Address Generators Units for Bidirectional Linear Processor Array

Mile K. Stojcev*. Tatjana R. Nikolic*.
Emina I. Milovanovic*. Igor Z. Milovanovic*

*Faculty of Electronic Engineering Nis, Aleksandra Medvedeva 14, 18000 Nis, Serbia
(Tel: + 381-18-529660; e-mail: mile.stojcev{tatjana.nikolic, emina.milovanovic, igor.milovanovic}@ elfak.ni.ac.rs)

Abstract: Most current RISC processors spend significant amount of time generating data addresses and accessing memory instead of performing the functional data operations required by the application programs. This is mainly direct consequence of the inadequate support provided by conventional architectures for the access of data types used in current applications. In this article we describe a method how to implement address generation unit (AGU) which acts as an interface between the host processor and processor array composed of a linear chain of identical processing elements. In our case, the processor array is bidirectional linear systolic array (BLSA). The proposed hardware implementation of address transformation gives a speedup of 2.2 with relatively low hardware overhead (in average < 10%).

Keywords: Address generator unit, Hardware accelerator, Processor Array.

1. INTRODUCTION

Nowadays embedded systems are growing at an impressive rate and provide enormous sophisticated applications characterized by having a complex array index manipulation and large number of data accesses. Those applications require specific address generations which general purpose processors can not deliver at a reasonable time. The address manipulation capabilities provided by current RISC processors are very limited, usually restricted by the addition of a base register and perhaps an index register to a fixed address or displacement. In other words, these processors are primarily designed for manipulation with addresses at lowest level of data representation such as bits, bytes and words. On the other hand, most embedded applications are developed using structured HLLs such as C, Ada, Pascal, etc. In such languages, the information is mainly handled in a structured way [Miranda et all, 1998]. For applications including real-time multimedia, medical image processing, digital signal processing and processing in high speed communication systems, the major part of arithmetic statements used in HLL programs written for these applications deals with complex data structures. Manipulations with these structures are not efficiently supported by current RISC architectures. Therefore, compilers must generate considerable amount of code intended for fast manipulations with array data structures, and various other complex data type such as records, etc, [Hertz et all, 2002]. This code imposes considerable overhead on system performance and slow-down the program execution in a great deal. In order to cope with latency of data access, i.e. to speedup address expression evaluation, special hardware building blocks, called address generation units, are designed. The function of AGU is threefold [Grant et all, 1989]. First, during the initialization, it transforms host address space into BLSA address space. Second, provides efficient memory data access during BLSA operation. Third, performs fast data transfer between BLSA and host at the end of the computation [Milovanovic et all, 2000] and [Stojcev et all, 2000]. In this paper, we examine the impact on area and performance of memory access related circuitry in eliminating computational intensive offset address calculations performed in software by implementing the needed address transformations with hardware AGUs.

2. PROGRAM FORMULATION

2.1 Target algorithms

A broad class of problems that can be solved on the BLSA has a form of nested loops such as [Aho et all, 1976] and [Milovanovic et all, 2009]:

Algorithm_1
for k:=1 **to** n **do**
for i:=1 **to** n **do**
for j:=1 **to** n **do**
$c_{ij}^{(k)} := c_{ij}^{(k)} \oplus \left(a_{ik} \otimes b_{ij}\right)$

where $A=\left(a_{ik}\right)$, $B=\left(b_{kj}\right)$ and $C^{(0)}=\left(c_{ij}^{(0)}\right)$ are given matrices, are given matrices, while $C^{(n)}=\left(c_{ij}^{(n)}\right)$ is a resulting matrix. "$\oplus$" and "$\otimes$" are operations from the set {+, -, *, min, max, =, $\wedge$, $\vee$, $\neg$, NOP}.

Some representative problems that can be described in the form of the above algorithm are listed below.

- Case_1. If $c_{ij}^{(0)}=0$ and $(\oplus,\otimes)=(+,*)$ Algorithm_1 corresponds to matrix multiplication algorithm.
- Case_2. If $c_{ij}^{(0)}=1$, $a_{ik}\in\{0,1\}$, $b_{kj}\in\{0,1\}$ and $(\oplus,\otimes)=(\vee,\wedge)$, Algorithm_1 relates to Boolean product of matrices.
- Case_3. If $c_{ij}^{(0)}=1$ and $(\oplus,\otimes)=(\wedge,=)$, Algorithm_1 deals with two-dimensional topple comparison.

