Chapter 13
In-Memory Data Compression Using ReRAMs

Debjyoti Bhattacharjee and Anupam Chattopadhyay

The fast decline of Moore's law is paving the way for a new set of emerging technology devices that offer improved reliability, performance, endurance, and energy-efficiency. Resistive Random Access Memories (ReRAMs) have emerged as one of the most promising technologies for logic and memory applications [1]. ReRAMs are non-volatile, ultra compact memories with low leakage power and high endurance. Large passive crossbar arrays can be realized by means of devices such as a select device in series to a switch (1S1R) or a Complementary Resistive Switch (CRS), to prevent parasitic currents [2]. 1S1R-based devices offer non-destructive readout, unlike CRS-based devices in which readouts are destructive, which makes 1S1R devices suitable for implementation of logic.

Internet-of-things (IoT) is an umbrella term encompassing a wide range of applications and diverse devices, that generally share two common characteristics—connectivity and low energy requirements. Irrespective of the specific network topology, bit rates and communication standards, a considerable amount of energy budget of the IoT nodes is allocated for communication. The energy consumed by the communication sub-system is more or less directly proportional to the amount of data transmitted or received. Therefore, it is of paramount importance to compress the data before transmission.

LZ77 is a lossless compression technique, introduced by Abraham Lempel and Jacob Ziv [3]. LZ77 along with LZ78 forms the basis for multiple variations such as

D. Bhattacharjee (✉)
Hardware and Embedded Systems Laboratory, School of Computer Science and Engineering, Nanyang Technological University, Singapore 639798, Singapore
e-mail: debjyoti001@ntu.edu.sg

A. Chattopadhyay
School of Computer Science and Engineering, School of Physical and Mathematical Sciences, Nanyang Technological University, Block N4 Nanyang Avenue #02c-105, Singapore 639798, Singapore
e-mail: anupam@ntu.edu.sg

© Springer International Publishing AG 2017

A. Chattopadhyay et al. (eds.), *Emerging Technology and Architecture for Big-data Analytics*, DOI 10.1007/978-3-319-54840-1_13

LZW, LZSS, LZMA, and others. In addition, it forms the core of several ubiquitous compression schemes such as GIF, DEFLATE, etc. LZ77 was awarded as an IEEE Milestone in 2004.

This chapter is devoted to the introduction of an in-memory computing architecture using ReRAM crossbar arrays and how various functions can be realized using the architecture. We demonstrate a low-area implementation of LZ77 compression algorithm using the ReRAM based in-memory architecture. In Sect. 13.2, the **Re**RAM based **V**LIW **A**rchitecture for in-**M**emory com**P**uting (ReVAMP) architecture is introduced. In Sects. 13.2.1 and 13.2.2, realization of identity comparator and priority multiplexer is presented using ReVAMP. In Sect. 13.3, we present the details of compression using LZ77 algorithm on ReVAMP and the performance of the proposed implementation is analyzed. Section 13.5 presents a review of the existing works in the domain of in-memory computing using ReRAMs.

13.1 LZ77 Compression Algorithm

LZ77 is a lossless compression algorithm that forms the basis of multiple other compression algorithms [3]. In LZ77, compression is achieved by replacing repeated occurrences of data with reference to a single copy of that data that existed earlier in the uncompressed data stream. Such a match is encoded as a *length-distance* pair of numbers. This implies that the next *length* number of characters match the characters at *distance* characters behind it in the uncompressed scheme. The term *length* is also referred to as *offset*. The pseudo-code for LZ77 compression is shown in Algorithm 1.

LZ77 uses a sliding window data structure to find matches. The sliding window is divided into two parts, namely the Look-ahead buffer and the Dictionary buffer. The Dictionary buffer stores the most recent uncompressed data stream that is used to look for matches. The Look-ahead buffer contains the uncompressed data stream that is yet to be encoded. The larger the sliding window is, the encoder searches for finding longer matches, but it adds to the overhead of higher number of comparisons required for finding the longest prefix. Determining the longest prefix is the major

Algorithm 1 LZ77 compression algorithm pseudo-code

1 Fill Look-ahead buffer from input ;
2 **while** *Look-ahead buffer is not empty* **do**
3   Find longest prefix p of view starting in Look-ahead buffer;
4   *offset* := position of p in window;
5   *length* := number of characters in p;
6 X := first char after p in view;
7   Output (*offset*, *length*, X);
8   Add *length*+1 chars to the Look-ahead buffer;
9 **end**

Table 13.1 LZ77 compression of string *aacaacbcabaaac* with dictionary buffer size 8 and Look-ahead buffer size 6

Sl#	Dictionary								Look-ahead						Output
	8	7	6	5	4	3	2	1	0	1	2	3	4	5	
1									a	a	c	a	a	c	(0,0,a)
2								a	a	c	a	a	c	b	(1,1,c)
3						a	a	c	a	a	c	b	c	a	(3,3,b)
4		a	a	c	a	a	c	b	c	a	b	a	a	a	(5,2,b)
5	c	a	a	c	b	c	a	b	a	a	a	c			(7,2,a)
6	c	b	c	a	b	a	a	a	c						(8,1,$)

computation in the LZ77 algorithm. To determine individual match in characters in Look-ahead buffer and Dictionary buffer, *identity comparator* is needed. For determining the correct values of *offset*, *length* and *X*, *priority multiplexers* would be needed, with the priority based on the length of the match.

Example 1 To facilitate understanding of the algorithm, we present an example for encoding the string *aacaacbcabaaac*. Table 13.1 demonstrates the encoding of the string using a dictionary buffer size of 8 and a Look-ahead buffer size of 6. The encoded output is shown in the last column of the table, is of the form (*distance*, *length*, *next character X*). It should be noted that the distance is relative to the right edge of the dictionary buffer. The buffers operate on the principle of a sliding window, i.e. the data stream to be compressed is pushed left into the buffer. As noted in the algorithm, the shift is equal to the length of the match found in the dictionary, and a further position.

