ReVAMP : ReRAM based VLIW Architecture for in-Memory comPuting

Debjyoti Bhattacharjee*, Rajeswari Devadoss† and Anupam Chattopadhyay‡
School of Computer Science and Engineering, Nanyang Technological University, Singapore - 639798
Email: {debjyoti001*,rajeswari†,anupam‡}@ntu.edu.sg

***Abstract*— With diverse types of emerging devices offering simultaneous capability of storage and logic operations, researchers have proposed novel platforms that promise gains in energy-efficiency. Such platforms can be classified into two domains—application-specific and general-purpose. The application-specific in-memory computing platforms include machine learning accelerators, arithmetic units, and Content Addressable Memory (CAM)-based structures. On the other hand, the general-purpose computing platforms stem from the idea that several in-memory computing logic devices do support a universal set of Boolean logic operation and therefore, can be used for mapping arbitrary Boolean functions efficiently. In this direction, so far, researchers have concentrated on challenges in logic synthesis (e.g. depth optimization), and technology mapping (e.g. device count reduction). The important problem of efficient technology mapping of arbitrary logic network onto a crossbar array structure has been overlooked so far. In this paper, we propose, ReVAMP, a general-purpose computing platform based on Resistive RAM crossbar array, which exploits the parallelism in computing multiple logic operations in the same word. Further, we study the problem of instruction generation and scheduling for such a platform. We benchmark the performance of ReVAMP with respect to the state of the art architecture.**

I. Introduction

The fast decline of Moores law is paving the way for a new set of emerging technology devices that offer improved reliability, performance, endurance and energy-efficiency. A few of these devices, notably ReRAM, can function as both logic and storage elements, and have caught the attention of system designers as the basis for multiple innovative and efficient platforms. On one hand, these platforms have been used for application-specific designs like neuromorphic computing [1], [2], arithmetic circuits [3], [4] and other logic circuits such as comparators, etc. On the other hand, it has been shown that ReRAM-based logic devices can realize an universal set of Boolean functions, and therefore, could be used to develop general-purpose programmable platforms [5], [6], akin to what has been reported earlier for Carbon Nanotubes [7]. In this paper, we focus on general-purpose programmable platforms using ReRAM-based crossbar array structures.

Efficient mapping solutions are crucial for efficient computing platforms. Multiple recent works to this end have proposed heuristic logic synthesis and technology mapping solutions. However, none have addressed parallelizing computations on a crossbar array. This results in under utilization of the crossbar array structure, which inherently allows parallel computation in each wordline. We address this key challenge in this paper.

Ideas about computation using ReRAM devices have grown over the years. Lehtonen et. al [8] presented a preliminary methodology for computing Boolean functions using memristive devices. Later, Poikonen et.al [9] showed that any Boolean expression can be computed using two working memristors which realize material implication. Logic synthesis and technology mapping solutions for memristors have been presented in [10]–[13].

Gaillardon et. al [5] have proposed Programmable Logic-in-Memory (PLiM)—the state of the art programmable architecture based on a ReRAM crossbar array, designed as a wrapper around a standard memory array that can still be operated as a regular memory array. It uses a single instruction identical to the native majority function computed by ReRAM devices. A compiler for the same was presented later in [14]. Despite the availability of a large amount of memory cells that can be exploited to perform operations in parallel, the PLiM architecture allows only sequential computations, thus under-utilizing the crossbar array. In this paper, we present a custom computing architecture using logic in memory operations on ReRAM crossbar arrays that supports VLIW instructions to exploit parallel computations in each word of the memory array. In particular, we make the following contributions:

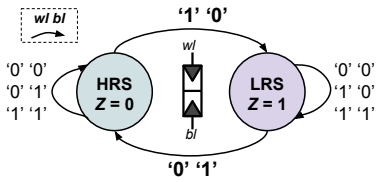
- We propose ReVAMP, a general-purpose programmable platform that allows for VLIW-like instruction-set and systematically handles the parallelization of computation on ReRAM crossbar array structures.
- Starting from a technology-independent logic network, we propose a multi-phase framework of heuristics to efficiently harness computation parallelism for ReVAMP.
- Through rigorous benchmarking, we establish significant improvement in runtime (cycles) compared to PLim, the state-of-the-art general purpose architecture using ReRAM arrays.

