FPGA Implementation of Associative Processors

Hongzheng Tian, Mohammed E. Fouda, Minjun Seo and F. J. Kurdahi

***Abstract*—In order to deal with increasingly complex computing problems, an In-memory-based computation system was proposed to replace the traditional Von-Neumann architectures. In-memory computing can save the time and energy of data movement between the memory and processor to avoid the memory-wall bottleneck of traditional Von-Neumann architecture. The associative processor (AP) is such an architecture that is proposed to implement in-memory computing. Content addressable memory (CAM), as a critical part of in-memory computing, plays an important role in an AP. In this paper, we proposed a novel FPGA implementation of the AP, including the CAM and its peripheral circuits, such as the controller, data cache, instruction cache, and program counter. The design details of the whole AP architecture are described by Verilog HDL. To the best of our knowledge, this is the first work that implements an associative processor on a real-world FPGA platform.**

***Index Terms*—In-Memory Computing, Associative Processors, FPGA.**

I. Introduction

To solve the memory bottleneck problem, the researchers designed the in-memory computation structure [1], which is considered the most efficient computing paradigm. All the computations are performed in the memory without moving the data between the processor and memory. In-memory computing not only reduces the amount of data access between the processor and memory but also reduces the computational complexity of the problem [2].

The associative processor (AP) is considered to be an excellent platform for implementing in-memory computation. The key idea of implementing an associative processor is to place a small arithmetic logic unit in each storage unit. Since APs can easily implement single instruction, multiple data (SIMD) [3], it has a strong performance in parallel computing. It is widely used in various computing forms, such as convolution [4], matrix multiplication [5], [6] and fast Fourier transform (FFT) [7]. This makes it popular in artificial intelligence fields such as machine learning and deep learning [8], [9].

The associative processors use content addressable memories (CAMs) to compute the data directly inside the memory. Currently, most CAM implementations are based on traditional SRAM designs [10]. However, this solution has a lot of static and dynamic power consumption, and it is area inefficient. Another possible implementation is the Field Programmable Logic Gate Array (FPGA). Due to the outstanding performance of FPGA in parallel processing and its high flexibility, it provides a possibility to implement associative processors based on FPGA. Compared with SRAM-based CAM, FPGA-based CAM is more flexible and can flexibly utilize various resources on FPGA. For instance, [11] contains some implementations of the CAM and TCAM based on the FPGA. According to this survey, there are three main directions to implement the CAM: by using block random-access memory (BRAM), lookup table RAM (LUTRAM), and flip-flops (FFs). Each of them has its advantages and disadvantages. [12] proposed a Ternary CAM (TCAM) implementation based on FPGA, which combines both BRAM and LUTRAM to implement the TCAM architecture.

Currently, most CAM implementations are based on BRAM or LUTRAM since they have more advantages in large CAM sizes. As aforementioned, the existing FPGA-based CAM designs can not perform computation due to the lack of capability of the bitwise read and write operations. In that case, our proposed design is based on the flip-flops to implement the CAM modules, the first CAM design that can compute data at the bit level based on FPGA.

In this paper, we introduced an FPGA-based design of an associative processor and dedicated CAM module designed to ease the AP operation. The contributions of this paper are summarized in the following points:

- To the best of our knowledge, this is the first work that proposed an AP design that contains a flip-flops-based CAM with calculation ability on the FPGA.
- proposed an instruction set architecture (ISA) based on this CAM design, which includes basic logical operations.
- proposed a possible controller and peripheral circuits design.

The rest of the paper is organized as follows: Section II gives the architecture of APs, and section II-C introduces a simple self-designed Instruction Architecture Set for this processor. Then Section III discusses the details of the FPGA-based CAM design for APs. Finally, Section V shows this architecture's experimental results and analysis. Finally, Section VI gives the conclusion and suggestions about future work.

II. AP Architecture

The implementation of the associative processor is mainly based on the CAM, which can automatically compare input data with all the data stored in the CAM. At the same time, it judges whether the input data matches any data stored in the CAM and then outputs the matching information corresponding to the matched data. In general, CAM can search and compare large amounts of data in a single step or time cycle. As a result, a search operation costs *O(1)* time complexity for a CAM-based architecture compared to *O(n)* in traditional architectures.