Initially, the dictionary buffer is empty and there are no matches, hence (0,0,a) is the output. The next character *a* in the Look-ahead buffer is a match with one character at distance 1 in the dictionary buffer, and hence the output is (1,1,*c*). At distance 3, three characters match the left most three characters of the Look-ahead buffer, and thus the output is (3,3,*b*). In the following step, two characters at distance 5 match the two left-most character of Look-ahead buffer, so the output is (5,2,*b*). In this step, two characters match and the output is (7,2,*a*). In the last step, the output is (8,1,$) as the last character c of the uncompressed string matches and to signify the end of string, $ symbol is used.

13.2 ReVAMP Architecture for In-Memory Computing

In this section, we explain the general purpose in-memory computing platform, ReVAMP, introduced in [4]. We also demonstrate how comparator and priority multiplexer can be realized using instructions of ReVAMP.

The ReVAMP architecture, presented in Fig. 13.1, utilizes two ReRAM crossbar memories with light weight peripheral circuitry. A ReRAM crossbar memory

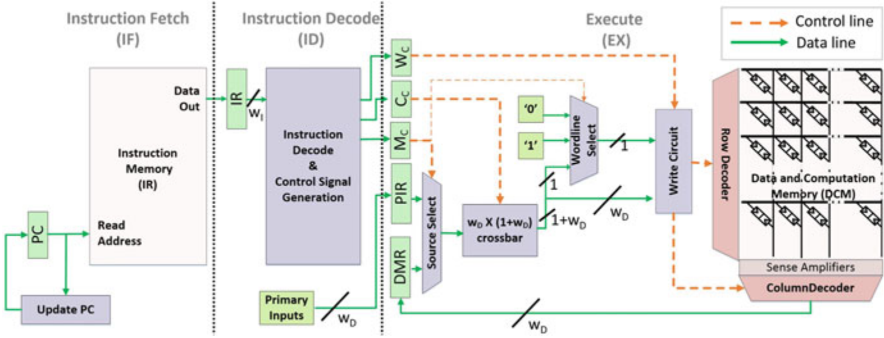


Fig. 13.1 ReVAMP architecture [4]

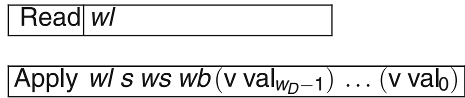


Fig. 13.2 ReVAMP instruction set

consists of multiple 1S1R ReRAM devices [5], arranged in the form of a crossbar [6]. Like conventional RAM arrays, ReRAM memories are accessed as w_D-bit wide words.

One of the memory arrays is used as instruction memory (IM). The IM is used as regular memory, with the program counter (PC) being used to access the next instruction. The other array is used as data storage and computation memory (DCM). In the DCM, in-memory computation using ReRAM devices takes place.

Each ReRAM device has two input terminals, namely the wordline *wl* and bitline *bl*. The internal resistive state Z of the ReRAM acts as a third input and stored bit. The next state of the device Z_n can be expressed as Boolean majority function with three inputs, with the bitline input inverted, as shown in the following equation.

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

This forms the fundamental logic operation that can be realized using ReRAM devices. Using the intrinsic function Z_n, inversion operation can be realized. Since majority and inversion operation form a functionally complete set, any Boolean function can be realized using the Z_n.

The ReVAMP architecture has a three-stage pipeline with Instruction Fetch (IF), Instruction Decode (ID), and Execute (EX) stages. The ReVAMP architecture can be programmed using two instructions—*Read* and *Apply*, with the format shown in Fig. 13.2.

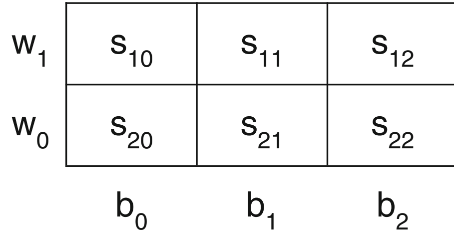


Fig. 13.3 A ReRAM crossbar array with two wordlines and three bitlines

Read instruction reads a specified word, *wl* from the DCM and stores it in the Data Memory Register (DMR). The read out word, available in the DMR, can be used as input by the next instruction.

The *Apply* instruction is used for computation in the DCM. The address *wl* specifies the word in the DCM that will be computed upon. A bit flag *s* chooses whether the inputs will be from primary input register (PIR) or DMR. Two-bit flag *ws* is used to select the wordline input—11 selects '1', 10 selects '0', 00 selects *wb* bit within the chosen data source for use as wordline input while 01 is an invalid value for *ws*. Pairs (v, val) are used to specify individual 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.

We introduce the notations used for the implementation of the logic operations on ReRAM crossbars. Figure 13.3 shows a ReRAM array with two wordlines and three bitlines. Input w_1 and w_0 are the wordline inputs while b_2, b_1, and b_0 are the bitline inputs. The variable s_{ij} represents the internal states of device at wordline i and bitline j. Input '1', '0', and 0 represent V/2, −V/2, and GND, respectively. From the perspective of logic, inputs '1' and '0' represent Boolean logic 1 and 0, respectively, while input 0 represents no-operation. In a readout phase, the presence of a 5 μA current is considered as Boolean logic 1 while absence of current is interpreted as Boolean logic 0.

