2018 31th International Conference on VLSI Design and 2018 17th International Conference on Embedded Systems

Floating Point Multiplication Mapping on ReRAM based In-Memory Computing Architecture

Tarun Vatwani, Arko Dutt, Debjyoti Bhattacharjee‡ and Anupam Chattopadhyay
School of Computer Science and Engineering, Nanyang Technological University, Singapore 639798
Corresponding Email: ‡debjyoti001@e.ntu.edu.sg

***Abstract*—Low leakage power, high endurance and non-volatile storage capabilities have made memristive devices, such as Resistive RAM (ReRAM) popular. ReRAMs also offer in-memory computing capabilities by means of stateful logic operations. However, there are no standard libraries for floating point operations on ReRAM-based in-memory computing platforms. In this paper, we propose a mapping for one such mathematical function, namely multiplication for floating point numbers. We undertook a detailed study to derive a mapping with low memory footprint on ReVAMP architecture, which is an in-memory computing platform based on ReRAM crossbar arrays. For multiplication of two IEEE-754 compliant single (or double) precision numbers, the proposed mapping requires 1.944 Kb (or 8.904 Kb) of computation memory, 374.4 Kb (or 3457 Kb) of instruction memory while consuming an energy equivalent to 5.6 pJ (or 471 pJ).**

I. Introduction

Resistive Random Access Memories (ReRAMs) are one of the most promising emerging technologies for logic and storage applications. ReRAMs offer non-volatile, high endurance, high density and multi-state storage capabilities with fast access times, while at the same time allowing stateful logic operations [1]. ReRAMs have also been used for designing neuromorphic circuits [2], content addressable memories and in-memory computing platforms [3], [4]. Realization of large passive crossbar arrays can be achieved by means of a select device in series to a switch (1S1R) or a Complementary Resistive Switch (CRS) that prevents parasitic currents [5].

The adoption of a computation platform depends on the available tools and libraries for the platform. For example, BLAS libraries are available for scientific computing on GPGPUs and CPUs [6]. Floating-point multiplication operation finds usage in multiple applications such as digital signal processing, data mining, etc. There has been existing works to map adder [7], integer multiplier [8] and binary matrix vector operations on ReRAMs [9]. In this work, we extend further in this direction to map floating point multiplication on ReVAMP, a ReRAM based in-memory computing platform [4]. ReVAMP harnesses bit-level parallelism inherent to ReRAM crossbar arrays. The key contributions of the current paper are :-

- We report the first in-memory implementation of IEEE-754 floating point multiplier on ReVAMP.
- A novel multiplier mapping for computing product of two binary numbers has been proposed.
- We studied the performance of the proposed implementation in terms of throughput and memory overhead.

The rest of the paper is organized as follows. In section II, a brief introduction to IEEE-754 floating point number representation is presented. In addition, the ReVAMP architecture is introduced. Section III presents the mapping for floating point multiplication on the ReVAMP architecture. In section IV, we present detailed experimental results and a summary of existing works. Section V presents a conclusion to the paper.

II. Preliminaries

In this section, we present the IEEE-754 floating point representation of any real number. In addition, we also introduce the basics of ReVAMP – a ReRAM based in-memory computing architecture.

A. IEEE-754 Floating point representation

The IEEE format for a single precision (32-bit) number is depicted in Fig. 1. Any real number X is expressed as:

$$X = (-1)^{Sign} \times (1.Fraction)_2 \times 2^{(Exponent-127)}$$

1-bit	8-bit	23-bit
Sign	Exponent	Fraction

Fig. 1: IEEE 754 format for 32-bit floating point number.

A leading sign bit is used in the format with value '0' indicating the number is positive else it is negative. The fraction stores the values after decimal places of the number. It is to be noted that a 1 is always implied in the decimal place and is not directly used in the representation. The mantissa represents the implied 1 with it - $(1.Fraction)_2$. The Exponent is expressed in an excess$-B$ representation such that the exponent is always a positive number. If the Exponent consists of $e-$bits, then the bias B is $2^{e-1} - 1$.

B. ReVAMP Architecture

One of the recently proposed ReRAM based in-memory architecture is ReVAMP [4]. It supports word-serial execution of instructions by exhibiting bit-level parallelism. It uses two separate memories — data and computation memory (DCM) and instruction memory (IM). In-memory computation takes place in the DCM. The DCM is a ReRAM crossbar array which constitutes of multiple 1S1R ReRAM devices [10]. The DCM is accessed as w_D-bit wide words. Each 1S1R device has two-terminals, namely a wordline wl and bitline bl and an internal resistive state Z. A ReRAM device has an intrinsic property of implementing its next state as a Boolean Majority-three function M_3 with bitline input bl inverted i.e., the next state $Z_n = M_3(Z, wl, bl') = Z.wl + wl.bl' + bl'.Z$.