The rest of the paper is organized as follows. In section II, the fundamentals of logic operation using ReRAM crossbar arrays are presented. In section III, the ReVAMP architecture and its instruction set is presented. In section IV, a method for generating instructions for the ReVAMP architecture is presented. Section V presents the results of benchmarking the proposed architecture.

II. Logic operation using ReRAM crossbar array

A ReRAM crossbar memory consists of multiple 1S1R ReRAM devices, arranged in the form of a crossbar array [15].

The ReRAM array is programmed using a V/2 scheme. Logic 1 and 0 are realized by voltage pulses of 2.4 V and -2.4 V respectively. The details of the ReRAM device model can be found in [16]. Unselected lines are kept grounded. In a readout phase, the presence of a 5 μA current implies logic 1 while its absence implies logic 0. For logic operations, the device



wl	bl	Z_n
'0'	'0'	Z
'0'	'1'	0
'1'	'0'	1
'1'	'1'	Z

Fig. 1: *Logic operation using 1S1R device - represented as (a) FSM (b) Truth Table.*

can be interpreted as a finite-state-machine (FSM), as shown in Fig. 1. Each device has two input terminals—the wordline wl and bitline bl. The internal resistive state Z of the ReRAM acts as a third input and the stored bit. As shown in following equation, the next state of the device Z_n is expressed as a 3-input majority function, with the bitline input inverted.

$$Z_n = M_3(Z, wl, \overline{bl}) \tag{1}$$

This forms the fundamental logic operation that can be realized using ReRAM devices. The inversion operation can be realized using the intrinsic function Z_n. Since majority and inversion operations form a *functionally complete* set, any Boolean function can be realized using only Z_n operations. Like conventional RAM arrays, ReRAM memories are accessed as words. It should be noted that all the devices in a word share a bitline. In the following section, we introduce a ReRAM based general purpose architecture that systematically exploits computation parallelism within a word effectively.

III. REVAMP ARCHITECTURE

In this section, we present the **Re**RAM based **V**LIW **A**rchitecture for in-**M**emory com**P**uting (ReVAMP), Illustrated in Fig. 2, the architecture uses two ReRAM crossbar memories—the Instruction Memory (IM) and the Data Storage and Computation Memory (DCM). The IM is a regular instruction memory accessed using the program counter (PC). The DCM hosts data and in-memory computation. Even though it is possible to update multiple words of the DCM in parallel, the necessary control circuitry would be complex and costly in terms of area. Thereby, we choose to limit the architecture to updating the devices in one word of the DCM simultaneously, with each update operation being the intrinsic Z_n function. Since multiple Z_n operations operate in parallel, the proposed architecture is VLIW in nature. Splitting the instruction and data memory allows reduction in overall execution time, by parallelizing instruction fetch and computation. The ReVAMP architecture is parameterized as shown in Table I, and can be configured as necessary.

The ReVAMP architecture has a three-stage pipeline with instruction fetch (IF), instruction decode (ID) and execute (EX) stages. In the IF stage, the instruction at the address held by the Program Counter (PC) is fetched from the IM and loaded into the instruction register (IR). before the PC is updated. In the ID stage, the instruction is read from IR to determine the control inputs for the source select multiplexer, crossbar interconnect and the write circuit.

TABLE I: Architecture parameters

Parameter	Description
S_D	Number of words in the DCM
w_D	Number of bits in a word in DCM
S_I	Number of word in IM
w_I	Number of bits in a word in IM
$p(= w_D)$	Number of primary input lines

The data memory register (DMR) stores the data read out from the DCM. The primary input register (PIR) buffers the primary input data. Both DMR and PIR are w_D bits wide. Depending on the control input M_c, the source select multiplexer selects either DMR or the PIR as the data source. Thereafter, the crossbar-interconnect is used to generate the wordline and w_D number of bitline inputs by appropriate permutation the input data, as per the control signals stored in C_c. The crossbar-interconnect interconnect is basically a set of multiplixers, one per output, which selects on of the input w_D bits. The write circuits reads the value of the target wordline from the W_c register and the output of the crossbar-interconnect to determine and apply the inputs to the row and column decoder of the DCM.

A. ReVAMP Instruction Set

The ReVAMP architecture supports two instructions—*Read* and *Apply*, in the formats shown in Fig. 3. The *Read* instruction reads the word at the address **wl** from the DCM and stores it in the DMR. Now available in the DMR, this word can be used as input by the next instruction.