Figure 13.4 shows how a 32-bit word can be loaded into the DCM using the ReVAMP instructions. The word $w_{31:0}$, available in the PIR, is loaded into the DCM by using an Apply instruction. This loads the words in inverted form in word i, as shown in Fig. 13.4a. In the next cycle, the inverted word is read out from word i using Read instruction, which stores it in the DMR. Another Apply instruction is used to write the word in non-inverted form to word j, by selecting writing the contents of the DMR as shown in Fig. 13.4c. The reader should understand the equivalence of the ReVAMP instructions and the representation of the crossbar operations, since these notations will be used interchangeably.

Figure 13.5 presents realization of basic Boolean functions. Figure 13.5a shows how an array can be reset to 0, irrespective of its contents. This is true because $M_3(x, 0, \neg 1) = 0$. To compute AND of two inputs, the first input is loaded into the array and then the negated second input is applied to the bitlines with '0' as wordline input, because $M_3(0, a, \neg(\neg b)) = a.b$. Similarly, OR of two inputs can be computed, by changing the wordline input to '1', since $M_3(0, a, \neg(\neg b)) = a + b$.

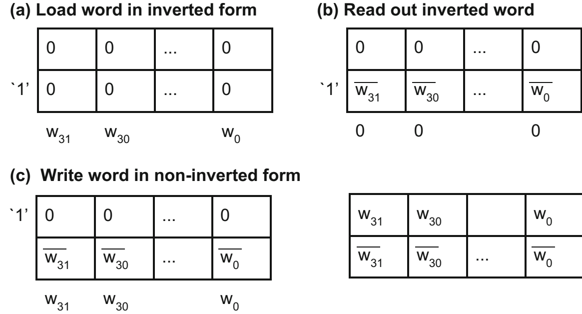


Fig. 13.4 Loading a word into DCM **(a)** Load word in inverted form, **(b)** Read out inverted word and **(c)** Write word in non-inverted form

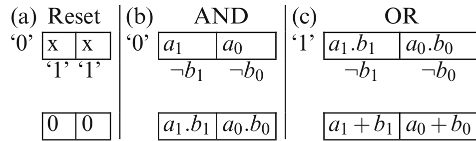


Fig. 13.5 Realization of basic Boolean functions

This concludes the description of the ReVAMP architecture and basics of logic function realization using it. In the following subsections, we present the realization for identity comparator and priority multiplexer, which are required for LZ77 compression.

13.2.1 Comparator Design

An identity comparator compares the value of the inputs and generates a HIGH output only when both the inputs are identical, otherwise the output is LOW. An identity comparator for 4-bit values can be represented by the following equation:

$$c_4 = (a_0 \odot b_0).(a_1 \odot b_1).(a_2 \odot b_2).(a_3 \odot b_3) \tag{13.1}$$

$$a \odot b = \overline{a}.\overline{b} + \overline{\overline{a} + \overline{b}} \tag{13.2}$$

where a_i and b_i represent the ith bits of input data signals a and b, respectively. $\overline{a}$ represents the negated value of Boolean variable a. Operators ., $+$, and $\odot$ represent Boolean AND, OR, and XNOR operations, respectively. The XNOR operation can be expressed in terms of AND and OR operations as shown in (13.2).

Without loss of generality, we demonstrate the implementation of identity comparator using ReRAM arrays, for 4-bit inputs, as shown in Fig. 13.6. The grayed wordline represents the read out word.

Step 1: Word a is read out and an inverted copy of the word is created.
Step 2: Similar to step 1, another copy of $\overline{a}$ is created.
Step 3: In this step, $\overline{a_i}.\overline{b_i}$ is computed in the by reading out and applying b via the bitlines and '0' as wordline input, since $M_3(\overline{a}, 0, \overline{b_i}) = \overline{a_i}.\overline{b_i}$.

