, with pipeline registers placed as indicated in Fig. 9, Fig. 11 and Fig. 12 by the blue horizontal lines.

For comparison purposes, we also implemented the baseline *binary32* double-path adder (B32-add), supporting RTNE, of Fig. 2, Fig. 3 and Fig. 4.

A post-place-and-route comparison of the implemented units is reported in Table 2. Both units meet the timing constraint of $T_C = 1.0\ ns$.

For both units, the delay of the critical path is considered the same, since the difference of a few pico-seconds is less than 1 FO4 delay. Table 3 reports the timing paths in the four parts (two stages and the CLOSE and FAR paths) of the TFP-add. All timing paths are close to 1 ns because the synthesizer uses the available slack to minimize the area and the power dissipation. The critical path originates in the second stage of the FAR path.

The area overhead for the TFP-add is about 32% and it is mostly due to the parts labeled (1)–(4) (zones) in Fig. 9 and Fig. 11. The area and power dissipation overheads are

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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2019.2906907, IEEE Transactions on Computers

	CLOSE (Fig. 11)	COMMON	FAR (Fig. 9)
stage 1	↙ shift mux CA adder bit invert	exp. diff.	↘ SWAP shift-right bit invert extract LGR
	974 ps		963 ps
stage 2	LOD (SHAMT) →	exp. update mux E_Z	OR array CPA1 ← (SH1L)
	976 ps		**980 ps**

TABLE 3
Timing paths in TFP-add (critical path in boldface).

	AREA(1)	%	Power(2)	%
Zone 1	480	7.7	51	0.5
Zone 2	340	5.5	216	2.3
Zone 3	390	6.3	687	7.3
Zone 4	360	5.8	143	1.5
decoder	112	1.8	1	0.0
regs. RW+MASK	288	4.6	2	0.0
MUX M_Z	80	1.3	98	1.0
Sum	2050	33.1	1200	12.7

(1)[NAND2 equiv.], (2)[μW] at 1 GHz

TABLE 4
Overheads in TFP-add with respect to the *binary32* adder.

detailed in Table 4. In addition to the four zones ①–④, extra area is due to the decoder, the pipeline registers to hold RW and MASK and the extra gates to mask the result M_Z in the mux of Fig. 12. The area overhead in Table 4 (33%) is partly compensated by other blocks in the TFP-add (32% overhead in Table 2.

The power dissipation is estimated by running post-layout simulations (Synopsys VCS) on random generated *binary32* operands for the TFP-add and the B32-add. The overhead for the power dissipation, also in Table 4, is limited to about 13% in the four zones. The smaller overhead with respect to the area is due to the reduced switching activity in the decoder and in the barrel shifter, shifting the same bit-vector MASK of the same amount if the precision m does not change. In the registers holding RW and MASK, the power dissipation is limited to the leakage power because the registers are clock-gated unless m is changed.

To evaluate the trends for power dissipation when the precision m is scaled, we created traces from the TFP simulator for matrix multiplication which is the most common operation in machine learning algorithms.

Matrix multiplication consists of dot-products, i.e., multiplications followed by additions. We assume a 10×10 square matrix size with elements of 24-bit dynamic range. The FP numbers are represented by exponent $e = 8$ and variable significand precision: $m = \{24, 20, 16, 14, 11, 8, 6\}$. We consider four cases:

1) TFP multiplication followed by B32-add.
2) TFP multiplication followed by TFP-add.
3) *binary32* multiplication followed by B32-add.
4) *binary32* multiplication followed by TFP-add.

The average power dissipation in the TFP-add and B32-add are reported in Table 5 and the trends shown in Fig. 13.

Since the TFP-add is obtained by adding functional blocks to the B32-add, clearly, additions of the same

m	CASE 1 TFP-B32	CASE 2 TFP-TFP	ratio	CASE 3 B32-B32	CASE 4 B32-TFP	ratio
24	8.89	10.11	1.14	8.90	10.12	1.14
20	8.47	9.29	1.10	8.91	9.75	1.09
16	7.85	8.22	1.05	8.90	9.24	1.04
14	7.48	7.59	1.01	8.89	8.89	1.00
11	6.87	6.71	**0.98**	8.56	8.21	**0.96**
8	6.15	5.73	**0.93**	7.67	6.96	**0.91**
6	5.70	5.08	**0.89**	6.84	5.99	**0.88**

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

TABLE 5
Average power dissipation for matrix multiplication for cases 1–4.