The *Apply* instruction is used for computation in the DCM. The address **w** specifies the word in the DCM that will be computed upon. A bit flag **s** chooses whether the inputs will be from primary input (PIR) or DMR. A two bit flas **ws** specifies the worline input — 00 selects Boolean 0, 01 selects Boolean 1 and 11 selects input specified by the **wb** flag. The **wb** bits are used to specify the bit within the chosen data source for use as wordline input. Pairs **(v,val)** pairs are used to specify bitline inputs. Bit flag **v** indicates if the input is NOP or a valid input. Similar to **wb**, bits **val** specifies the bit within the chosen data source for use as bitline input.

Read	w

Apply	w	s	ws	wb	(v val_1)	(v val_2) ... (v val_{w_D})

Fig. 3: ReVAMP instruction format

In each instruction, one bit is required to specify the opcode, and $log_2(S_D)$ bits are required to select the word. Each (v val) pair requires one bit for the r flag and $log_2(w_D)$ bits for specifying the bit in the selected input source. The field wb also requires $log_2(w_D)$ bits. Thus, the lengths of these

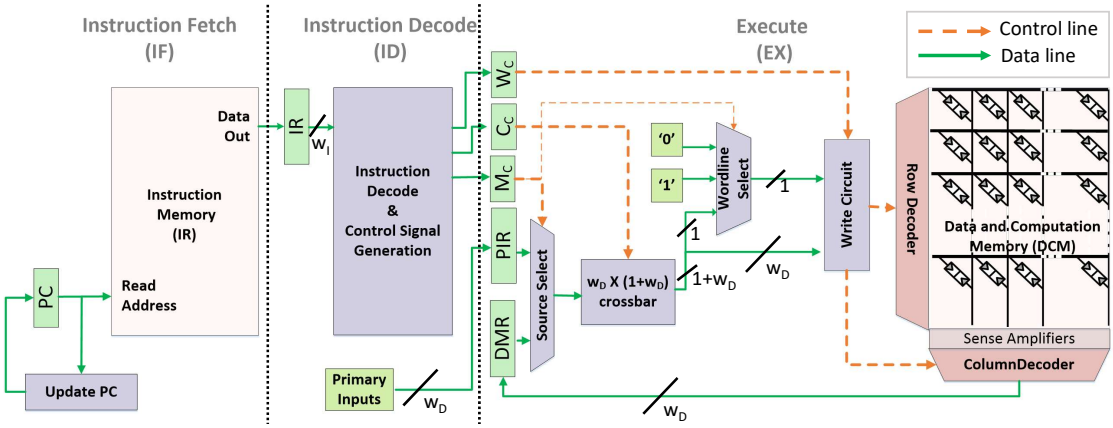


Fig. 2: *ReVAMP architecture*

instructions are:

$$IL_{Read} := 1 + log_2(S_D) \quad (2)$$

$$IL_{Apply} := 2 + log_2(S_D) + w_D(1 + log_2(w_D)) \quad (3)$$

The word length w_I of the IM should be a power of 2 greater than or equal to $max(IL_{Read}, IL_{Apply})$.

In contrast to the ReVAMP instruction set, the PLiM computer uses a single instruction, RM_3. Each instruction computes a 3-input majority function and requires 9 cycles to complete. Even though the PLiM instruction is shorter than those of ReVAMP, the total number of instructions required is much higher than for ReVAMP, since ReVAM's VLIW instructions pack multiple majority operations. In the next section, we present a multi-phase instruction generation method that identifies majority operations that can be computed in parallel and packs them into VLIW instructions.

IV. Instruction Generation for ReVAMP Architecture

In this section, we present a methodol to generate instructions for the ReVAMP architecture that exploits the word-level parallelism offered by it.We choose the Majority-Inverter Graph (MIG) [17] to represent logic. Since the proposed architecture is capable parallelism of a certain type, a method for mapping of Boolean function to the architecture effectively to harness the parallelism is necessary. We propose an algorithm to this effect with four phases—assignment of host and inputs to nodes, grouping nodes to blocks, packing blocks to words and finally generation and scheduling of instructions.