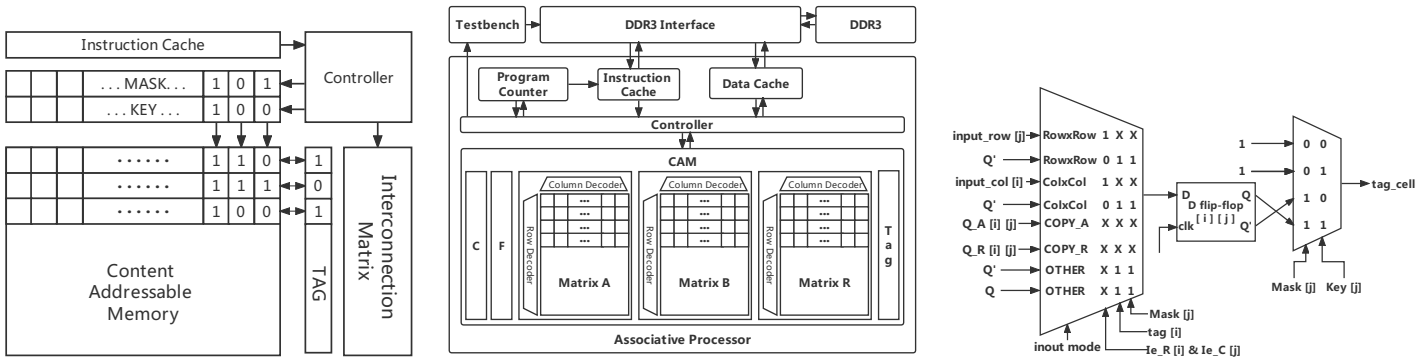
A. General AP Architecture

The architecture of a general AP is shown in Fig. 1a [3]. It contains an *Instruction Cache*, *Controller*, *MASK* registers, *KEY* registers, CAM, *TAG* registers, and an optional interconnection circuit. *Instruction cache* stores the instructions

This work was partially supported by the National Science Foundation under award ECCS-2028782, as well as King Abdullah for Science and Technology under award ORA-2021-CRG10-4704.

Hongzheng Tian, Minjun Seo, and F. J. Kurdahi are with Center for Embedded & Cyber-physical Systems, University of California-Irvine, Irvine, CA, USA 92697-2625.

Mohammed Fouda is with Rain Neuromorphics Inc., San Francisco, CA, USA. Email: foudam@uci.edu



(a) (b) (c)

Fig. 1: (a)General architecture of APs, (b)proposed architecture on FPGA, and (c) proposed design of the B cell.

that are operated on the CAM. The *Controller* then generates the required *MASK* and *KEY* values for the corresponding instruction. The *KEY* register is used to store the value that will be written or compared. *MASK* register indicates which bits are activated during the operation phase. Each bit of *MASK* and *KEY* will be the inputs for all cells of the corresponding column. For each cell, if the *MASK* input is 1, then it means that this cell is activated and will compare the *KEY* input with the value stored in the cell. If they are matched, then the output of this cell will be logic 1. Else if the stored value is unmatched with the given *KEY*, then the output of the cell is 0. In addition, if the *MASK* is 0, this column of cells is not activated, and the outputs of these kinds of cells are 1 by default. The value stored in each cell of the *TAG* area is the result of AND operation of all cells' output of the corresponding row.

For example, in Fig. 1a, we set the *MASK* register as 101, which means the first and third columns are activated to be compared. Furthermore, we set the *KEY* register as 100. Then the cells in the first and third columns will compare the *KEY* values between the stored values. The bits in the *TAG* of the corresponding rows will become logic 1 as shown in Fig. 1a.

To implement logic operations inside the CAM, each CAM cell needs to have the ability to determine whether it needs to be inverted itself according to whether the data matches or not, or, whether the *TAG* is 1 or not. The method to implement this will be illustrated in detail in section IV.

B. Proposed AP Architecture

To make the structure and function of this processor more complete, in addition to the modules mentioned in section II-A, we have added two other modules: the *program counter*, and the *data cache*. The overall design architecture is shown in Fig. 1b.