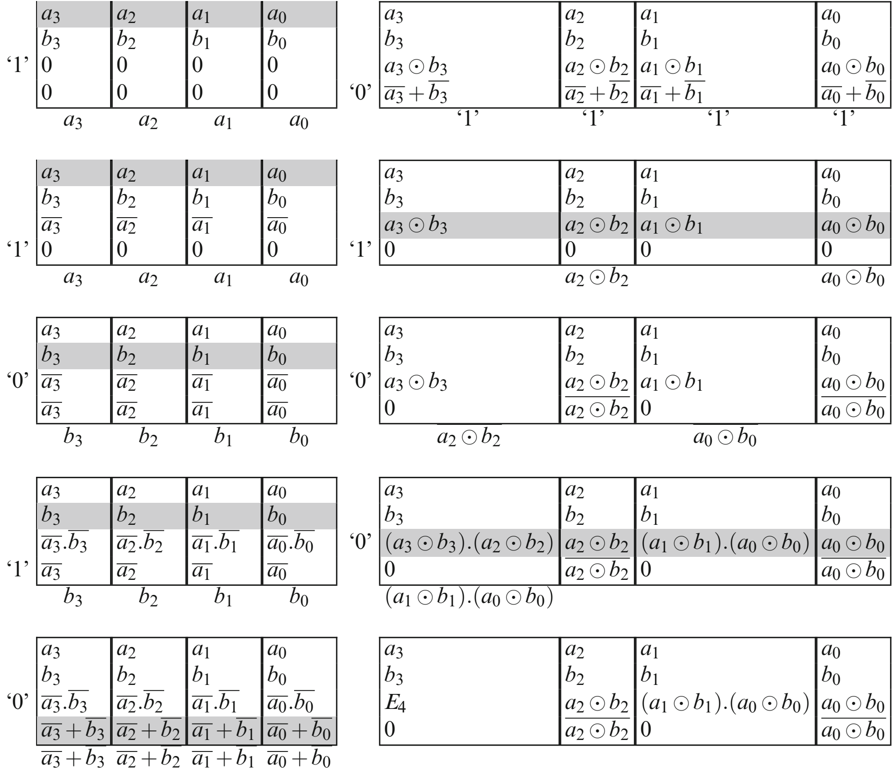


Fig. 13.6 Four bit identity comparator realization

Step 4: $\overline{a_i} + \overline{b_i}, 0 \leq i \leq 3$ is computed in the by reading out and applying b via the bitlines and '1' as wordline input, since $M_3(\overline{a_i}, 1, \overline{b_i}) = \overline{a_i} + \overline{b_i}$.
Step 5: The intermediate term $\overline{a_i} + \overline{b_i}$ is read out and ORed with corresponding $\overline{a_i}.\overline{b_i}$ to compute $a_i \odot b_i$. This completes completion of XNOR computation. It should be noted that as long as the number of bits in the word is less than w_D, the bitwise XNOR of the words a and b can be computed using fixed number of steps.
Step 6: The word holding the intermediate results $\overline{a_i} + \overline{b_i}$ is reset to 0.
Step 7: Now, the computed XNOR terms are combined together using an AND-reduction tree, as shown in Fig. 13.7. XNOR terms $a_2 \odot b_2$ and $a_0 \odot b_0$ are read out and stored in inverted forms.
Step 8: The inverted XNOR terms are read out and ANDed with the appropriate $a_i \odot b_i$ terms.
Step 9–10: The last two steps are similar to **Step 7–8** and compute the final identity comparator result E_4.

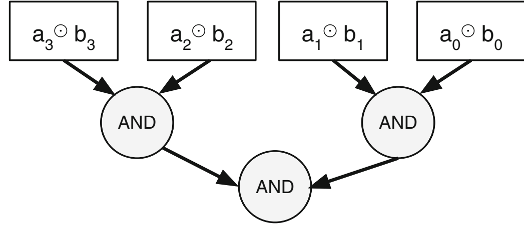


Fig. 13.7 AND-reduction tree for identity comparator

13.2.1.1 Analysis

Each step except Step 6 involves a read operation followed by a computation—which implies a Read instruction followed by an Apply instruction and therefore requires two cycles. For computation of the bit-wise XOR, ten cycles are required. One cycle is required to reset the word holding the intermediate term. For the AND-reduction tree, there are $\lceil \log_2 n \rceil$ levels, where n is the number of bits in the inputs. Each level requires four cycles, therefore the reduction tree computation requires $4\lceil \log_2 n \rceil$ cycles. Thus, an n-bit ($n \leq w_D$) identity comparator would require $11 + 4\lceil \log_2 n \rceil$ cycles to be realized on ReVAMP architecture.

13.2.2 Priority Multiplexer Design

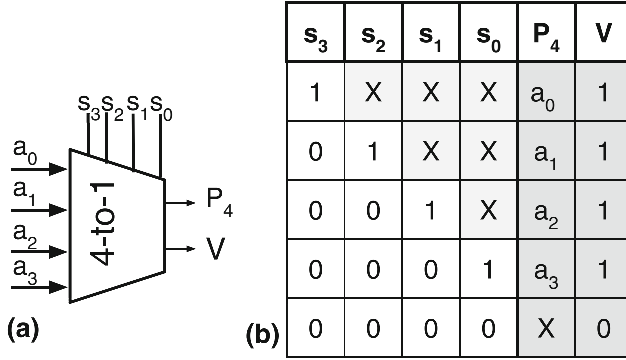
A priority multiplexer selects from one of the n data signals, based on the n control signals, which have a predefined priority. Basically, the priority multiplexer selects input signal a_k, if control signal s_k is '1' and none of the other control signals with priority more than s_k are '1'. If none of the select signals are '1', then the output is invalid. A 4-bit priority multiplexer is represented by the truth table in Fig. 13.8b and the following equations.

$$p_4 = s_3.a_3 + \overline{s_3}.s_2.a_2 + \overline{s_3}.\overline{s_2}.s_1.a_1 + \overline{s_3}.\overline{s_2}.\overline{s_1}.s_0.a_0 \tag{13.3}$$

$$V = s_0 + s_1 + s_2 + s_3 \tag{13.4}$$

where s_j and a_k represent the control and data signals, respectively. Priority of control signal s_j is greater than s_k, if $j < k$. The output signal valid V is '1' when the output is valid, otherwise it is low.

We demonstrate how priority multiplexers for 4, 3, 2, and 1 input can be realized simultaneously using ReRAMs, with the overall delay being determined by the delay of the 4-input priority multiplexer computation. Let a, b, c, and d by the data inputs and s^1, s^2, s^3, and s^4 be the select signals to the four priority multiplexers, respectively. The initial steps of the computation is shown in Fig. 13.9. In Step 1, the select signals are read out and an inverted copy is written back. In Step 2, the s_i^t is ANDed with the appropriate data signal. From Steps 3–5, the $\overline{s_i^t}$ terms are ANDed



s_3	s_2	s_1	s_0	P_4	V
1	X	X	X	a_0	1
0	1	X	X	a_1	1
0	0	1	X	a_2	1
0	0	0	1	a_3	1
0	0	0	0	X	0

Fig. 13.8 4-Input priority multiplexer. (**a**) Block diagram. (**b**) Truth table

with the appropriate intermediate AND terms. In Step 7, the wordline storing the inverted select signals is reset. From Step 8 onwards, the final result of the priority multiplexer P_n is computed by using an OR-reduction tree (similar to Fig. 13.7). To compute the valid output V, another OR-reduction tree for the select signals would be required.