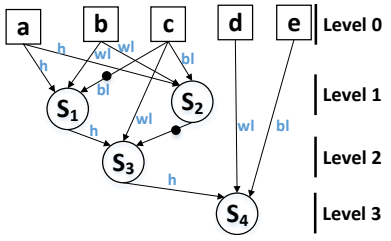


Fig. 4: *A Majority Inverter Graph*

A. Logic representation

Since ReRAM devices inherently compute 3-input Boolean majority with one input inverted, the MIG is apt for logic representation. A Boolean function can be represented as an MIG, where each node represents a Majority-3 (M_3) function and a directed edge $i \rightarrow j$ exists if the output of the (*parent*) node i is an input to the (*child*) node j. Each edge is marked as either regular or inverted. A Primary Input (PI) node is either a logic constant $0/1$ or a Boolean variable. If a node is not a PI, then it is an *internal* node. A Primary Output (PO) node represents the output of the function. We define the $level$ of a node as the length of the longest path from a PI to the node. The level of PI nodes is zero.

Example 1. *For the MIG in Fig. 4, a, b, c, d and e are the primary inputs. S_1, S_2 and S_3 are the internal nodes while S_4 is a primary output.*

B. Assign Host and Inputs to Nodes

A ReRAM device has an internal state Z, and two inputlines—the wordline and bitline. A computation on it updates its internal state Z, in effect making the device the $host$ for the computation. For each node in an MIG, one of its parents hosts its computation and the remaining parents act as wordline and bitline inputs. The computations of multiple independent nodes can be grouped into an Apply instruction if they have a common operand. Based on this, we present a few rules to assign the host and inputs of the nodes of an MIG.

- If a node has multiple children in the same level, then it can be used as common wordline input for computing those nodes. For instance, in Fig. 4, input b can be used as common wordline input to compute S_1 and S_2.
- If an incoming edge to a node is marked inverted, then the corresponding parent can be used as the bitline input. In Fig. 4, c and S_2 are used as bitline inputs to compute S_1 and S_3 respectively.
- If there are no inverted incoming edges to a node, then a negated parent is used as input to that node. For node S_2 in Fig. 4, input $\bar{c}$ is used as bitline input.

- The remaining parent is used as host for the node. The nodes a and S_1 act as $host$ to compute S_1 and S_3 respectively in Fig. 4.

Based on these rules, we can determine host and inputs for each internal node in a level of the MIG such that the nodes with common inputs can shared wordline inputs. This is used for scheduling computation of the MIG. We mark this assignments on the edges of the MIG.

C. Group Nodes to Blocks

To compute an internal node in a MIG, we need to read out the wordline and bitlines inputs of the node and then apply these inputs to the host. Given that only a single word can be read out in a clock cycle, the wordline and bitline inputs of the node must reside on the same wordline to allow efficient computation of the node. This creates a constraint that for each node in an MIG, the wordline and bitline inputs should be placed in the same word. We call this grouping a *block*.

Further, as read-outs are non-destructive, blocks can be merged if they have common inputlines. This reduces the number of devices required, with the merged block having only one copy of the common inputline. Note that blocks can be merged only if the result does not exceed the word length.

Also, a pair of blocks in the same level that have hosts which share a wordline input should be merged. This host-based merge along with merge of the corresponding blocks with the inputlines of these hosts would allow computation of the nodes in level with shared wordline in a single cycle, reducing delay.

Algorithm 1: Block Formation Algorithm

```
   Data: G, pi, po
   Result: blockList
1  blockCount = 0;
2  for node_out ∈ po do
3      addBlock([(node_out.host)],blockCount);
4      addBlock([(node_out.wl, node_out.bl)],blockCount);
5      addInversionBlock(el);
6  mergeBlock();
7  for l = level_max; l > 0; l = l − 1 do
8      for block ∈ blockList do
9          for el ∈ block do
10             if el ∉ pi and el.level == l then
11                 replace(el, el.host);
12                 addBlock([el.wl, el.bl], blockCount);
13                 addInversionBlock(el);
14     mergeBlock();
```

The algorithm of block formation is shown in Algo. 1. In Lines 2-5 creates blocks considering the placement contraint on the input-lines of the output nodes. The addInversionBlock method adds the positive nodes as blocks to the blockList, if the added blocks have inverted values. Only a single positive node is added to blockList, corresponding to multiple copies of a negated node. The mergeBlock method merges blocks based on the inputline and host based merge constraints. The replace method replaces a node in a block with its host node.