The proposed AP architecture contains the *Program Counter* (PC), the *Instruction Cache* (IC), the *Data Cache* (DC), the *Controller*, and the CAM. Like a traditional processor, the PC is responsible for counting the addresses of the current program. The PC communicates with the IC and the controller at the same time, because when the program needs to perform operations, such as jumping, the controller needs to send the jumping target address to the PC. After the IC receives the address sent by the PC, it will check whether there is an instruction corresponding to this address in the current cache. If hit, the instruction is sent to the controller for decoding. Otherwise, the IC will make the controller wait, and at the same time send the new address to the *DDR3 Interface*. A group of instructions that corresponds to that address is moved from the DDR3 to the IC. When the IC is full, the needed commands are sent to the waiting controller. The function of the DC is similar to that of the IC. The controller sends the address of the required data to the DC, and the DC judges whether it is a hit or not, and then sends the required data to the controller. The controller is responsible for controlling the operation of the entire processor and data transmission between the cache and CAM. The CAM module contains three matrices A, B, and R, which are used to store and process the data required for one operation. The one-dimensional vectors C and F are used to store the carry and flag bits, and the *Tag* part is used to indicate whether each row in the CAM match or not. This part will be explained in section III in detail.

In addition to this, there are other 3 extra modules, *Testbench*, DDR3, and *DDR3 Interface*. *Testbench* is used to test the entire processor; it reads an instruction file and a data file from the computer and inputs them into the associative processor for processing. After its processing is complete, *Testbench* will output the results of the associative processor to the computer.

To be able to process more data, instead of using a complex testbench function, we store a large amount of data in the DDR3 of the FPGA board. In the data input process of the system. After the DDR3 is initialized, the *DDR3 Interface* module will first send a data request signal to the *Testbench*, and then the *Testbench* will send instructions and data to it in turn and store them in DDR3. After this process is completed, the *DDR3 Interface* module will automatically load the instruction at address 0 from DDR3 into the *Instruction Cache*. When the *Instruction Cache* is loaded fully, the instruction pointed to by the *Program Counter* will be loaded into the *Controller*. Then, the entire system will start executing the program.

C. Instruction Set Architecture

This section introduces a sample of the instruction set architecture (ISA) of this processor. This ISA is a simple self-designed instruction set prototype that focuses on the AP's processing. An instruction contains four parts, 4 bits opcode,

	opcode	example
0 operand opcode	RESET	RESET;
	ADD	ADD;
	SUB	SUB;
	TSC	TSC;
	ABS	ABS;
	STOP	STOP;
1 operand opcode	PRINT	PRINT 0x1057;
2 operands opcode	COPY	COPY M_A M_B;
3 operands opcode	LOADRBR	LOADRBR 0x05 M_A 0x1005;
	LOADCBC	LOADCBC 0x00 M_B 0x1010;
	STORERBR	STORERBR 0x01 M_B 0x1021;
	STORECBC	STORECBC 0x03 M_B 0x1040;

TABLE I: The opcodes of ISA.

8 bits operand-1, 2 bits operand-2, and 16 bits operand-3. Operand-1 indicates the address inside a CAM matrix, and its maximum depends on the depth of CAM. Operand 2 indicates which matrix of CAM will be loaded, which can be chosen from matrix A, matrix B, and matrix R. Operand-3 indicates the address inside the DDR3. If one operand is vacant, fill it with 0.

There are 12 different opcodes, and they can be divided into 0 operand opcodes, 1 operand opcodes, 2 operands opcodes, and 3 operands opcodes. The details and examples are listed in the TABLE I.

Let us emphasize the load and store operations here. In this design, the data in the CAM matrix can be accessed either row by row or column by column. Therefore, storage operations are divided into STORERBR and STORECBC, representing row-by-row storage and column-by-column storage. Similarly, the load operation is divided into LOADRBR and LOADCBC. Whenever a load operation is performed, the controller sends the address (operand 3) that it wants to load to the data cache, and the data cache checks whether the data at this address has been loaded into the cache. If it misses, the cache will load data from DDR3 that starts at the address indicated by operand-3 until the cache is full. Then, if it is LOADRBR, load the data on the row indicated by operand-2 in the cache to the corresponding row in the CAM. If it is LOADCBC, the corresponding column is loaded. Storage is the same. In a word, row by row or column by column is for data exchange between CAM and data cache, not between data cache and DDR3. Data exchange between cache and DDR3 is always row-by-row.

III. Proposed CAM Implementation

As introduced in section II-B, there are three different kinds of matrices (matrix A, B, and R) and registers C, F, and Tag. The matrix A is used to store the addend in addition, subtrahend in subtraction, or the original data of 2's complement and absolute value operations. The matrix B stores the augend, minuend, and in-place addition or subtraction results. And the matrix R can store the results of the out-of-place 2's complement and absolute value operations. Register C will store the carry bit of addition, or borrow a bit of subtraction. Register F will store the flag bit of the out-of-place 2's complement and absolute value operations.