In general, two cycles are required to compute and write the inverted select signals. n steps are required to compute all the AND terms with each step involving a Read and Apply instruction. The reset operation requires one cycle. Finally, the OR-reduction tree requires $4\lceil \log_2 n \rceil$ cycles, similar to the AND-reduction tree. For the computation of the valid output signal, additional $4\lceil log_2 n \rceil$ cycles would be required. Therefore, an n-input priority multiplexer requires $3+2n+8\lceil log_2 n \rceil$ cycles to complete execution. For the specific case of 4-input priority multiplexer, 27 cycles are required.

In the next section, we demonstrate how LZ77 compression can be realized using logic operations on ReRAM. We will be required to use the comparator and priority multiplexer designs introduced above for LZ77 compression.

13.3 LZ77 Compression Using ReVAMP

In this section, we present the implementation details of LZ77 on the ReVAMP architecture. We assume word length w_D of the ReVAMP architecture to be 32. For the LZ77 compression, we assume each character to be 8-bits, since ASCII text representation uses 8-bits. In addition, we consider the Dictionary buffer to hold 4-characters and Look-Ahead buffer to hold 5-character. Let the contents of the Dictionary buffer and the Look-ahead buffer be as shown in Fig. 13.10.

The key computation in LZ77 is finding the longest prefix p of view starting in Look-ahead buffer that is present in the dictionary buffer. Initially, the individual characters are compared in the Look-ahead buffer and Dictionary buffer—s_i^j is the

