Journal of Engineering Sciences

Vol 12, Issue 11, NOV /2021

ISSN NO:0377-9254

DESIGN OF A POWER EFFICIENT 32- BIT POSIT MULTIPLIER

L.SRAVANI[1], Mr. MIDDE ADISESHAIAH[2]

[1]P.G Scholar, Chadalawada Ramanamma Engineering College, Tirupathi – 517506.
[2]Assistant professor, Chadalawada Ramanamma Engineering College, Tirupathi – 517506.
lakidisravani457@gmail.com[1],
adi.techster@gmail.com[2]

Abstract—

Posits are tapered precision number to replace IEEE floating point. It provides more precision, lower complexity and low power implementations than IEEE floating point. In this project, a power efficient posit multiplier architecture is proposed.

The mantissa multipliers is still designed for the maximum possible bit-width, how ever, the whole multiplier is divided into multiple smaller multipliers. Only the required small multipliers are enabled at run-time. Those smaller multipliers are controlled by the regime bit width which can be used to determine the mantissa bit-width by using this method, power dissipation can be achieved with negligible area. The effectiveness of the proposed design is synthesized and simulated using Xilinx software. This design technique we applied to 32-bit posit formats in this brief and an average of 12% power reduction can be achieved with negligible area and timing overhead.

Keywords: Posit number system, Posit Multipliers, Power dissipation

I. INTRODUCTION

Posits, a new data-type modeled to build as an alternative for IEEE floating point number.

Posit number do not demand any operand to be of varying size (variables) because if they find any answer to be wrong, they do the rounding process. This behavior is very much unlike the universal numbers or also known as Unum. The posit system yields many benefits, which may include their dynamic range being large, use of a simple hardware execution system, handling of exceptional cases are comparatively better, being greater in terms of accuracy. Posits do not take the values of infinity and zero. As enunciated before, posit processing system occupies less equipment compared to IEEE floats processing machine. Additionally, they have a irregular or inconsistent distribution of data that settles the need in certain applications, for example it may be useful in deep learning. The 8-bit or 16-bit posits are generally utilized in deep learning applications. 32-Bit posits arithmetic may be utilized scientifically computing fields. The generalized pattern for the posit data-type or number system is shown in Figure. Posit (nb, es) is the representation of posits where nb means total or absolute bit width and es means the bit width foe exponent. It constitutes 4 parts: 1.sign-s 2.regime-rg 3.exponent-exp 4.mantissa-frac. The component width of the bit is variable. The regime values are always varying. The rest of the positions for the bits shall be taken by mantissa and also the exponent. This happens only when the regime does not reserve all the positions. A numeral illustrated in the format for posits arithmetic:

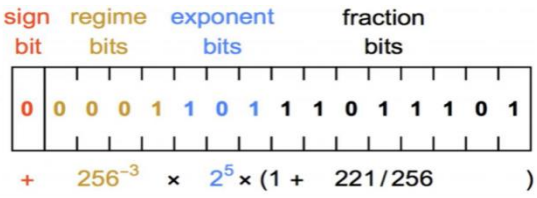


Figure-1: Posit number system

$$value = (-1)s \times useedrg \times 2exp \times (1 + frac)$$
where $useed = 2^{2es}$.

We have dynamic ranging bit width here as shown in above, exemption being the sign bit sized 1. The regime as well as the sign bit should always be present in the general structure. Exponent and mantissa sections occur if there any leftover positions for the bits. As such the fraction part which includes the implicit bit also has a bit-width ranging from 1 to nb – es value. Every time we do any operations, the value of mantissa need not be to its full extent always. Using a maximum bit multiplier all the times results in unwanted consumption of power. We establish the successful

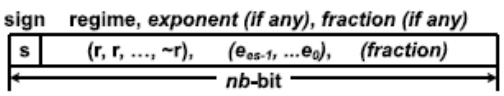
implementing of a 16– bit posit multiplier, where the mantissa (fraction) multiplier is split into several small ones and use them when required. Thus we efficiently design a low power consuming posit multiplier and reduce increasing power consumption. This architecture can also be used in various multipliers other than posit multipliers to improve power efficiency of the corresponding component or device.

1.1 The use of Posit Arithmetic in NNs

Posits were introduced by John Gustafson in 2017. Since then, multiple works have explored the benefits of this novel format against the standard floating-point, and most of them focusing on NNs. Johnson designed an arithmetic unit for combining posit addition together with logarithmic multiplication for performing CNN inferences. Authors in employ a posit DNN accelerator to represent weights and activations combined with an FPGA soft core for 8-bit posit exact-multiply-and-accumulate (EMAC) operations. In all these works, DNN training is performed in floating-point, while the inference stage is performed in low-precision posit format. Later works proposed different approaches for training NNs using the posit format, either directly training on this format with different precision or with the help of a warm up training using floating-point format.