To implement the arithmetic calculation, the controller will set the key and mask values to the CAM matrices according to the corresponding lookup table (LUT) [3]. Then, for the matched rows, we mark the corresponding rows in the tag area and then invert the column whose *MASK* value is 1 according to the LUT. The LUT determines the value of the *KEY* that needs to be set at each phase and whether the corresponding cell of the CAM needs to be inverted.

As shown in Fig. 1b, for each matrix in CAM, there will be two address decoders. One is for decoding the row address, and the other one is for the column address. The output of the row decoder is called Ie_row, and that of the column decoder is called Ie_col. Moreover, there will be an input signal called *inout mode*. If the inout mode is *Row*×*Row*, the input or output method of the CAM will be row-by-row. As a result, all the Ie_col signals will be 1, but only the row indicated by the row address will have Ie_row of 1. Similarly, inout mode is *Col*×*Col*, all the Ie_row signals will be 1, and only 1 Ie_col signal will be 1. And for each cell, only if both Ie_row and Ie_col signals are 1, this cell will be able to get an input or output its value.

In the following part, we will introduce the implementation of cell B as an example. The cell of A, R, C, and F will be similar to it.

A. Cell B Design

The architecture of a B cell is shown in Fig. 1c. Each cell contains a D flip-flop to store 1 bit of data. D[i][j] represents the data stored in the cell in the i^{th} row and j^{th} column storing either 1 or 0. According to Section II-C, there are three input scenarios of a CAM matrix, input row-by-row, column-by-column, and copy from another matrix. As a result, we need a multiplexer to accommodate these scenarios. When this cell is chosen (in other words, Ie_row & Ie_col equals 1) and the inout mode is *Row*×*Row*, then the D will get the input data from the j^{th} bit of the input. On the other hand, if the inout mode is Col×Col, then the D will update to the i^{th} bit of the input. If the instruction is copy from another matrix, then the D[i][j] will be directly equal to the Q[i][j] of the other matrix. When copying from another matrix, signal Ie_R or Ie_C will not be considered. During the evaluation, the multiplexer on the right of Fig. 1c will be considered. If Mask[j] is 1, this column of cells is selected. If Key[j] is 1 and D[i][j] is 0, it means the cell and the key are not matched. So, the output tag will be 0. And if D[i][j] is 1, the output tag will be 1. In other words, if the key is 1, the tag of the cell is equal to Q, if the Key is 0, the tag will be equal to Q'. All the tags from the same row will be ANDed together to generate a tag signal of the row.

The Tag output signal is sent back to the multiplexer of the B cell as shown in Fig. 1c. For instance, in the calculation mode or the non-input mode (i.e. Ie_row & Ie_col equals 0), if Mask[j] is 1 (The column j is selected), Tag[i] is 1 (the row i matches), then D[i][j] is equal to Q', that is, the bit stored in the current cell has been inverted. By inverting the bit value

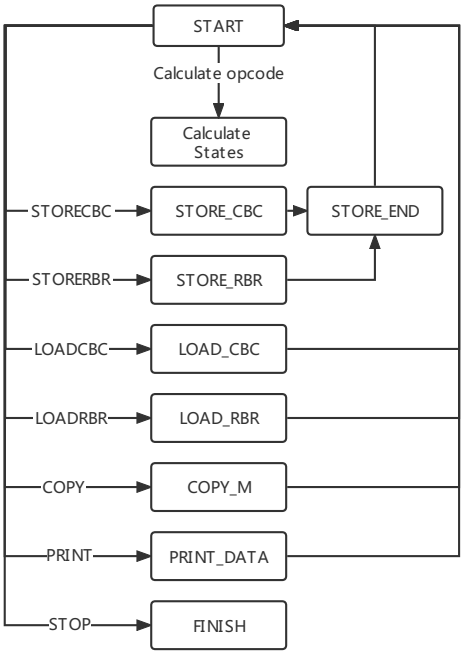


Fig. 2: The state machine of the AP's controller.

of each cell in the CAM, we can change the data stored in the CAM, in other words, we can implement calculations in the memory.

B. Tag Cell Design

As we introduced before, each matrix will output a tag_row to indicate whether this row is matched or not. The tag matrix is used for checking the line of each matrix together. For example, for the ADD operation, we need to calculate the tag for matrices A, B, and C; as a result, we need to connect the tag_row of each row from each matrix to the tag area's corresponding row.