2380-6923/17 $31.00 © 2017 IEEE
DOI 10.1109/VLSID.2018.104

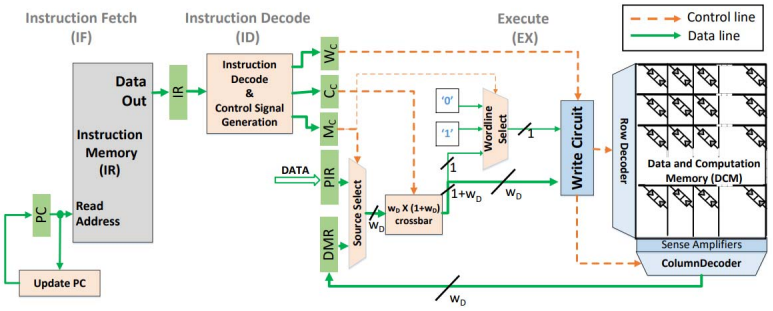

Fig. 2: ReVAMP Architecture [4].

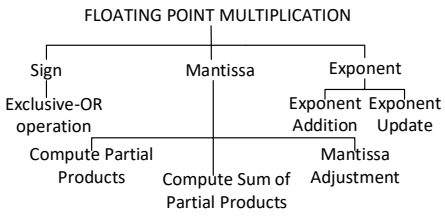

Fig. 3: Components of Floating point multiplication.

The instruction set has two instructions, namely, 'Read' and 'Apply' whose format, as shown below.

Read wl
Apply $wl\ s\ ws\ wb\ (v\ val_{w_D-1})\ \ldots\ (v\ val_0)$

A word wl is read out from DCM and stored in Data Memory Register (DMR) using the 'Read' instruction. This word available in the DMR can act as input for the following instructions. For computation, 'Apply' instruction is used. In 'Apply' instruction, wl specifies the word on which computation occurs, 1-bit flag s selects the input data source (either primary input register PIR or DMR), 2-bit flag ws selects the wordline input - 00 selects logic zero '0', 01 selects logic one '1', 10 is forbidden and 11 selects the bit specified by wb-address within the chosen data source for use as wordline input. Pairs (v, val) specify individual bitline inputs where v=1 indicates the input is valid else it is not used, and val specifies the address within the chosen data source from which the intended bit is used as bitline input.

III. IEEE-754 Multiplication on ReVAMP

In this section, we present the mapping for multiplication of IEEE-754 floating-point numbers on ReVAMP, with major focus on low memory rather than highly-parallelized floating point operations. For generalization, let us consider that the n-bit floating point number in IEEE format is represented by e-bit exponent, $(k-1)$-bit fraction with 1-bit leading sign-bit, such that $n = e + k$. The operations required to perform floating point multiplication is shown in Fig. 3. The individual operations are described below.

A. Computing Sign

For computing the sign bit of the result, we require to XOR the sign bits of the two inputs. The XOR of two one-bit operands a and b can be expressed as:

$$a \oplus b = a.b' + a'.b = a.b' + (a + b')'$$

The steps to realize XOR of 2-inputs (of 1-bit each) using ReRAM crossbar array is depicted in Fig. 4. It is implemented using a 3×1 ReRAM crossbar array, i.e., a crossbar with 3 wordlines and one bitline. We refer to the top wordline as wordline 1 and thereafter the next wordline as 2 and so one. Similarly, for referring to the bitline, we refer to the left-most bitline as 1, the next bitline on the right as 2 and so on. In addition, we provide two basic formations using $Apply$ instruction which will be used in the rest of the paper – applying a '1' as wordline input to a crossbar device will perform a Boolean *OR* operation of the inverted bitline input and crossbar state, while applying a '0' as wordline input to the device will perform a Boolean *AND* operation of the inverted bitline with crossbar state.

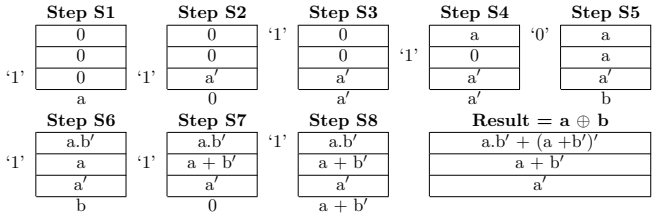

Fig. 4: 2-input (1-bit each) XOR Implementation Steps

Now, we present the steps to realize two input XOR.