Example 2. *For a word length (w_D) of 3, Table II shows the working of the block formation algorithm on the MIG of Fig. 4. Starting at the output node, blockList has a single block. At level 3, node S_4 is replaced with its host and inputlines. Since these two blocks do not have any common inputlines or hosts, they cannot be merged. At level 2, node S_3 gets replaced and the inputlines are added to a new block. At level 1, nodes S_1 and S_2 are replaced by their hosts a, and the inputlines are inserted in two new blocks. Blocks 4 and 5 have a common inputline b and are hence merged. Blocks 2 and 4 have common inputs, but cannot be merged as the length (four) of the resultant block will exceed the given word length. Thereafter, since the two a host nodes have the same wordline, blocks 1 and 3 get merged, but both copies of the host are retained, using the host-merge constraint.*

TABLE II: BlockList update with BlockMerge algorithm for MIG of Fig. 4

Level	BlockList
Output	[[1,S_4)]]
3	[[(1,S_3,h)],[(2,d,i),(2,$\overline{e}$,i)]]
2	[[(1,S_1,h)],[(2,d,i),(2,$\overline{e}$,i)],[(3,c,i),(3,S_2,i)]]
1	[[1:(a,h)],[2:(d,i),($\overline{e}$,i)],[3:(c,i),(a,h)],[4:(b,i),(c,i)],[5:(b,i),($\overline{c}$,i)]]
1	[[1:(a,h)],[2:(d,i),($\overline{e}$,i)],[3:(c,i),(a,h)],[4:(b,i),(c,i),($\overline{c}$,i)]]
1	[[1:(a,h),(c,i),(a,h)],[2:(b,i),($\overline{e}$,i)],[4:(b,i),(c,i),($\overline{c}$,i)]

D. Pack Blocks in Words

At the end of scheduling computation, we have blocks of elements, which have to be placed in the same wordline. The number of elements in each block is less than or equal to w_D, the number of bits in a word. Now, these blocks have to packed in the DCM using the minimum number of words. The problem can be formulated as a bin packing problem as defined below.

Algorithm 2: First-fit Algorithm

```
   Data: blockList,wordlength
   Result: blockToWord
1  wordToBlock = HashMap();
2  wc = 0;
3  for block ∈ blockList do
4      assigned = False;
5      for w ∈ wordToBlock do
6          if wordToBlock[w].occupied + block.length < wordlength
           then
7              wordToBlock[w].occupied =wordToBlock[w].occupied +
               block.length;
8              wordToBlock[w].append(block);
9              assigned = True;
10         if assigned == False then
11             wc = wc+1;
12             wordToBlock[wc].append(block)
               wordToBlock[wc].occupied = block.length;
```

Consider each word in DM as a bin, with capacity w_D. Each block b_i has a value v_i, $v_i > 0$. Each block must be assigned to a bin such the total value of the objects assigned to the bin is less than or equal to w_D. The objective is to minimize the number of bins required to assign all the block, without violating the capacity constraint.

This *first-fit* algorithm provides a 2-factor approximation, that is the number of words required by the algorithm is atmost twice the number of words required by the optimal solution.

Example 3. *For the example, the blocks determined by the Block Formation algorithm are placed in a separate wordline, as shown in Fig. 5(i).*

E. Generation and Scheduling instructions

The primary inputs have to be loaded into the DCM before computation of the internal nodes of the MIG can begin. In each clock cycle, w_D primary inputs can be read. The primary inputs are loaded via the bitline and hence the inverted values are stored in a single clock cycle. To store non-inverted primary inputs, the primary inputs are written to a wordline, thereby storing it in inverted form. Then, the inverted value is read out and applied via the bitline to store the non-inverted value to the required wordline. A single extra wordline is used for storage of the intermediate inverted primary input, and this wordline is reset, after each use.

All the nodes in level i are scheduled for computation before any node at level $i+1$ is scheduled. The nodes in the same level can be scheduled in any order as they do not have any data dependencies. The nodes in a level with hosts of the which are in the same block, and the corresponding inputlines are also placed together in the same block, are scheduled for computation together. Once all the nodes in a level have been computed, we determine whether any inverted copies of the nodes are required for computation of nodes are greater level. If inverted copies are needed, the node is read out and stored in inverted form in the required block by writing through the bitline. Each computation is expressed as a **Apply** instruction and read operations are expressed as **Read** instruction.