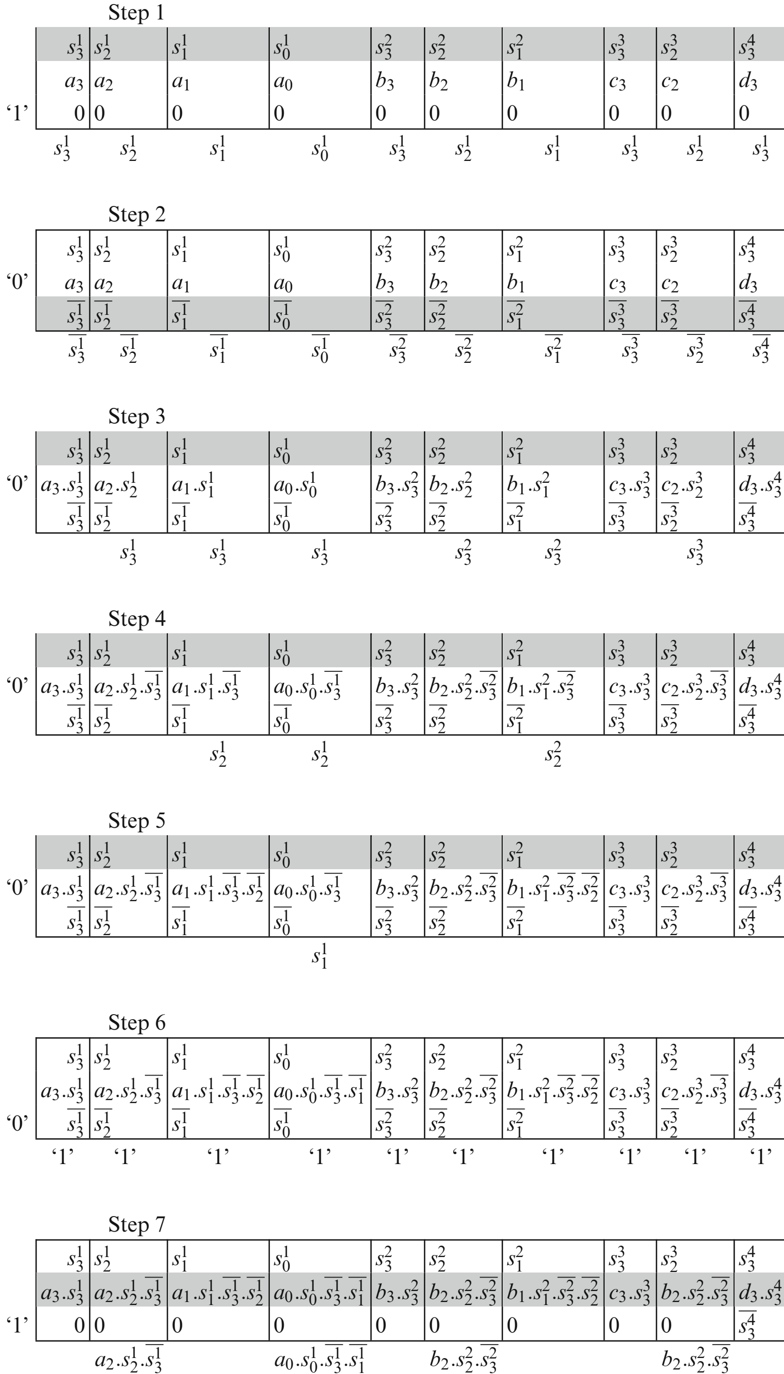


Fig. 13.9 Realization of priority multiplexer

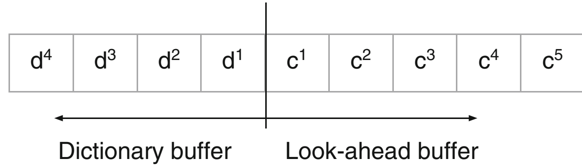


Fig. 13.10 Dictionary and Look-ahead buffer

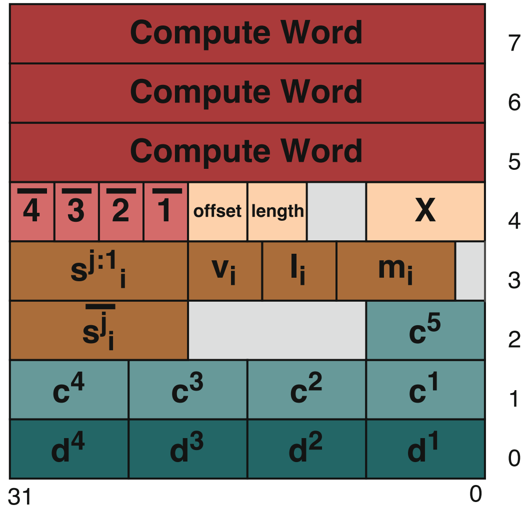


Fig. 13.11 DCM layout

result of comparison of ith character in Look-ahead buffer and jth character in the Dictionary buffer. This is followed by determining what are the series of characters that match in the dictionary buffer—$s_i^{j:1}$ is 1, if characters from 1 to j positions in the Look-ahead buffer matches the characters from location i in Dictionary buffer. Using these results, the *offset* are determined. This is followed by determining the length of the priority multiplexers—l_i indicates that a prefix of length i is present. Using l_i, the outputs *length* and next character X is determined. Finally, the Dictionary buffer is shifted appropriately, depending on the length of the longest prefix.

The computation in the ReVAMP architecture takes place in the Data and Computation Memory (DCM). The layout of the DCM is shown in Fig. 13.11. Word 0 holds the contents of the Dictionary buffer while word 1 and the first 8-bits of word 2 act as Look-ahead buffer. In addition, word 2 holds results of character comparisons. Word 3 holds the select signals for the priority multiplexers, priority multiplexer outputs, and the valid bits. Word 3 holds constants 4, 3, 2, and 1 in inverted forms and will also store the *offset*, *length*, and next character X that is output by the algorithm. Finally, words 5–7 are used for computation.

To do so, we undertake the following sequence of operations. Compare the characters to determine if a prefix of length 1 is present. $s_i^1 = (d^i == c^1), 1 \geq i \geq 4$.

Similarly, comparisons are undertaken to determine if prefix of length 2, 3, and 4 are present. Each set of comparisons to determine a prefix of certain length t can be performed in parallel on the words 5–6 in the DCM, using the comparator realization

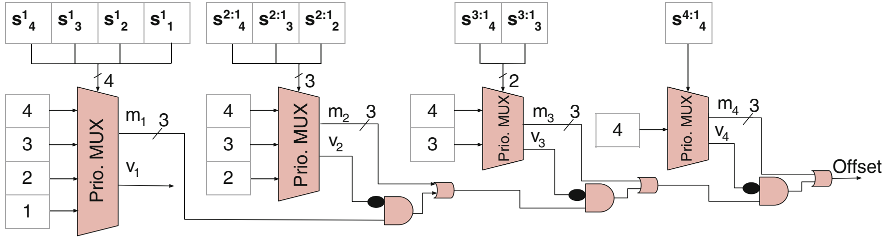


Fig. 13.12 Offset computation using Priority multiplexers

present in subsection 13.2.1. Once the comparison is complete, the s_i^t terms are read out and written in inverted form to the word 2. Then the words 5–6 are reset and the next set of comparisons are performed.

This is followed by computation of all the terms $\overline{s_i^{j:1}}$, $4 \geq \{i, j\} \geq 1$ in parallel. The $\overline{s_4^{4:1}}$ requires the most number of cycles to be computed, equal to 8 cycles. Finally, two cycles are need to read out the $\overline{s_i^{j:1}}$ terms and write it to word 3 in non-inverted form.

$$s_i^{j:1} = s_i^j.s_i^{j-1} \ldots s_i^2.s_i^1 \tag{13.5}$$

Using the $s_i^{j:1}$ terms, the offset can be determined by using a series of priority multiplexers as shown in Fig. 13.12 along with some additional computation for the AND and ORs. The priority multiplexers are realized in parallel, by the steps described in Sect. 13.2.2. Once the priority multiplexer computations are over, the AND and OR terms are computed sequentially to compute "*offset*." The computed offset is written to word 4. The computation words 5–7 are reset once again.

For computation of "*length*," a single priority multiplexer is used with select signals l_i, as defined below.

$$l_4 = s_4^{4:1} \tag{13.6}$$

$$l_3 = s_4^{3:1} + s_3^{3:1} \tag{13.7}$$

$$l_2 = s_4^{2:1} + s_3^{2:1} + s_2^{2:1} \tag{13.8}$$

$$l_1 = s_4^{1:1} + s_3^{1:1} + s_2^{1:1} + s_1^{1:1} \tag{13.9}$$

The select signals are computed in parallel with $\overline{l_i}$. This requires six cycles. Additional cycles are required to read out $\overline{l_i}$ and store l_i. This is followed by computation for the priority multiplexer to determine "length."

In order to compute the output character X, another priority multiplexer computation is used, with l_i as select signals as shown in Fig. 13.13, similar to that for computation of "length." If there is no-match in the Dictionary buffer, then the valid bit v is 0 and c^1 is the next character X. Using this, we can determine the correct next