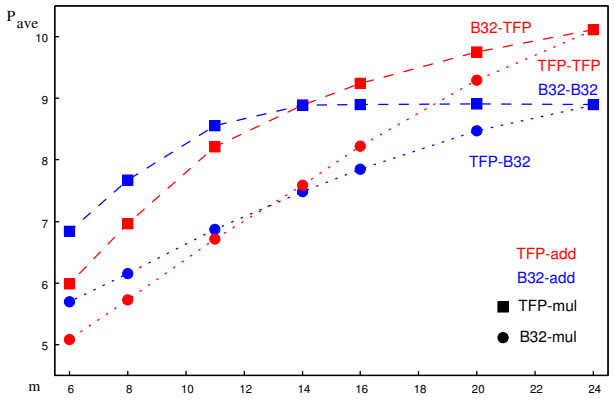

Fig. 13. Trends of average power dissipation in adders for matrix multiplication.

operands result in higher power dissipation for the TFP unit, as shown in Table 2. However, the TFP-add produces results of arbitrary precision m correctly rounded, while the B32-add produces *binary32* results. As m is scaled, the extra bits in the *binary32* representation of intermediate results produced by the B32-add cause the computation to consume extra power. The trend is more evident when the addends are produced by the *binary32* multiplier (values ■ in Fig. 13). For precisions below m=11, the TFP-add is more power efficient than the B32-add for matrix multiplication.

6 TFP and Deep Learning

Neural Networks (NNs) typically contain a very large number of parameters (weights w_i) and are usually trained iteratively over vast amounts of data. After training, the optimal weights are determined and the NN is ready to classify new input data (inference).

Fig. 14 shows the architecture for the two-hidden-layers NN used as example to illustrate the properties of TFP. The NN is used to interpolate a function $y = f(x)$ approximating the distribution of some sample points in the XY-plane. More detail on the example is given in [10].

We apply TFP to the operations in the NN of Fig. 14 for both training and inference, and we evaluate the trade off precision vs. error and power dissipation. Differently from [10], both the TFP multiplier (TFP-mul) and the TFP-add implement round-to-the-nearest unbiased rounding (RTNE).

Table 6 reports the trade off for training the NN with a "cosine-like" distribution for several TFP precisions. The table lists the approximation relative errors (ϵ_{ave}, ϵ_{max}) of the interpolated function with respect to the training points,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2019.2906907, IEEE Transactions on Computers

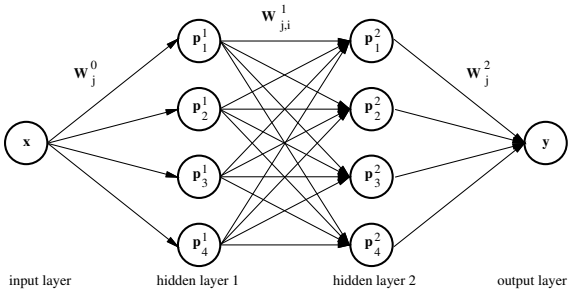

Fig. 14. Neural network with two hidden layers (depth=2).

m	e	ϵ_{ave}	ϵ_{max}	P_{add}	P_{mul}	P_{tot}	ratio
24	8	0.12	0.29	8.18	15.90	24.08	1.00
20	8	0.12	0.29	7.83	14.04	21.86	0.91
16	8	0.12	0.27	7.34	11.86	19.20	0.80
14	8	0.18	0.44	7.03	11.00	18.03	0.75
11	5	0.11	0.26	6.58	9.97	16.55	0.69
9	5	0.25	0.57	6.27	8.74	15.01	0.62
7	5	0.25	0.58	5.98	8.03	14.01	0.58
5	5	0.25	0.58	5.59	6.97	12.55	0.52

P_{ave} [mW] at 1 GHz.

TABLE 6
Relative errors and average power dissipation for TFP training.

the average power dissipation (for addition, multiplication and total), and the ratio to the *binary32* total dissipated power of the scaled precision runs.