1.2 Posit Number System

The general format of a posit number is shown in below figure. A posit number Posit(nb, es) is defined with the total bit-width nb and the exponent bit-width es. It has four components: sign (s), regime (rg), exponent (exp), and mantissa (frac). The component bit-width is not constant. The regime bit-width varies for different values. The exponent and the mantissa will occupy the remaining bit positions and they will not be included in the format when the regime occupies all bit positions. The value of a number represented in posit format is:



$$value = (-1)^{s} \times useed^{rg} \times 2^{exp} \times (1 + frac)$$

II. RELATED WORK

A posit format is defined as a tuplehn, esi, where n is the total bit width and es is the maximum number of bits reserved for the exponent field. As Figure shows, posit numbers are encoded with four fields: a sign bit (s), several bits for encoding the regime value (k), up to es bits for the exponent (e), and the remaining bits for fraction (f). Thus, the numerical value X of a generic Posithn, esi is expressed by (1).

$$X = (-1)^{s} \times (2^{2^{es}})^{k} \times 2^{e} \times (1 + f), \quad (1)$$

$$k = -x_{n-2} + \sum_{i=n-2}^{x_i \neq x_{n-2}} (-1)^{1-x_i}. \quad (2)$$

The main differences with floating-point format are the utilization of an unsigned and unbiased exponent, if there exists such exponent field, and the existence of the regime field. This new field consists of a sequence of bits with the same value finished with the negation of such value. Provided that X = xn−1xn−2...x1x0, this regime can be expressed as (2) shows. It is noteworthy that, while the new regime field provides important scaling capabilities that improve the dynamic range of posits, detecting the resulting varying-sized fields adds hardware overhead.

III. EXISTING METHOD

Currently the working method of posit multiplier uses the modified booth's algorithm technique. This method is also known as bitpair algorithm or radix-4 algorithm. There is a possibility to decrement the partial products number. Here we do not shift and add for all the columns of the multiplier and later doing multiplication with 0 or 1. But what we do here is to multiply every 2nd column with 0 or ±1 or ±2. Both the methods yield similar results. Radix-4 booth encoder compares 3 bits at a time which is also known as the overlapping method. By adding a zero the the left end of the number, we start pairing them into a batch of 3 numbers together shown in Figure.

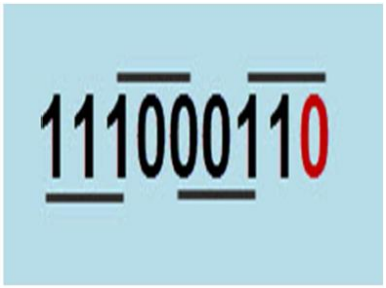


Figure-2: 3-bit pairing for Booth recoding

Operating procedure for Radix-4 booth encoder is as shown in the table. The results achieved from multiplying the different multiplier states with 0, ±1 and ±2 are described.

Radix-4 Booth encoding procedure is as follows: (1) Make sure n is even, so if necessary create an extension using the sign bit. (2) Adjoin zero value to the LSB in our multiplier. (3) Depending upon each and every value, we form partial products as -y, +y, -2y, +2y or 0. Two's complement procedure is done to deal with negative values. After shifting the multiplier y bit one by one, multiplication process is thus proceeded. We obtain partial produced reduced by twice its size which is a main advantage. This reduction facilitates the decreased delay in propagation while the circuit is operating. The main disadvantage of the circuit mentionable must be the difficulty in the construction of the circuit hardware.

Table-1: Radix-4 method booth recoding

Multiplier Bits Block			Recoded 1-bit pair		2 bit booth	
i+1	i	i-1	i+1	i	Multiplier Value	Partial Product
0	0	0	0	0	0	Mx0
0	0	1	0	1	1	Mx1
0	1	0	1	-1	1	Mx1
0	1	0	1	0	2	Mx2
1	0	0	-1	0	-2	Mx-2
1	0	1	-1	1	-1	Mx-1
1	1	0	0	-1	-1	Mx-1
1	1	0	0	0	0	Mx0

IV. PROPOSED SYSTEM

The proposed posit multiplier; the design differs from the existing method in the mantissa multiplier. The mantissa multiplier uses the Modified booth multiplication, consisting of a (nb - es) bit mantissa multiplier. While doing multiplication, we do not always require a maximum bit-width mantissa multiplier. That is, there is no need to always use nb – es mantissa unit. Generally, the bits which are not used in multiplier and multiplicand will be assigned to zero normally. But those bits will be reverted to value one during recoding the multiplier when it is negative. Thus it results in an unwanted signal toggling. The unnecessary signal toggling must be avoided to reduce power consumption. Though we use the same radix-4 booth multiplication in the proposed system, the power efficiency can be achieved by splitting the multipliers into smaller ones and accessing or controlling it through a control signal. Such that this design also involves generating a control signal to enable the required smaller multiplier component when required during our run time. The design details of 32-bit multiplier are discussed in detail.