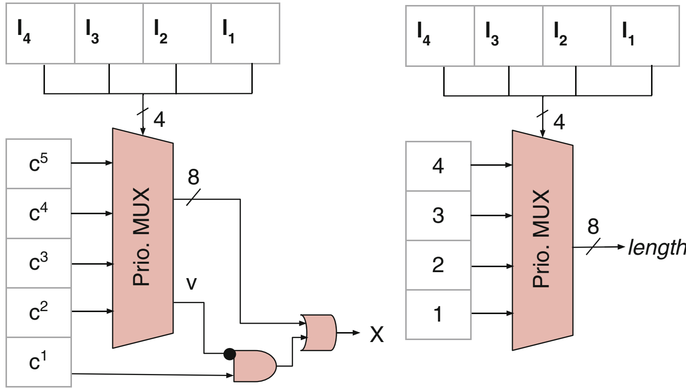


Fig. 13.13 Computation of output "length" and character X

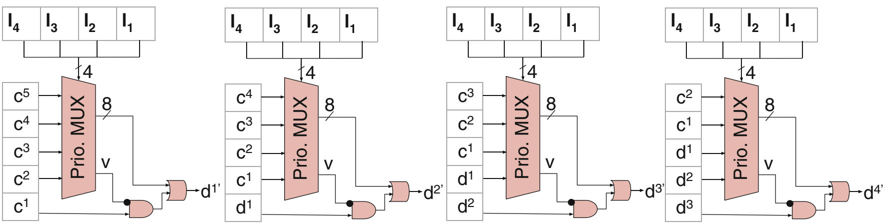


Fig. 13.14 Dictionary buffer update

character. This completes computation of the output (*offset*, *length*, *X*) for this round of the LZ77 algorithm. The computed output is readout using a Read instruction and available in DMR to be read out.

The dictionary buffer and the Look-ahead buffer need to be updated before the next iteration of the LZ77 can begin. For updating each character in the dictionary buffer, a priority multiplexer operation is used followed by an OR operation, with an inverted input. The priority multiplexers and the corresponding inputs and select signals are shown in Fig. 13.14. We should note that the valid bit v is computed once, since the select signals to the priority multiplexers are identical. Once the new character at a given position has been computed, the old character is reset in word 0 and the new character is written.

All the characters in the Dictionary buffer locations are reset to 0 and based on the length output, the contents of the Look-ahead buffer are loaded via the PIR and Apply instructions for the next iteration of LZ77 algorithm.

13.4 Performance Estimation

About 375 cycles are required for each iteration of the LZ77 algorithm. Updating the dictionary buffer requires 148 cycles while the initial comparisons along with computation of the $s_i^{j:1}$ terms require 92 cycles. To estimate the performance, we assume mature ReRAM technology with 1 ns access time, based on [7]. For the uncompressed text *aacaacbcabaaac* given in Example 1, seven iterations would be needed to compress it using the proposed implementation of LZ77 and 2.625 μs would be required to complete all the iterations.

The area of the proposed implementation can be measured in terms of the number of words required in DCM and IM. The proposed implementation requires seven words only, with each word 32-bit wide in DCM. Assuming the DCM to be addressed by 3-bits, each Apply instruction would require 201 bits and we assume that the Read instruction is padded with 0 s to make it of the same length as the Apply instruction. The proposed implementation requires ≈5.46 KB of memory for storing the instructions, considering 32-bit aligned memory access.

13.5 Related Works

The majority of the work related to in-memory computing related to ReRAMs can be broadly classified into three categories—dedicated circuit proposals, general purpose computing architectures using ReRAMs, and design automation tools for the architectures.

In [8], ReRAM cells were shown to be conditionally switchable sequential logic devices, thereby allowing logic-in-memory operations directly. Feasibility and performance of multiple logic-in-memory adder designs have been presented in the recent literature by means of memristive simulations [9–11]. Level-1 and Level-2 Binary Basic Linear Algebra Sub-routines (BiBLAS) were realized using ReRAM crossbas arrays [12, 13]. Neuromorphic computing has also been realized using ReRAMs [14–16]. Authors in [17] utilized the crossbar array as a Content-Addressable Memory (CAM) structure similar to those earlier proposed in [18] for realizing integer matrix multiplication.

A general approach to designing in-memory architecture for data-intensive applications was presented in [19]. Gaillardon et al. [20] introduced a light weight controller to enable general purpose computing using ReRAM arrays, using a bit-serial operation mode with a single instruction. The ReVAMP architecture [4] has two instructions and uses separate instruction and computation memories and allows bit-level parallel operations, thereby offering considerable speedup over PLiM computer [20].

Considerable amount of research has been undertaken for developing automation tools related to logic synthesis and technology mapping using memristors. In [9], the authors presented a basic methodology for computing Boolean functions using

memristive devices. In [21], it has been shown that with two working memristors which realize material implication, any Boolean expression can be computed. In [22] and [23], logic synthesis solution for memristors that realize material implication has been proposed. In [24], a compiler for flow for generating RM_3 instructions, for the ReRAM based PLiM computer [20] for realization of Boolean functions has been presented. In [25], heuristics for logic synthesis of MIG for two variants of ReRAM has been proposed —one realizing material implication and the other realizing majority function. In [25], the authors used a naïve technology mapping with delay of $3k + c$ cycles, for an MIG with k levels and c number of levels with ingoing complemented edges. In [26], the authors demonstrated logic realization using memristive crossbar arrays using multi-bit adders and multipliers as case studies. In [27], the authors proposed a delay optimal technology mapping solution for memristive devices. Further, area-constrained technology mapping for ReRAM devices was presented in [28].

13.6 Summary

This chapter introduced the ReVAMP in-memory computing architecture that utilizes stateful logic operations on ReRAM crossbar arrays. Realizations of comparator and priority multiplexer was presented using the ReVAMP instructions. We presented implementation of LZ77 compression algorithm using the ReVAMP instructions and analyzed the performance in terms of number of cycles and area in terms of number of devices. Finally, we presented the landscape of research in the field of in-memory computing using memristors.