Step S1: Input a is loaded into memory in inverted form using $Apply$ instruction, by applying '1' to third wordline and input a to bitline. This is because $M_3(1, 0, a') = a'$.

Step S2: a' is read out using a $Read$ instruction and is now available in the DMR.

Step S3-S4: a is computed and stored in memory locations corresponding to wordlines 1 and 2, by applying '1' to the wordlines respectively and a' to the bitline in subsequent steps, since $M_3(1, 0, a'') = a$.

Step	ReVAMP instruction
S1	Apply 3 0 01 0 1 1
S2	Read 3
S3	Apply 1 1 01 0 1 1

We formally introduce how these steps can be represented as ReVAMP instructions. In step S1, '1' is applied as wordline 3 input. So the opcode is $Apply$ for this instruction, $wl = 3$, $ws = 01$ and the fifth bit after $Apply$ is $wb = 0$ may be neglected. Since bitline 1 input is from primary input a, the second bit after $Apply$ is $s = 0$. Only one (v,val) pair exists for this mapping since one bitline is used. The last two bits specify $v = 1$ and $val = 1$ respectively. In step S2, a $Read$ instruction is used to read from memory corresponding to wordline 3. So for second instruction, opcode is $Read$ and $wl = 3$. The read out data is stored in DMR, available for use in next cycles. Step S3 uses an $Apply$ instruction with '1' as wordline 1 input and DMR data a' as bitline 1 input. So $wl = 1$, and $s = 1$ denoting DMR input; ws, wb and pairs (v,val) remain same as

the instruction for step 1. Similarly, the rest of the steps can be represented in terms of ReVAMP instructions.

Step S5-S6: $a.b'$ and $a + b'$ are computed and stored in memory, by applying '0' and '1' to wordlines 1 and 2 respectively in subsequent steps and input b to bitline input, since $M_3(a, 0, b') = a.b'$ and $M_3(a, 1, b') = a + b'$.

Step S7-S8: $a + b'$ is read out from memory in step S7 and applied as bitline input along with '1' applied as wordline 1 input in step S8 to compute Result = $a \oplus b$.

The result of XOR is available in first wordline position of the DCM. Each Read or Apply instruction is effectively executed in a single cycle. Therefore, the computation of Sign bit requires 8 cycles in total.

B. Computing Exponent

It is evident that the exponents add up when the two numbers are multiplied. The exponent in the IEEE-754 format are biased, therefore the resultant exponent $E_R = E1 - B + E2$, where $B = 2^{e-1} - 1$ is the bias for a given precision. The computation of E_R can be treated as two subsequent addition operations. The exponent might have to be updated based on Mantissa MSB Carry or MMSBC. We will discuss about MMSBC later in mantissa adjustment phase of mantissa multiplication.

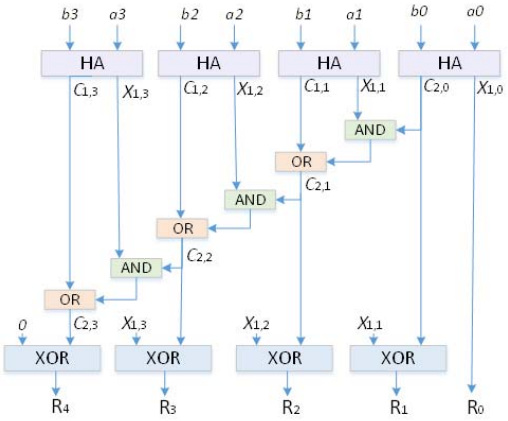

Fig. 5: 4-bit Adder Schematic used to compute exponent. HA - Half Adder, Result of Addition $R = (R_4R_3R_2R_1R_0)_2$.

We use a $3\times(e+1)$ ReRAM crossbar to implement addition of two e-bit exponents. For demonstration, we consider adding 4-bit exponents E1 and E2 using the schematic shown in Fig. 5, which is mapped to a 3×5 ReRAM crossbar. The mapping steps are depicted in Fig. 6 and explained below.

Steps S1-S2: $4-$bit primary input a is loaded into memory in inverted form to wordlines 1, 2 as shown in Fig. 6.

Step S3: $a'.b'$ is computed by applying '0' to wordline input and $4-$bit primary input b via the bitlines.

Step S4: $a'+b'$ is computed by applying '1' to wordline input and $4-$bit primary input b to the bitlines.

Steps S5-S6: $a' + b'$ is read out and ORed with the first wordline to compute XNOR of a and b.