TABLE III: Instruction sequence to compute MIG in Fig. 4.

I_1	Read 2
I_2	Apply 0 1 0 1 1 0 0 1 2
I_3	Read 0
I_4	Apply 0 1 1 1 2 0 0 0 0
I_5	Read 1
I_6	Apply 0 1 0 1 1 0 0 0 0

Example 4. *Table III shows the sequence of instructions used to compute the example MIG, and Fig. 5 shows the changes in DCM state on application of the **Apply** instructions. Note that the additional instructions needed to initialize the DCM are not shown. The inputs to compute nodes S_1 and S_2 are in word 2 and are read out. The hosts of nodes S_1 and S_2 are in word 0, and therefore I_2 computes these nodes in word 0. The inputs to compute S_3 are in word 0, and are read out by I_3. I_4 computes S_3 in host S_1. Finally to compute S_4, I_5 reads out word 1 and I_5 applies the required inputs to S_3.*

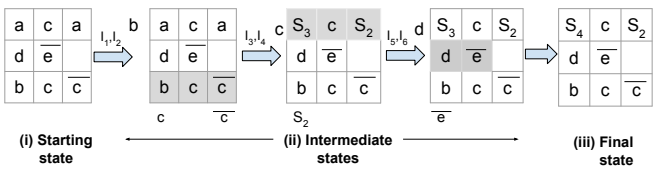


Fig. 5: DCM state transition during computation. The gray wordline represent the read out wordline.

V. Experimental Results

We have implemented the proposed compilation flow for the ReVAMP architecture.The algorithm was evaluated using the EPFL benchmarks[1]. and the results are presented in Table IV for word length (w_D) of 16-bits. For most of the benchmarks, the compilation time to generate the instructions, was a few seconds while for the larger benchmarks, compilation completed under 20 minutes.

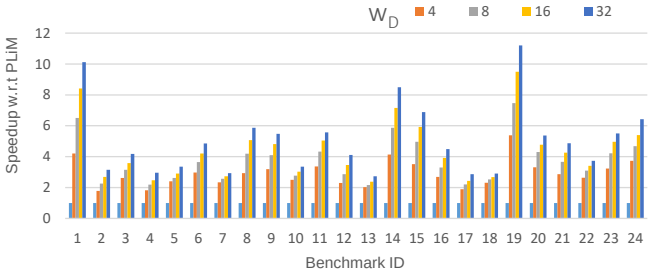


Fig. 6: *Impact of word length on word utilization*

The total delay (D_{total}) of the mapping solution is the number of cycles to complete computation of the benchmark by the ReVAMP architecture. The minimum number of cycles D_{P*} required by PLiM to compute any MIG is 9#N, where #N is the number of nodes in MIG. For each benchmark, D_{total} is significantly lower than the D_{P*} achieved by the PLiM computer. This is a fair comparison since PLiM computer also used a word-length of 16-bits. The ReVAMP architecture outperforms PLiM computer for the same 16-bit word by a factor of 4.38 on average and 9.5 at the maximum. For the ReVAMP architecture, on average, almost 30% of the computation time is spent in computing negated value of the nodes. Thus, the ReVAMP architecture would further outperform the PLiM computer, when the actual number of cycles required by PLiM computer for computation with negations will be considered.

In Fig 6, the speed up achieved by our proposed ReVAMP architecture against the PLiM computer is presented for various word lengths. Even for a small word length of 4, the ReVAMP architecture gains in performance over the PLiM architecture by a factor of 2.9 on average. This shows that harnessing the inherent parallelism of ReRAM crossbar arrays for computation provides considerable performance gains, justifying the VLIW nature of the ReVAMP architecture.

The proposed ReVAMP architecture uses only Apply instructions for loading the initial DCM state. Thereafter, each computation requires operands from the DCM. Hence, the number of Read instructions (I_R) is comparable to the number of Apply instructions (I_A). The total number of instructions required by the mapping is shown in column I_{total}.