References

1. R. Waser, R. Dittmann, G. Staikov, K. Szot, Redox-based resistive switching memories–nanoionic mechanisms, prospects, and challenges. Adv. Mater. **21**, 2632–2663 (2009)
2. E. Linn, R. Rosezin, C. Kügeler, R. Waser, Complementary resistive switches for passive nanocrossbar memories. Nat. Mater. **9**(5), 403–406 (2010)
3. J. Ziv, A. Lempel, A universal algorithm for sequential data compression. IEEE Trans. Inf. Theory **23**(3), 337–343 (1977)
4. D. Bhattacharjee, R. Devadoss, A. Chattopadhyay, ReVAMP : ReRAM based VLIW Architecture for in-Memory comPuting, in *Design, Automation & Test in Europe Conference & Exhibition, DATE 2017* (2017)
5. A. Siemon, S. Menzel, A. Marchewka, Y. Nishi, R. Waser, 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
6. E. Linn, R. Rosezin, S. Tappertzhofen, U. Böttger, R. Waser, Beyond von neumann-logic operations in passive crossbar arrays alongside memory operations. Nanotechnology **23**(30), 305205 (2012)
7. Emerging Research Devices (ERD) report, International Technology Roadmap for Semiconductors (ITRS) (2013)

8. J. Borghetti, G.S. Snider, P.J. Kuekes, J. Joshua Yang, D.R. Stewart, R. Stanley Williams, ‘Memristive’ switches enable ‘stateful’ logic operations via material implication. Nature **464**(7290), 873–876 (2010)
9. E. Lehtonen, M. Laiho, Stateful implication logic with memristors, in *Proceedings of the 2009 IEEE/ACM International Symposium on Nanoscale Architectures* (2009), pp. 33–36
10. S. Kvatinsky, G. Satat, N. Wald, E.G. Friedman, A. Kolodny, U.C. Weiser, Memristor-based material implication (imply) logic: design principles and methodologies. IEEE TVLSI **22**(10), 2054–2066 (2014)
11. A. Siemon, S. Menzel, R. Waser, E. Linn, A complementary resistive switch-based crossbar array adder. IEEE JETCAS **5**(1), 64–74 (2015)
12. D. Bhattacharjee, F. Merchant, 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)*, September 2016, pp. 1–6
13. D. Bhattacharjee, A. Chattopadhyay, Efficient binary basic linear algebra operations on reram crossbar arrays, in *2017 30th International Conference on VLSI Design*, January 2017
14. 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, R.S. Williams, Hybrid cmos/memristor circuits, in *ISCAS*, pp. 1967–1970 (2010)
15. K.-H. Kim, S. Gaba, D. Wheeler, J. M. Cruz-Albrecht, T. Hussain, N. Srinivasa, W. Lu, A functional hybrid memristor crossbar-array/cmos system for data storage and neuromorphic applications. Nano Letters **12**(1), 389–395 (2011)
16. M.P. Sah, H. Kim, L.O. Chua, Brains are made of memristors. IEEE Circuits Syst. Mag. **14**(1), 12–36 (2014)
17. L. Ni, Y. Wang, H. Yu, W. Yang, C. Weng, J. Zhao, An energy-efficient matrix multiplication accelerator by distributed in-memory computing on binary RRAM crossbar, in *ASP-DAC*, January 2016, pp. 280–285
18. F. Alibart, T. Sherwood, D.B. Strukov, Hybrid cmos/nanodevice circuits for high throughput pattern matching applications, in *Adaptive Hardware and Systems (AHS), 2011 NASA/ESA Conference on* (2011), pp. 279–286
19. S. Hamdioui, L. Xie, H. Anh Du Nguyen, M. Taouil, K. Bertels, H. Corporaal, H. Jiao, F. Catthoor, D. Wouters, L. Eike, J. van Lunteren, Memristor based computation-in-memory architecture for data-intensive applications, in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition, DATE 2015, Grenoble, March 9–13, 2015* (2015), pp. 1718–1725
20. P.-E. Gaillardon, L. Amaru, A. Siemon, E. Linn, R. Waser, A. Chattopadhyay, G. De Micheli, The Programmable Logic-in-Memory (PLiM) computer, in *DATE* (2016), pp. 427–432
21. J.H. Poikonen, E. Lehtonen, M. Laiho, On synthesis of Boolean expressions for memristive devices using sequential implication logic. IEEE TCAD **31**(7), 1129–1134 (2012)
22. A. Raghuvanshi, M. Perkowski, Logic synthesis and a generalized notation for memristor-realized material implication gates, in *ICCAD* (2014), pp. 470–477
23. A. Chattopadhyay, Z. Endre Rakosi, Combinational logic synthesis for material implication, in *IEEE/IFIP 19th International Conference on VLSI and System-on-Chip, VLSI-SoC 2011, Kowloon, Hong Kong* (2011), pp. 200–203
24. M. Soeken, S. Shririnzadeh, P.-E. Gaillardon, L. Amarú, R. Drechsler, G. De Micheli, An mig-based compiler for programmable logic-in-memory architectures, in *DAC* (2016)
25. S. Shirinzadeh, M. Soeken, P.-E. Gaillardon, R. Drechsler, Fast logic synthesis for RRAM-based in-memory computing using majority-inverter graphs, in *DATE* (2016)
26. L. Xie, H. Anh Du Nguyen, M. Taouil, K. Bertels, S. Hamdioui, Fast boolean logic mapped on memristor crossbar, in *33rd IEEE International Conference on Computer Design, ICCD 2015*, New York City, NY, October 18–21 (2015), pp. 335–342

27. D. Bhattacharjee, A. Chattopadhyay, Delay-optimal technology mapping for in-memory computing using reram devices, in *Proceedings of the 35th International Conference on Computer-Aided Design, ICCAD 2016*, Austin, TX, November 7–10 (2016), p. 119
28. D. Bhattacharjee, A. Easwaran, A. Chattopadhyay, Area-constrained technology mapping for in-memory computing using reram devices, in *Asia and South Pacific Design Automation Conference, ASP-DAC* (2017), pp. 1–6