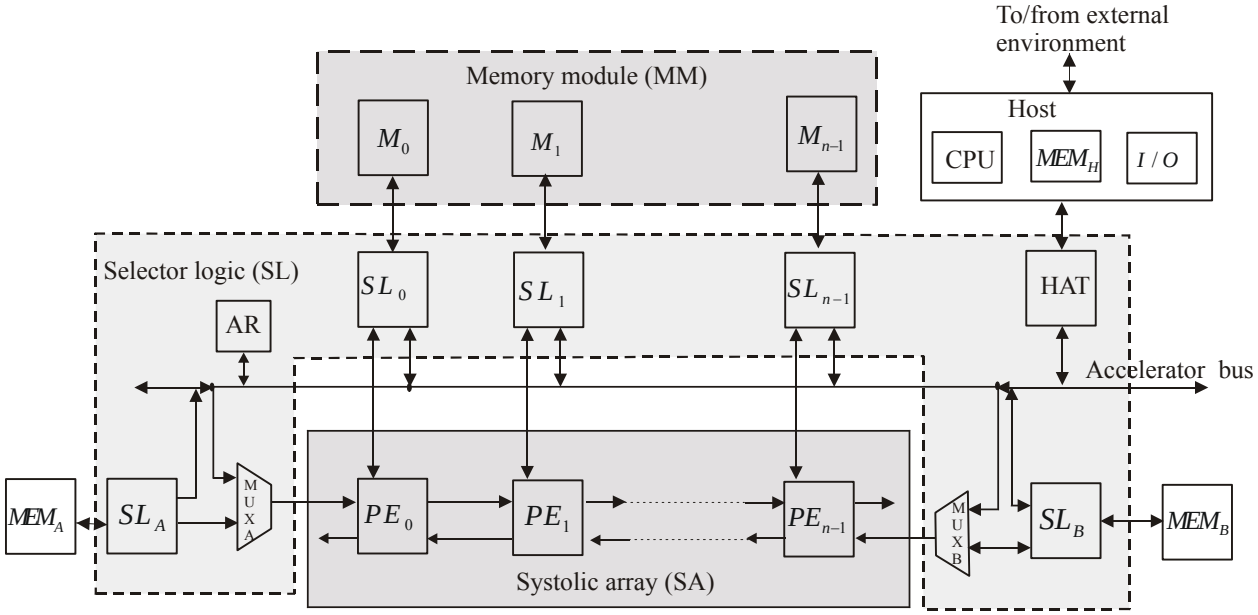

Fig. 1. Global structure of the system

- Case_4. If $c_{ij}^{(0)} = +\infty$ and $(\oplus, \otimes) = (\min, +)$, Algorithm_1 covers distance matrix multiplication algorithm.

A common property of the algorithms that can be represented in the above form is that their data dependency graphs are regular. Consequently, these problems are suitable for implementation on both two-dimensional (2D) and linear (1D) arrays. Primary (design) reasons why we decide to use 1D instead of 2D SA , are the following:

- Number of processing elements (PEs): n in 1D SA, vs. n^2 in 2D SA;
- Simpler I/O interface between the host and SA;
- Number of I/O channels: n+2 for 1D SA vs. 3n in 2D SA;
- Number of AGUs: n+2 for 1D SA vs. 3n in 2D SA.

The main drawback of implementing the above algorithms on 1D SA instead of on 2D SA is longer execution time. Namely, the resulting matrix $C^{(n)}$ is obtained in n iterations, such that input values for the k-th, k=1,2,…, n, are matrices A and B, and matrix $C^{(k-1)}$ obtained in the previous iteration.

2.2 Global system structure

A schematic diagram of the system proposed for implementation of the problems covered by algorithms of type I (Subsection Target algorithms), is given in Fig. 1. Two major components can be distinguished: a host and an accelerator. Accelerator itself is decoupled into data access and execution part. Data access part, consisting of memory modules (*MM*, MEM_A and MEM_B), and selector logic (SL), is responsible for address generation and memory access. Execution part, represented by the BLSA, is responsible for the computation of the particular algorithm. All system constituents are connected via the accelerator bus.