The ratios in Table 6 show that the power dissipation drops linearly as m is scaled to about 50% for m=5, e=5. Therefore, finely tuning the precision, as in TFP, may significantly improve the performance of the training with respect to standard-precision formats.

Moreover, TFP can be advantageous also in the inference that usually requires lower precision than the training. A NN with two hidden layers has three different weights' levels: w_j^0, w_{ji}^1, and w_j^2 (Fig. 14). In TFP, the precision can be adjusted depending on the level.

In the next experiment, we estimate the performance of several combinations of the precision m=$\{24, 16, 8\}$ (e=8) for the classification of points within a given distance from the interpolated curve/function (more detail is given in [10]). Due to the approximation error (value of weights), out of 1,000 randomly distributed points in the XY-plane, 55 result miss-classified (about 5%). Table 7 reports the quantization error of the reduced precision weights as N_{quant}^{MC}: number of miss-classified points due to quantization. In the table, the first three columns indicate the precision in the levels of the NN and the last columns the corresponding power dissipation. The best trade off is probably for (m^0,m^1,m^2)=(16,8,8) with N_{quant}^{MC}=4 points and power savings of about 30%. Instead, the largest power reduction (40%, ratio$=0.61$) is reached for (m^0,m^1,m^2)=(16,8,8) with N_{quant}^{MC}=11 points.

7 Conclusions

In [7], we presented the Tunable Floating-Point representation to reduce precision and dynamic range of FP numbers for applications tolerating some error. In [7], we also presented architectures to implement TFP multiplication.

m^0	m^1	m^2	N_{quant}^{MC}	P_{add}	P_{mul}	P_{tot}	ratio
24	24	24	-	8.43	16.82	25.25	1.00
16	16	16	6	7.44	12.86	20.30	0.80
16	16	8	6	6.76	11.79	18.55	0.73
16	8	16	6	6.92	12.40	19.32	0.77
16	8	8	4	6.33	11.35	17.68	0.70
8	16	16	9	7.23	11.00	18.23	0.72
8	16	8	14	6.55	9.76	16.32	0.65
8	8	16	9	6.65	10.46	17.11	0.68
8	8	8	11	6.11	9.33	15.44	0.61

P_{ave} [mW] at 1 GHz.

TABLE 7
Classification errors (quantization) and average power dissipation for inference in TFP.

The main contribution of this work is the design of a TFP adder based on the double-path architecture. The greatest challenge in the design of the TFP adder is to provide unbiased rounding to-the-nearest in a variable position with a limited overhead in delay, area and power dissipation. The adopted hardware solutions guarantee no timing overhead with respect to a *binary32* adder, and a reasonable overhead in power dissipation, reduced when the precision is scaled.

The main advantage of TFP is the capability to tailor the necessary precision in different parts of an algorithm to obtain a higher power efficiency. Moreover, TFP precision can be adjusted on a cycle basis giving great flexibility to programmers, and, at the same time, alleviating programmers from inserting scaling operations in the code necessary for fixed-point formats.

Future work may address both increasing the flexibility of TFP units by making programmable the rounding mode (i.e., by disabling blocks not needed), and increasing the area utilization by splitting the data-paths in two TFP units when the precision $m < 12$.

References

[1] NVIDIA Inc. "NVIDIA Tesla V100 GPU Architecture". [Online]. Available: http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

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

[3] 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

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

[5] N. P. Jouppi *et al.*, "In-Datacenter Performance Analysis of a Tensor Processing Unit," in *ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017, pp. 1–12.

[6] V. Popescu *et al.*, "Flexpoint: Predictive Numerics for Deep Learning," in *25th IEEE Symposium on Computer Arithmetic*, Jun. 2018.

[7] A. Nannarelli, "Tunable Floating-Point for Energy Efficient Accelerators," in *25th IEEE Symposium on Computer Arithmetic*, Jun. 2018, pp. 29–36.

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

[9] J. D. Bruguera and T. Lang, "Using the reverse-carry approach for double datapath floating-point addition," in *Proc. of 15th IEEE Symposium on Computer Arithmetic*, 2001, pp. 203–210.

[10] M. Franceschi, A. Nannarelli, and M. Valle, "Tunable Floating-Point for Artificial Neural Networks," in *25th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, Dec. 2018, pp. 289–292.