Steps S7-S9: XNOR of a and b is read out, all bits of first wordline are reset to '0' and XOR of a and b is computed

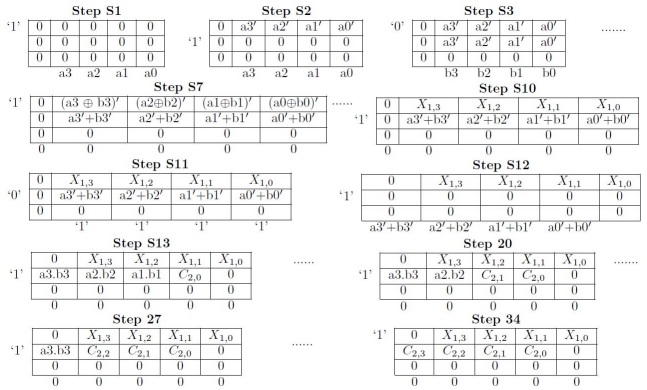

Fig. 6: 4-bit Adder Implementation Steps - Computation of $X_{1,i}$ and $C_{2,j}$.

and stored in first wordline of memory leaving the first bitline position. The output from step S9 represents X_1.

Steps S10-S11: $a' + b'$ are read out from memory locations corresponding to second wordline in step S10; the corresponding memory devices are then reset in step S11.

Step S12: $a.b$ is computed by applying '1' to second wordline input and the read-out data to bitline inputs. The output from this step represent C_1.

As of now, 4 half-adder circuits have been implemented in parallel to generate sum $X_{1,i}$ and carry-in $C_{1,i}$, represented by a set of two equations:

$$X_{1,i} = a_i \oplus b_i \qquad i \in \{0, 1, 2, 3\}$$
$$C_{1,i} = a_i \cdot b_i$$

The carry generated from each bit addition of two inputs needs to be propagated in a sequence of steps from the least significant bit position to the most significant bit position allowing addition of immediate left-adjacent input bits. The carry input for addition of immediate left-adjacent bits $C_{2,j}$ can be computed from

$$C_{2,j} = C_{1,j} + X_{1,j} \cdot C_{1,j-1} \qquad j \in \{1, 2, 3\}$$

Since addition of least significant bits of two inputs has no carry, we can say $C_{2,0} = C_{1,0}$. In steps S13-S19, it is shown how $C_{2,1}$ can be computed.

Steps S13: $C_{2,0}$ is read out using a *Read* instruction.

Steps S14: Using *Apply* instruction, ${C_{2,0}}'$ is computed by applying '1' as third wordline input and $C_{2,0}$ as third bitline input.

Steps S15: $X_{1,1}$ is read out from memory.

Steps S16-S17: $({X_{1,1}}' + {C_{2,0}}')$ is computed by applying '1' as third wordline input and $X_{1,1}$ as third bitline input in step S16. $({X_{1,1}}' + {C_{2,0}}')$ is read-out in step S17.

Steps S18-S19: $C_{1,1} + ({X_{1,1}}' + {C_{2,0}}')'$ is computed by applying '1' as second wordline input and $({X_{1,1}}' + {C_{2,0}}')$ as third bitline input in step S18. This output is equivalent to $C_{2,1}$. ReRAM devices corresponding to third wordline input are then reset in step S19 for next carry generation steps.

Consequently, these 7 steps (S13-S19) are repeated to generate remaining $C_{2,j}$ ($C_{2,2}, C_{2,3}$ in this case) for adjacent left-bit sum computation and resetting devices. From Fig. 5,

we can compute the final sum result $R = (R_4R_3R_2R_1R_0)_2$ as follows:

$$R_0 = X_{1,0}$$
$$R_j = X_{1,j} \oplus C_{2,j-1} \qquad j \in \{1,2,3\}$$
$$R_4 = 0 \oplus C_{2,3} = C_{2,3}$$

Three more instructions are required to read $X_{1,i}$ and $C_{2,i}$ ($C_{2,i}$ instead of $C_{2,j}$ since it includes $C_{2,0}$) from memory and reset devices corresponding to wordline 1 (except $X_{1,0}$). In total, $12 + (3 \times 7) + 3 + 9 = 45$ instructions are required in this case. The result will be available as five bits stored in DCM corresponding to first wordline. In general, an e-bit adder computation will require $24 + 7(e-1)$ instructions. Between subsequent addition operations, 4 more instructions are required to read out the sum result and reset all devices corresponding to the three wordlines. So a total of $2[24 + 7(e-1)] + 4$ instructions are required to compute the exponent (disregarding MMSBC update).