The number of words (#W) used determines the area of the mapping solution. To determine the effectiveness of the packing algorithm to pack blocks into words, we introduce the word utilization (W_{Util}) metric, which is the percentage of total number of bits in #W words, that are used by the mapping solution. For the example MIG, out of 9 bits (3 words, each

[1]http://lsi.epfl.ch/benchmark

TABLE IV: Performance of the ReVAMP architecture on EPFL benchmarks for w_D=16.

ID	Benchmark	#N	I_A	I_R	I_{total}	#B	#W	W_{Util}	D_{total}	D_{P*}
1	ac97_ctrl	15253	8803	7520	16323	2330	933	99.88	16325	137277
2	comp	18967	32297	31293	63590	3114	1182	99.96	63592	170703
3	des_area	4629	5971	5639	11610	1073	305	99.92	11612	41661
4	div16	4440	8375	7825	16200	702	342	99.96	16202	39960
5	hamming	2280	3603	3450	7053	623	180	99.58	7055	20520
6	i2c	1263	1450	1247	2697	297	83	99.32	2699	11367
7	MAC32	9489	15980	15363	31343	3045	784	99.85	31345	85401
8	max	4854	4431	4184	8615	990	314	99.2	8617	43686
9	mem_ctrl	9569	9497	8405	17902	2032	625	99.65	17904	86121
10	MUL32	9226	14047	13389	27436	2406	718	99.28	27438	83034
11	pci_bridge32	25653	23826	21914	45740	6535	1546	99.82	45742	230877
12	pci_spoci_ctrl	1096	1523	1328	2851	196	85	99.04	2853	9864
13	revx	7563	14575	14004	28579	1703	611	100	28581	68067
14	sasc	889	602	514	1116	170	57	99.01	1118	8001
15	simple_spi	1135	930	794	1724	209	75	99.17	1726	10215
16	spi	3890	4615	4301	8916	885	267	99.98	8918	35010
17	sqrt32	2206	4216	3948	8164	442	170	99.85	8166	19854
18	square	18080	30988	29880	60868	5640	1454	99.78	60870	162720
19	ss_pcm	604	313	257	570	97	31	98.59	572	5436
20	systemcaes	11299	11100	10229	21329	2602	721	99.84	21331	101691
21	systemcdes	3028	3312	3090	6402	627	195	99.49	6404	27252
22	tv80	8177	11219	10368	21587	1672	578	99.73	21589	73593
23	usb_funct	16704	16054	14269	30323	2943	1057	99.77	30325	150336
24	usb_phy	599	538	460	998	140	37	97.47	1000	5391

with 3 bits), 8 bits are used and therefore W_{Util} is 88.8%. The proposed packing algorithm achieves more than 97% utilization for all the benchmarks, when w_D = 16, including 100% utilization for the *revx* benchmark. Fig. 7 shows that with increase in word length from 4 to 8, leads to considerable improvement in W_{Util}. However, the W_{Util} is comparable for the word lengths, 8, 16 and 32, and approaches 100%. This shows the effectiveness of packing the blocks into words.

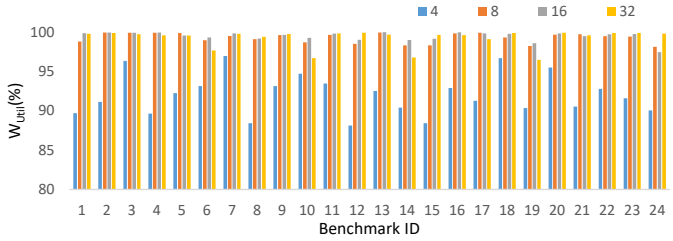


Fig. 7: *Impact of word length on word utilization*

VI. Conclusion and Future Work

We have proposed a novel general purpose computing architecture using ReRAMs. It is the first ReRAM-crossbar based general purpose computing architecture that harnessess the inherent parallelism of the crossbar arrays. This is based on the observation that computations in same word can be executed in parallel, under certain crossbar constraints. This is enabled by the instruction set of the proposed architecture which packs multiple computations into a single instruction.

We have also proposed a 4-phase technology mapping methodology that identifies parallelism suitable for the proposed architecture and generates VLIW instructions to exploit the parallelism. We have demonstrated the performance of the architecture in terms of delay, number of words and word utilisation on a large benchmark set. In addition, we have presented a detailed comparsion with the PLiM architecture, which is also based on ReRAMs and shown considerable gains in performance by the ReVAMP architecture.