IV. Controller Implementation

The controller is an essential part of the AP because it controls the various parts of the entire processor, decodes the instructions, and transfers data between the cache and the CAM. To make the controller operate correctly, this design uses a state machine to manage the controller. For the convenience of description, here we only introduce the state machine jump process of the controller. The state machine of the entire controller is shown in Fig. 2.

In the beginning, the entire controller will be in the START state, and depending on the opcode, the controller will jump to different states to perform different operations. In the following, we will briefly introduce some details about implementing some operations.

When the opcode is ADD, SUB, ABS, or TSC, the controller enters the calculation state, which is the state of all rectangles in Fig. 2. For APs, as mentioned in the previous section, ADD, SUB, and ABS have four passes, while TSC has three passes. Since the computation of the CAM takes 8 clock cycles to reach the steady state, each PASS state waits for 8 clock cycles before jumping to the next state.

Let us take the ADD operation as an example in Fig. 3. Initially, the controller is in the START state. After receiving

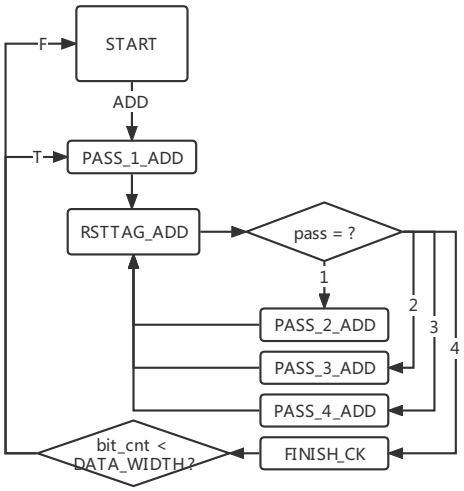


Fig. 3: The state machine of the ADD operation.

the ADD command, the controller jumps to the PASS_1_ADD state and sets the values of the corresponding MASK and KEY registers. After that, the state jumps to the RSTTAG_ADD state. In this state, the controller will issue the rst_tag signal to reset the value of the TAG register in the CAM. The controller will record the current PASS value so that the current PASS information can be saved for return when the interrupt occurs. Then because the current PASS is 1, the next state jumps to PASS_2_ADD, and then back to RSTTAG_ADD. Until the PASS value is 4, the state machine jumps from RSTTAG_ADD to FINISH_CHECK state. In this state, it will check whether the currently calculated bit has reached the MSB of the data; if not, shift the MASK register to the left for 1 bit, and jump to the PASS_1_ADD state to repeat the above process. Otherwise, return to the START state. The operation process of the other three calculations is the same.

Lastly, to load the data from board memory to the CAM, we use LOADCBC or LOADRBR operation, which means loading from the data cache to the CAM column by column or row by row, respectively.

When the controller is at the LOAD_RBR or LOAD_CBC states, it will set the inout_mode as Row×Row or Col×Col, so that the corresponding matrix in CAM can get the input correctly. Then the controller will send the memory address to the data cache, and the data cache will check whether the address misses or hits. If it hits, then the data cache will set the data_cache_rdy signal to indicate that the data cache is ready for the required data. If the address misses, the controller will remain at the load state until the data cache is loading. After getting the data, the controller will transfer the data to the matrix and address of CAM that the instruction indicates.

Similar to the LOAD operation, in the STORE operation, the controller will jump to STORE_RBR or STORE_CBC states and then set the inout_mode to the matrix we want to store and set the corresponding output the enable signal to the CAM matrix. Then the controller will jump to STORE_END state to reset some registers to 0, such as the address for output and the output data registers.

V. Results and Analysis

We use Xilinx Vivado 2018.3 as our design platform and Verilog HDL as the development language. We used Xilinx

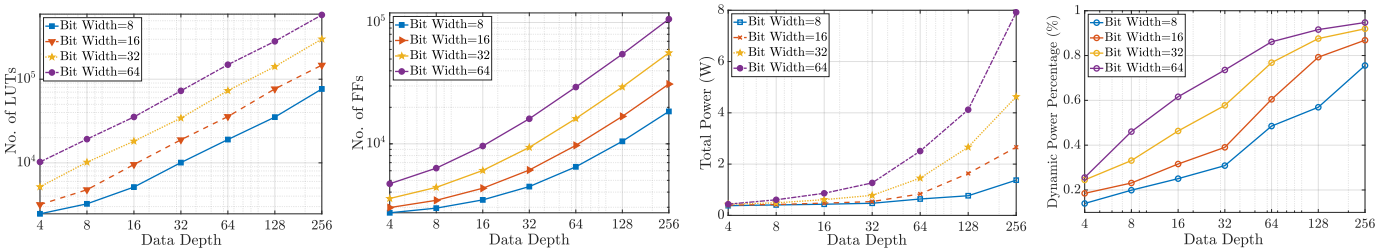


Fig. 4: FPGA utilization and power consumption with increasing data depth.