C. Computing Mantissa

The fractional number is $(k-1)$-bits long, which is normalized with a leading implied 1 to represent k-bits long mantissa. For demonstration of the computation, we consider a 4-bit mantissa ($k = 4$). Let the two operand mantissas be $M_a = a0 \bullet a1a2a3$ and $M_b = b0 \bullet b1b2b3$, where $a0 = a1 = 1$ and $\bullet$ represents decimal point. In Fig. 7, we show regular binary multiplication. In case of floating point representation, mantissa multiplication can take place similarly. However the resultant mantissa might need to be normalized depending on the value of *s0*. If *s0* is 0, Mantissa will be $(1 \bullet s1s2s3s4)_2$ else it will be $(1 \bullet s1s2s3)_2$.

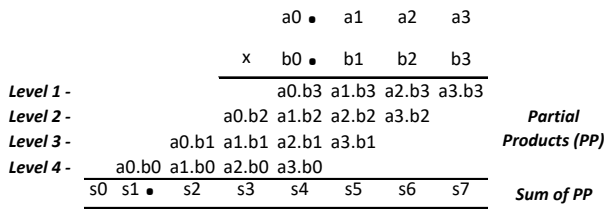

Fig. 7: 4-bit multiplication.

As evident from Fig. 7, three operations are needed to compute mantissa, namely, partial product calculation, sum of partial products and mantissa adjustment. All these operations are realized using $3(k+1) \times k$ ReRAM crossbar array. We demonstrate a 4-bit mantissa multiplication implemented with a (15×4) crossbar array (with 15 wordlines and 4 bitlines) as follows.

1) Computing Partial Products: In the context of multiplication of two input mantissa $M_a = a0 \bullet a1a2a3$ and $M_b = b0 \bullet b1b2b3$, a partial product can be represented as:

$$p_{i,j} = ai \cdot bj \qquad i,j \in \{0,1,2,3\}$$

From Fig. 7, we can see that 16 partial products have to be computed for 4-bit multiplication. As evident from Fig. 7 with 4-bit multiplication, we will have 16 partial dot products (each level contains 4 partial dot products and there are 4 levels in total, shown in Fig. 7). We have shown the computation of partial dot products for level 1 (with only wordline 1 from the set of 15 wordlines) in Fig. 8 and discuss the steps as follows.

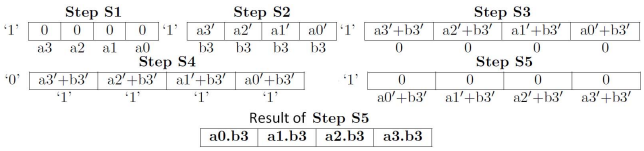

Fig. 8: Steps for 4-bit Partial Product Computation.

Step S1: Using *Apply* instruction, primary input bits from *a* are loaded in inverted form.

Step S2: Using *Apply* instruction, '1' is applied to the first wordline input and LSB (i.e *b3*) from primary input *b* is applied to all bitline inputs to compute first level partial products in inverted form, e.g. $(a0.b3)' = a0' + b3'$.

Step S3: Using *Read* instruction, data from devices corresponding to first wordline are read out and stored into DMR.

Step S4: Using *Apply* instruction, all bits of the first wordline are reset to zero.

Step S5: As shown in Fig 8, already read out first level partial products in inverted form are applied to the bitline inputs in reverse order and '1' is applied to the first wordline input to compute the required partial products and store them in memory locations corresponding to the first wordline.

The set of five steps are repeated in similar fashion with next set of bits (*b2*, *b1*, *b0*) from input *b* – three more times to compute and store all the remaining 12 partial dot products in memory locations corresponding to wordlines 2-4. This requires a total of $5 \times 4 = 20$ instruction cycles. In general with k-bit mantissa multiplication, it will require $5k$ cycles to compute all the partial products and store them in the devices corresponding to the first k-wordlines.

2) Computing Sum of Partial Products: The method to add the partial products of k-bit mantissa is explained here. Summation of multiple partial products can be performed by serial addition but that would lead to higher delay. Therefore, we use a fast adder construction that requires the carry to be propagated within same level only once in the final level. This adder construction can be mapped to the ReVAMP architecture using the steps illustrated as a flowchart in Fig. 10. We suppose $P_{i,j} = p_{3-i,3-j}$ and $R_m = s_{7-m}$. With this supposition, we can clearly relate how partial products shown in Fig. 7 are utilized in the adder implementation depicted in Fig. 9. For shifted partial product addition, there will be an additional carry generation level after the initial XOR and Carry generation steps and before final XOR operation. This will result in a $(k+1)$-bit XOR output and the MSB of 2-input sum result can be found in MSB of Carry generated from last second carry implementation step.