4.1 Reduction of Multiplier

We do extension as 32 bit multiplier from the steps performed in the multiplier after the partial product generation. From this picture it is clear, that the number of bits used to generate an answer for 32-bit multiplication is very time, space and also memory consuming. Since we multiply all the 32 bit of the multiplier and multiplicand, an annoyingly long process takes place.

Figure-3: Partial product addition after multiplying two 16-bit numbers in the existing method.

Thus in the proposed system we have decided to divide or split the multiplier digits. In the proposed two 16-bit mantissa multiplier. eight partial products are generated while using radix-4 booth algorithm. Each partial product is of 16 bits. In the proposed design, the 16-bit multiplier is divided into 4 groups: the MSB 3-bit forms one group, and the remaining 12-bit can be grouped into three 4-bit

groups. Correspondingly, the eight partial products are divided into 4 groups, PPH_1, PPH_2, PPH_3 and PPH_4. If the multiplier is less than 3-bit, then only the two partial products in PPH_4 are generated while all others are set to zero. If the multiplier is more than 3-bit but less than 7- bit, then partial products in PPH_3 and PPH_4 are generated. If the multiplier is more than 7-bit but less than 11-bit, then partial products in PPH_2, PPH_3, and PPH_4 are generated. Finally, if the multiplier is more than 11-bit, then all the partial products are generated. Similarly, the 15-bit multiplicand is also divided into 4 groups. Each partial product is correspondingly divided into4 groups, PPV_1, PPV_2, PPV_3, and PPV_4. If the multiplicand is less than 3-bit, then only PPV_4 is generated while all others are set to zeros. If the multiplicand is more than 3-bit but less than 7- bit, then PPV_3 and PPV_4 are generated. If the multiplicand is more than 7- bit but less than 11-bit, then PPV_2, PPV_3, and PPV_4 are generated. Finally, if the multiplicand is more than 11-bit, then all bits in the partial product are generated.

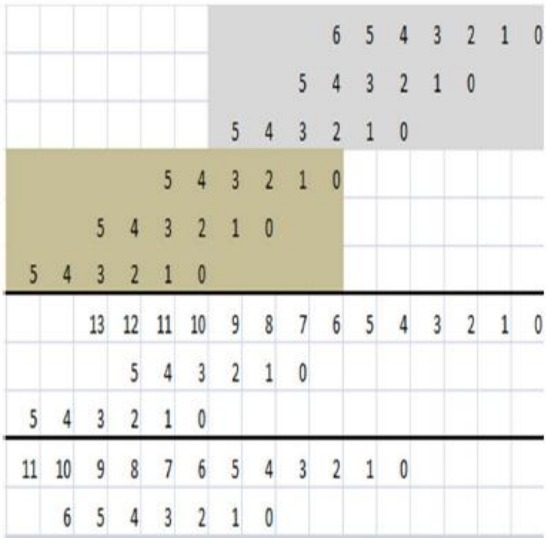


Figure-4: shows an example when both the multiplier and multiplicand are less than 7-bit.

4.2 Control Signal

For completing the design of our multiplier, we need a control signal. We divide the 16-bit digit input into four parts as already mentioned above. To generate Control Signal for both the inputs, one has to identify the position in where the binary digit '1' makes its appearance. After discovering the position of 1, comparison of the divided subgroups and our input digit is performed. According to that grouping, we generate the control signal for both inputs. This in turn implies the selection of smaller multiplier which we need to use for the process. All in all we sum-up the use of control signal to select the smaller multiplier we have partitioned for the operation.

V. SYNTHESIS RESULTS

In order to get comparable results from this work, several multipliers are synthesized using Xilinx Design and without placing any timing constraint. We measured the delay, area, power and energy of the different multipliers. Not only multiple posit configurations have been synthesized, but also for reach configuration three different designs are taken – sequential (or pipelined) design, combinational one and combinational with no hard multipliers nor DSP blocks. These three designs are obtained using the different options that FloPoCo provides for generating the VHDL code. Table presents the delay, area, power, and energy of the posit multipliers after the synthesis. In case of sequential designs, the number of stages is indicated between parenthesis next to delay value.

Table-2: Posit multipliers synthesis results.