Virtex-7 FPGA VC709 board, which has 433,200 LUTs and 866,400 FFs running at 100MHz. The resource usage, power, and latency values are obtained from the Vivado synthesis report." The main component of an AP is its CAM matrix. We evaluated the resource utilities and power consumption increase when the bit width and data depth grow. Except for the CAM, the data cache is also sensitive to these two parameters. By changing the two parameters, data depth, and bit width, we can get utilization results about how many LUTs and flip-flops we will use in different data sizes and the power consumption.

The trends of the LUTs and flip-flop utilization as depth grows are shown in Fig. 4(a) and (b) respectively. Different color lines represent different data width in bits, and we can see that their utilization of them will increase linearly as the data depth increases. It also shows that the utilization of LUTs and flip-flops will also approximately double as data width doubles.

Resource constraints undoubtedly limit the scale of AP. As shown in Figure 4a, when the bit width is 64 and the data depth is 256, the consumption of LUTs reaches 587,040. This already exceeds the total LUT resources of the FPGA used. Therefore, in our implementation, the maximum size is 64 bit width x 128 data depth, and 32 bit width x 256 data depth. Compared with the high consumption of LUT, the consumption of FF is much smaller. For a 64 bit width x 256 data depth CAM, the FF consumption is only 106675, as shown in Figure 4b, which is about one-eighth of the total resources of FFs. This is because each cell in the CAM, shown in Figure 1cc, contains one FF only to store data, but it contains two complex LUTs to judge the input and output of a single cell. This leads to high consumption of the LUT. At the same time, the controller logic and the cache heavily rely on the LUT utilization. In short, resource constraints can lead to small-scale CAM arrays which will lead to lower performance due to the high increase in data exchange with external storage. For different use cases, such as ADD, SUB, TSC, and ABS instructions listed in Table I, the impact of resource constraints is the same. The total number of resources limits the product of bit width and data depth. To calculate more data at the same time, it is necessary to reduce the bit width as much as possible, to make the data depth larger.

The growth trend of the total power consumption also increases roughly linearly with the increase of the single dimension of the CAM, as shown in Fig. 4 (c). And according to Fig. 4 (d), the proportion of dynamic power consumption will increase as the size of the CAM increases. This is because the power consumption of peripheral circuits, such as controllers, is not sensitive to the size of the CAM. Also, as the memory gets bigger, it consumes more dynamic power. In addition, we evaluate the energy for some instructions such as ADD and SUB to be 4.68 μJ and latency of 5940 ns for 16 bits and 64 data depth. While other instructions such as TSC which consume 3.55 μJ and 4500 ns latency. When the bit width is doubled, the energy and latency are also doubled.

Although our CAM design is not optimized for memory operation like the previous works which rely more on BRAM and LUT, our design can still be used as traditional CAM where the search is the primary goal. Our design uses 45x and 76x LUTs and FFs, respectively, consumes 24x more power and runs at 1.7x slower frequency as compared to the design proposed in [13] for 512x36 CAM. On the other hand, when compared with ASIC designs based on SRAM or emerging technologies such as RRAMs, ASIC is always superior in terms of performance compared to FPGA-based design due to many factors including the ability to design dedicated hardware and higher operating frequency. To give more insights on our design as compared to ASIC, the 64 of 16-bit ADD operation consumes 4.68uJ on our design that is running at 100 MHz frequency as compared to the RRAM-based ASIC which consumes 1.5nJ running at 1GHz [14]. So, by scaling our design to 1GHz, the energy will improve by 2-5x which could lead to better performance than ASIC. However, this would require further analysis and evaluation. We consider our work as a platform for fast prototyping and evaluating the performance of APs.