In Fig. 10, $XOR-2$ and $CARRY-2$ represents a 2-input k-bit XOR operation and 2-input k-bit carry generation respectively, together acts as a half adder. So, we have

$$XOR-2(a,b) = a \oplus b$$
$$CARRY-2(a,b) = a.b$$

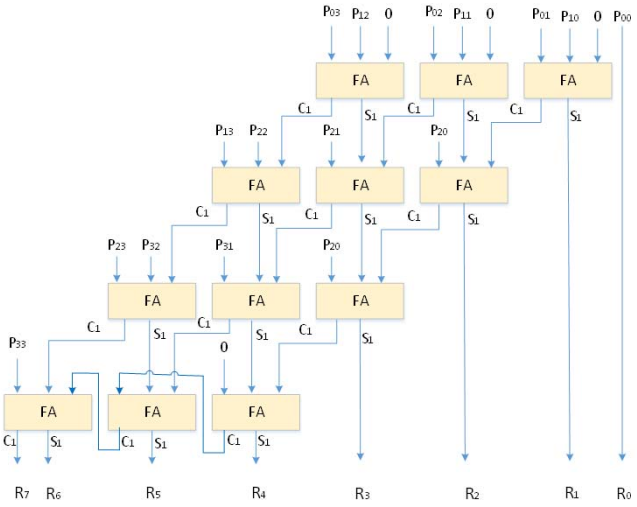

Fig. 9: 4-bit adder schematic to compute sum of partial products. FA represents a Full Adder and $\{S_1, C_1\}$ is $\{Sum, Carry\}$ generated by each FA. $P_{i,j} = p_{3-i,3-j}$ for $i, j \in \{0, 1, 2, 3\}$ and $R_m = s_{7-m}$ for $m \in \{0, 1, 2, 3, 4, 5, 6, 7\}$. Result $= (R_7R_6R_5R_4R_3R_2R_1R_0)_2$.

Similarly, computing $XOR-3$ and $CARRY-3$ components yield a 3-input k-bit XOR output and 3-input k-bit Carry generation, acts as a full adder. Thus,

$$XOR-3(a,b,c) = a \oplus b \oplus c$$
$$CARRY-3(a,b,c) = a.b + b.c + c.a$$

Boolean AND, OR and XOR can be implemented by using a set of instructions similar to approach shown in Fig. 6. Since k-bit multiplication generates $2k$-bit output, the sum of products needs to be copied to two k-bit words. With Manipulate Crossbar operation, the resultant sum of products is copied to DCM devices corresponding to wordline position $3(k+1)-3$ and $3(k+1)-2$ with its significant k-bits in devices corresponding to wordline position $3(k+1)-3$.

The number of instructions required to compute all these operations can be determined as follows. Computation of XOR-2 and CARRY-2 operations together require 19 cycles. From Fig. 10, it is evident that two sets of $XOR-2$, $CARRY-2$ will be required irrespective of the value of k. Therefore, 38 instructions will be executed. Depending on the value of k, there will be iterations of $XOR-3$ and $CARRY-3$ operations. If $k > 2$ (signifying that fractional part is more than one bit), $XOR-3$ and $CARRY-3$ cycles will be executed. The $XOR-3$ and $CARRY-3$ operations together need 41 cycles for complete computation. From Fig. 10, it can be interpreted that two rounds of $(k-2)$ iterations of $XOR-3$, $CARRY-3$ need to be implemented. So $2(k-2)*41$ instruction cycles will be required for this purpose. The 'Manipulate Crossbar' operation requires $5(k+1)$ instruction cycles in total. Therefore, the number of instructions add up to $87k-121$ for k-bit mantissa computation.

3) Mantissa Adjustment: The sum of the partial product will have $2k$ resultant bits available in the $3(k+1)-3$ and

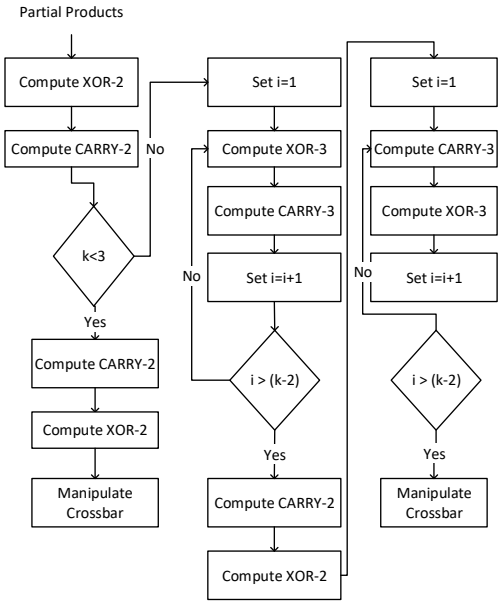

Fig. 10: Flowchart to compute sum of partial products on ReVAMP.