The currently proposed heuristics can be improved further. For example, the choice of host and input lines for nodes are currently based on local information of the nodes in the MIG. Also in this work, we did not address the problem of logic synthesis. Since the architecture relies on multiple nodes sharing a common wordline, logic synthesis techniques for generation of MIG from Boolean logic need to be developed to maximize the number of common inputs.

References

[1] K.-H. Kim, S. Gaba, D. Wheeler, J. M. Cruz-Albrecht, T. Hussain, N. Srinivasa and W. Lu, "A functional hybrid memristor crossbar-array/cmos system for data storage and neuromorphic applications," *Nano Letters*, vol. 12, no. 1, pp. 389–395, 2011.

[2] D. B. Strukov, D.R. Stewart, J. Borghetti, X. Li, M. Pickett, G. M. Ribeiro, W. Robinett, G. Snider, J. P. Strachan, W. Wu, Q. Xia, J. J. Yang and R. S. Williams, "Hybrid cmos/memristor circuits," in *Proceedings of 2010 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1967–1970, 2010.

[3] A. Siemon, S. Menzel, R. Waser, and E. Linn, "A complementary resistive switch-based crossbar array adder," *Emerging and Selected Topics in Circuits and Systems, IEEE Journal on*, vol. 5, no. 1, pp. 64–74, 2015.

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

[5] P. E. Gaillardon, L. Amarú, A. Siemon, E. Linn, R. Waser, A. Chattopadhyay, and G. D. Micheli, "The programmable logic-in-memory (plim) computer," in *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 427–432, March 2016.

[6] S. Hamdioui, L. Xie, H. A. D. Nguyen, M. Taouil, K. Bertels, H. Corporaal, H. Jiao, F. Catthoor, D. Wouters, L. Eike, *et al.*, "Memristor based computation-in-memory architecture for data-intensive applications," in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, pp. 1718–1725, EDA Consortium, 2015.

[7] M. M. Shulaker, G. Hills, N. Patil, H. Wei, H.-Y. Chen, H.-S. P. Wong, and S. Mitra, "Carbon nanotube computer," *Nature*, vol. 501, no. 7468, pp. 526–530, 2013.

[8] E. Lehtonen and M. Laiho, "Stateful implication logic with memristors," in *NanoArch*, pp. 33–36, IEEE Computer Society, 2009.

[9] J. H. Poikonen, E. Lehtonen, and M. Laiho, "On synthesis of Boolean expressions for memristive devices using sequential implication logic," *IEEE TCAD*, vol. 31, no. 7, pp. 1129–1134, 2012.

[10] A. Raghuvanshi and M. Perkowski, "Logic Synthesis and a Generalized Notation for Memristor-Realized Material Implication Gates," *ICCAD*, pp. 470–477, 2014.

[11] D. Bhattacharjee and A. Chattopadhyay, "Delay-optimal technology mapping for in-memory computing using reram devices," in *Proceedings of the 35th International Conference on Computer-Aided Design*, p. 119, ACM, 2016.

[12] S. Shirinzadeh, M. Soeken, P.-E. Gaillardon, and R. Drechsler, "Fast Logic Synthesis for RRAM-based In-Memory Computing using Majority-Inverter Graphs," in *DATE*, 2016.

[13] D. Bhattacharjee, A. Easwaran, and A. Chattopadhyay, "Area-constrained technology mapping for in-memory computing using reram devices," in *22nd Asia and South Pacific Design Automation Conference, ASP-DAC 2017*, 2017.

[14] M. Soeken, S. Shririnzadeh, P.-E. Gaillardon, L. Amarú, R. Drechsler, and G. D. Micheli, "An mig-based compiler for programmable logic-in-memory architectures," in *DAC*, 2016.

[15] E. Linn, R. Rosezin, S. Tappertzhofen, U. Böttger and R. Waser, "Beyond von neumann-logic operations in passive crossbar arrays alongside memory operations," *Nanotechnology*, vol. 23, no. 30, 2012.

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

[17] L. Amarú, P.-E. Gaillardon, and G. De Micheli, "Majority-inverter graph: A novel data-structure and algorithms for efficient logic optimi tion," in *Proceedings of the 51st Annual Design Automation Conference*, pp. 1–6, ACM, 2014.