VI. Conclusion and Future Work

In this paper, we introduced FPGA-based architecture for the associative processor including the initial instruction set for this AP. Verilog HDL language is used to digitally design CAM, tag, controller, and other peripheral circuits based on its arithmetic operation principle. We showed the linear scalability of the proposed architecture through power, latency, and hardware utilization results. Our CAM designs are much smaller in scale compared to BRAM and LUT implementations of CAM. Importantly, our design demonstrates the feasibility of implementing compute-enabled APs on FPGAs. While our study has shown promising results, further optimization is required. In future work, we plan to optimize the CAM and controller structures to improve the operating frequency and reduce resource usage. Additionally, we aim to expand the initial instruction set to achieve more complex functions, e.g., the program interruption and return, and parallel execution of multiple instructions. Furthermore, after further optimization is completed, we will compare the performance and power

consumption with the existing AP design to observe the improvement brought about by the optimized design. We believe these efforts will help further enhance the applicability and effectiveness of our FPGA-based associative processor architecture.

REFERENCES

[1] N. Verma, H. Jia, H. Valavi, Y. Tang, M. Ozatay, L.-Y. Chen, B. Zhang, and P. Deaville, "In-memory computing: Advances and prospects," IEEE Solid-State Circuits Magazine, vol. 11, no. 3, pp. 43–55, 2019.

[2] A. Sebastian, M. Le Gallo, R. Khaddam-Aljameh, and E. Eleftheriou, "Memory devices and applications for in-memory computing," Nature nanotechnology, vol. 15, no. 7, pp. 529–544, 2020.

[3] M. E. Fouda, H. E. Yantır, A. M. Eltawil, and F. Kurdahi, "In-memory associative processors: Tutorial, potential, and challenges," IEEE Transactions on Circuits and Systems II: Express Briefs, 2022.

[4] H. E. Yantır, A. M. Eltawil, and F. J. Kurdahi, "A hybrid approximate computing approach for associative in-memory processors," IEEE Journal on Emerging and Selected Topics in Circuits and Systems, vol. 8, no. 4, pp. 758–769, 2018.

[5] L. Yavits, A. Morad, and R. Ginosar, "Sparse matrix multiplication on an associative processor," IEEE Transactions on Parallel and Distributed Systems, vol. 26, no. 11, pp. 3175–3183, 2014.

[6] M. A. Neggaz, H. E. Yantır, S. Niar, A. Eltawil, and F. Kurdahi, "Rapid in-memory matrix multiplication using associative processor," in 2018 Design, Automation & Test in Europe Conference & Exhibition (DATE). IEEE, 2018, pp. 985–990.

[7] H. E. Yantir, W. Guo, A. M. Eltawil, F. J. Kurdahi, and K. N. Salama, "An ultra-area-efficient 1024-point in-memory fft processor," Micromachines, vol. 10, no. 8, p. 509, 2019.

[8] R. Kaplan, L. Yavits, R. Ginosar, and U. Weiser, "A resistive cam processing-in-storage architecture for dna sequence alignment," IEEE Micro, vol. 37, no. 4, pp. 20–28, 2017.

[9] T. Higuchi, K. Handa, N. Takahashi, T. Furuya, H. Iida, E. Sumita, O. Oi, and H. Kitano, "The ixm2 parallel associative processor for ai," Computer, vol. 27, no. 11, pp. 53–63, 1994.

[10] K. Pagiamtzis and A. Sheikholeslami, "Content-addressable memory (cam) circuits and architectures: A tutorial and survey," IEEE journal of solid-state circuits, vol. 41, no. 3, pp. 712–727, 2006.

[11] M. Irfan, A. I. Sanka, Z. Ullah, and R. C. Cheung, "Reconfigurable content-addressable memory (cam) on fpgas: A tutorial and survey," Future Generation Computer Systems, vol. 128, pp. 451–465, 2022.

[12] M. Irfan, H. E. Yantır, Z. Ullah, and R. C. C. Cheung, "Comp-tcam: An adaptable composite ternary content-addressable memory on fpgas," IEEE Embedded Systems Letters, vol. 14, no. 2, pp. 63–66, 2022.

[13] Z. Ullah, M. K. Jaiswal, and R. C. Cheung, "Z-tcam: an sram-based architecture for tcam," IEEE transactions on very large scale integration (VLSI) systems, vol. 23, no. 2, pp. 402–406, 2014.

[14] M. Hout, M. E. Fouda, R. Kanj, and A. M. Eltawil, "In-memory multi-valued associative processor," arXiv preprint arXiv:2110.09643, 2021.