$3(k+1)-2$ wordline memory. Now, there needs to be a set of instructions which can decide whether mantissa should be considered right from MSB (or MSB-1) bit-position of the result depending on the value of MSB being 1 (or 0). We call this as Mantissa MSB Carry or MMSBC since it is the carry output of last sum (from sum of products computation) operation and the MSB of the result of this sum. MMSBC may be referred as $s0$, depicted in Fig. 7. In regard to Fig. 7, if MMSBC = 1, Mantissa is selected as $(1 \bullet s1s2s3)_2$, and if MMSBC = 0, Mantissa is selected as $(1 \bullet s2s3s4)_2$. In this case, 15 instructions are required to adjust mantissa of the resultant product. This instruction count remains same even if the bit-length of the input is different.

D. Exponent Update

We need to incorporate addition of MMSBC value to sum of the exponents E_R computed in subsection III-B. After MMSBC is available, a sum operation similar to the method explained in subsection III-B is adopted to compute the new resultant exponent $E_R = E_R + MMSBC$. Therefore, an additional 24+7(e-1)+4 cycles will be required to completely compute the Exponent. Moreover, an extra read instruction is needed to read MMSBC, totaling the number of instructions to $3[24+7(e-1)]+9$.

IV. Experimental Results And Analysis

The instruction sequence for multiplication of IEEE-754 compliant numbers has been developed using Matlab®. The correctness was verified by means of behavioural simulator for ReVAMP and also via device accurate simulations using Cadence Spectre® [10].

In order to compute latency of the proposed mapping, we assume a n-bit floating point number in IEEE-754 format that has a leading sign bit, e-bit exponent and $(k-1)$-bit fraction

and calculate the total number of instructions. The resultant sign computation needs 8 instructions, mantissa computation needs $92k - 106$ instructions and exponent computation requires $60 + 21e$ instructions. Each instruction requires one cycle effectively, except the first instruction which requires three cycles due to the 3-stage pipeline architecture. The number of instructions for 32-bit and 64-bit IEEE-754 floating point multiplication is reported in TABLE I.

TABLE I: Number of instructions for 32-bit and 64-bit IEEE FP multiplication on ReVAMP.

Operation		Number of Instructions	
		32-bit	64-bit
Sign		8	8
Mantissa	Partial Products	120	265
	Sum of Partial Products	1967	8306
	Adjustment	15	15
Exponent	Addition	150	192
	Update	78	99
Total		2338	8885

The DCM crossbar dimensions are (81×24) and (168×53) for 32-bit and 64-bit precision floating point multiplication, hence the corresponding crossbar size are $1.944\ Kb$ and $8.904\ Kb$ for respective precisions. The instruction size for the 32-bit and 64-bit are $160\ bits$ and $389\ bits$ respectively The overall instruction memory size is approximately $374.4\ Kb$ with 32-bit aligned access and $3457\ Kb$ with 64-bit aligned access.

As per ITRS report on emerging devices [11], the read/write cycle for ReRAM devices is projected to have a duration time of $1\ ns$ and a write cycle energy of $0.1\ fJ/bit$. The proposed mapping is estimated to achieve roughly 10×10^6 and 6×10^6 floating point operations per second (FLOPS) at the cost of $5.6\ pJ$ and $471\ pJ$ of energy consumption for 32-bit and 64-bit IEEE-754 multiplication. We should note that the current implementation is aimed at minimizing the number of devices used for multiplication and does not aim at parallelizing multiple floating point operations. In TABLE II, we report the throughput of the proposed floating point multiplication mapping with ReVAMP architecture.

TABLE II: Implementation Summary.