		Posit(*n, es*) configuration				
		(8, 0)	*(8, 1)*	*(8, 2)*	*(16, 1)*	*(32, 2)*
Delay (ns)	Sequential	0.8 (8)	0.79 (8)	0.78 (7)	1.06 (10)	2.3 (15)
	Combinational	3.59	3.52	3.17	6.2	10.34
	Combinational, No hm	3.36	3.23	3.18	6.2	9.6
Area (µm)	Sequential	2799	2745	2481	6898	24299
	Combinational	1488	1483	1415	3865	15459
	Combinational, No hm	1271	1152	1048	3865	21894
Power (µW)	Sequential	397	384.3	313.7	862.1	2269.2
	Combinational	631.3	562.1	428.4	2609.6	12693.6
	Combinational, No hm	612.4	503.9	424	2609.6	13053.3
Energy (pJ)	Sequential	0.317	0.303	0.244	0.913	5.219
	Combinational	2.266	1.978	1.358	16.179	131.251
	Combinational, No hm	2.057	1.627	1.348	16.179	125.311

Forexample,8,bitmultipliersrequire at least 7states,whichisalotforthiskindofcomponents.A sexpected,incombinationaldesignstheareaande nergyconsumptionare smaller, but the delay increases compared with the corresponding pipe lined designs.Thisareaandconsumer education isalsoexpectedtooccurwhencomparingcombin ationaldesigns with and without hard multipliers, but to a lesser extent. However, as Table2shows,therearefewexceptions:Posit(16,

1),whichprovidethesameresultsforbothcombin ational multipliers, and Posit(32, 2), whose area – and therefore power – increases when not using hard multipliers.

Table-2: Comparison of posit multipliers synthesis area results

Datapath	Posit(16, 1)		Posit(32, 2)	
	Slice LUT	Used DSP	Slice LUT	Used DSP
Literature	**218**	**1**	**572**	**4**
Sequential	321	1	891	2
Combinational	266	1	927	2
Combinational, No hm	266	1	1640	0

VI. CONCLUSION

The idea proposed in the paper is a 32-bit Posit multiplier architecture with power efficiency. Intrigued by the idea of reconstructing the multiplier unit for mantissa into smaller parts, because of the whole mantissa unit is not used entirely all the time, we have built the hardware. To limit the power consumption, we use only the necessary potion of the multiplier. Our method is evaluated for 16-bit multiplier, whereas we can extend the work for 8-bit and 32-bit posit multipliers using the same technique. For futuristic purposes, more power reduction techniques for multiplier architecture can be developed. The work need not be necessarily limited to multipliers alone. Future works can be deployed also for Posits Adder or Posits Multiply Accumulate functions.

REFERENCES

[1] J. L. Gustafson and I. Yonemoto, “Beating floating point at its own game: Posit arithmetic,” Supercomput. Front. Innovat. Int. J., vol. 4, no. 2, pp. 71–86, Jun. 2017.

[2] IEEE Standard for Floating-Point Arithmetic, IEEE Standard 754-2008, Aug. 23, 2008, pp. 1–70.

[3] Z. Carmichael, S. H. F. Langroudi, C. Khazanov, J. Lillie, J. L. Gustafson, and D. Kudithipudi, “Deep positron: A deep neural network using the posit number system,” CoRR, vol. abs/1812.01762, pp. 1–6, Dec. 2018.

[4] J. Johnson, “Rethinking floating point for deep learning,” CoRR, vol. abs/1811.01721, pp. 1–8, Nov. 2018.

[5] M. Klöwer, P. D. Düben, and T. N. Palmer, “Posits as an alternative to floats for weather and climate models,” in Proc. Conf. Next Gener. Arithmetic, Mar. 2019, pp. 1–8.

[6] R. Chaurasiya et al., “Parameterized posit arithmetic hardware generator,” in Proc. IEEE 36th Int. Conf. Comput. Design (ICCD), Orlando, FL, USA, Oct. 2018, pp. 334–341.

[7] M. K. Jaiswal and H.-K. So, “Architecture generator for type-3 unum posit adder/subtractor,” in Proc. IEEE Int. Symp. Circuits Syst. (ISCAS), Florence, Italy, May 2018, pp. 1–5.

[8] H. Zhang, J. He, and S.-B. Ko, “Efficient posit multiply-accumulate unit generator for deep learning applications,” in Proc. IEEE Int. Symp. Circuits Syst. (ISCAS), Sapporo, Japan, May 2019, pp. 1–5.

[9] A. Podobas and S. Matsuoka, “Hardware implementation of POSITs and their application in FPGAs,” in Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops (IPDPSW), Vancouver, BC, Canada, May 2018, pp. 138–145.

[10] A. D. Booth, “A signed binary multiplication technique,” Quart. J. Mech. Appl. Math., vol. 4, no. 2, pp. 236–240, 1951.

[11] SoftPosit-Python. Accessed: Oct. 2018. [Online]. Available: https://posithub.org/docs/PositTutorial_Part1.html

[12] Z. Carmichael, H. F. Langroudi, C. Khazanov, J. Lillie, J. L. Gustafson, and D. Kudithipudi, “Performance-efficiency trade-off of low-precision numerical formats in deep neural networks,” in Proc. Conf. Next Gener. Arithmetic, Mar. 2019, pp. 1–9.