Implementation	Frequency (in GHz)		Throughput (in GFLOPS)	
	32-bit	64-bit	32-bit	64-bit
FPGA [12] (40nm Virtex-6)	0.25	0.25	0.010	0.005
GPU [12] (55nm)	1.3	1.3	0.002	0.001
CMOS ASIC [13] (250nm)	1.05	1.12	1.05	0.56
CMOS ASIC [13] (65nm)	1.05	1.12	1.05	0.56
CMOS ASIC [13] (90nm)	1.05	1.12	1.05	1.12
Using ReVAMP	1	1	0.010	0.006

It is to be noted that the throughput of one floating-point unit is considered for GPGPU and FPGA implementation and reported in TABLE II. A dual mode double precision floating point multiplier architecture based on ASIC implementation is proposed that can be configured to compute two single precision multiplications in parallel [13]. This concept was implemented to achieve an efficient resource sharing. A comparative analysis between GPU and FPGA implementations of matrix multiplications based on IEEE 754 floating point formulation is presented in [12]. The results showed that GPUs were suitable for larger matrix multiplications, whereas FPGAs ensured higher throughput for multiplication of smaller matrices.

V. Conclusion

In this work, an efficient mapping of IEEE-754 floating point number multiplication on ReVAMP architecture has been proposed. The implementation uses a novel multiplication scheme for computing the resultant mantissa. The mapping is estimated to achieve $6 \times 10^{-3} GFLOPs$ throughput, which is comparable to the performance of FPGA based floating point multiplication unit. The implementation has an overall memory footprint($\approx 3831\ Kb$) with a low energy footprint ($\approx 471\ pJ$). We plan to extend the work in the direction of realizing floating point BLAS operations on the ReVAMP architecture using the proposed mapping.

References

[1] R. Waser, R. Dittmann, G. Staikov, and K. Szot, "Redox-based resistive switching memories–nanoionic mechanisms, prospects, and challenges," *Advanced Materials*, vol. 21, no. 25-26, pp. 2632–2663, 2009.

[2] S. H. Jo, T. Chang, I. Ebong, B. B. Bhadviya, P. Mazumder, and W. Lu, "Nanoscale memristor device as synapse in neuromorphic systems," *Nano letters*, vol. 10, no. 4, pp. 1297–1301, 2010.

[3] P.-E. Gaillardon, L. Amarú, A. Siemon, E. Linn, R. Waser, A. Chattopadhyay, and G. De Micheli, "The Programmable Logic-in-Memory (PLiM) Computer," *DATE*, 2016.

[4] D. Bhattacharjee, R. Devadoss, and A. Chattopadhyay, "ReVAMP: ReRAM based VLIW architecture for in-memory computing," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, Mar. 2017, pp. 782–787.

[5] E. Linn, R. Rosezin, C. Kügeler, and R. Waser, "Complementary resistive switches for passive nanocrossbar memories," *Nature materials*, vol. 9, no. 5, pp. 403–406, 2010.

[6] "BLAS (Basic Linear Algebra Subprograms)," http://www.netlib.org/blas/, accessed: 2017-07-31.

[7] A. Siemon, S. Menzel, R. Waser, and E. Linn, "A complementary resistive switch-based crossbar array adder," *IEEE journal on emerging and selected topics in circuits and systems*, vol. 5, no. 1, pp. 64–74, 2015.

[8] D. Bhattacharjee, A. Siemon, E. Linn, and A. Chattopadhyay, "Efficient complementary resistive switch-based crossbar array booth multiplier," *Microelectronics Journal*, vol. 64, pp. 78–85, 2017.

[9] D. Bhattacharjee, F. Merchant, and A. Chattopadhyay, "Enabling in-memory computation of binary blas using reram crossbar arrays," in *Very Large Scale Integration (VLSI-SoC), 2016 IFIP/IEEE International Conference on*. IEEE, 2016, pp. 1–6.

[10] A. Siemon, S. Menzel, A. Marchewka, Y. Nishi, R. Waser, and E. Linn, "Simulation of TaOx-based complementary resistive switches by a physics-based memristive model," in *2014 IEEE International Symposium on Circuits and Systems (ISCAS)*, Jun. 2014, pp. 1420–1423.

[11] "Emerging Reseach Devices report, International Technology Roadmap for Semiconductors, 2013." [Online]. Available: https://www.semiconductors.org/clientuploads/Research_Technology/ITRS/2013/2013ERD.pdf

[12] U. I. Minhas, S. Bayliss, and G. A. Constantinides, "GPU vs FPGA: A Comparative Analysis for Non-standard Precision," in *Reconfigurable Computing: Architectures, Tools, and Applications*, ser. Lecture Notes in Computer Science. Springer, Cham, Apr. 2014, pp. 298–305.

[13] M. K. Jaiswal and H. K. H. So, "Dual-mode double precision / two-parallel single precision floating point multiplier architecture," in *2015 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, Oct. 2015, pp. 213–218.