A host controls the operation of the system. Its major functions are: i) initial loading of memory modules *MM*, MEM_A and MEM_B; ii) SL initialization; iii) data transfer from *MM* to host memory (MEM_H); iv) communication with the external environment.

The BLSA consists of n multi-functional processing elements (PEs) denoted with PE_0 through PE_{n-1}, connected in a chain via bi-directional links. Each PE performs an operation of type $c \oplus (a \otimes b)$.

Memory module *MM* is composed of n single port memory blocks, M_0, M_1, ..., M_{n-1}. Each processing element PE_i is connected with a corresponding memory block M_i. During system initialization, the host loads data into *MM*. At the end of a BLSA computation, *MM* holds the resultant matrix. In addition, during initialization, host loads initial matrices A and B into MEM_A and MEM_B, respectively.

The SL acts as an interface among PEs, memory modules and a host. SL defines source/destination of data that are written/read to/from a corresponding memory module. Each address generated by a host is mapped into a corresponding accelerator address. This mapping is of type 1:1. The accelerator memory consists of three address spaces, denoted as *MM*, MEM_A, and MEM_B. As a constituent of SL the host address translator (HAT) performs host-to-accelerator address space mapping. A separate selector logic block, SL_i is accompanied to each memory block M_i, i = 0, 1, ..., n-1. Memory modules MEM_A and MEM_B are connected to selector blocks SL_A and SL_B, respectively. Multiplexor *MUXA* (*MUXB*) selects input a_{in} (b_{in}) for PE_0 (PE_{n-1}). For the algorithms of type I, input data a_{in} and b_{in}, for boundary PEs, are driven from MEM_A and MEM_B, respectively. Memory block M_i, i = 0, 1, ..., n-1, can be accessed by a host via the HAT, and by PEs through SL_i, i = 0, 1, ..., n-1, SLA_A and SL_B selector logic blocks. The arbiter, AR, provides conflict-free access to M_i in the case of simultaneous requests.

More details related to realization of the bidirectional linear systolic array and PE hardware structure can be found in [Milovanovic et all, 2008].

2.3 Constituents of selector logic

The selector logic, SL, acts as an interface between the BLSA, memory modules and a host. There are three types of selector logic blocks, called *HAT*, SL_i and $SL_{A/B}$.

The address part of the selector logic block *HAT* is composed of three building blocks: T_Adr_C - matrix *C* elements address transformer; T_Adr_A - matrix *A* elements address transformer; T_Adr_B - matrix *B* elements address transformer.

The selector logic SL_i acts as an interface between the accelerator-bus protocol and PE-component protocol. The SL_i is active during the following data transfers: a) Host memory to M_i memory module via the accelerator bus - typical for initial loading of M_i; b) M_i to host memory - result transfer at the end of the computation; c) M_i to c_{in} input of PE_i - typical for the start of each P_i machine cycle; d) PE_i to M_i - at the end of PE_i machine cycle.

The selector logic block SL_i is composed of the following three entities: a) $DATA_{I/O}$ - selects data that are written/read to/from memory block M_i; b) ADR_M - generates the M_i's address for read/write operation; c) $CTRL_M$ - generates control signals *R/W*, *CS* and *OE* for memory blocks M_i.

The selector logic $SL_{A(B)}$ is used as an interface between boundary PEs (PE_0 and PE_{n-1}) and memory modules MEM_A, MEM_B. The hardware structure and principles of operation of SL_A and SL_B are identical. The $SL_{A(B)}$ is active during the following data transfers: i) Host memory to MEM_A (MEM_B) memory module via the accelerator bus - during initial loading of MEM_A (MEM_B); ii) MEM_A to input a_{in} of PE_0 (MEM_B to input b_{in} of PE_{n-1}) - at the beginning of each machine cycle. The $SL_{A(B)}$ is composed of three entities: 1) $DATA\_M_{A/B}$ - passes data that are written/read to/from memory module MEM_A (i.e. MEM_B); 2) $ADR_{A/B}$ - selects the address for access to MEM_A (i.e. MEM_B) location; 3) $CTRL_{A/B}$ - generates control signals *R/W*, *CS* and *OE* for memory module MEM_A (i.e. MEM_B).

2.4 Passive and active computations

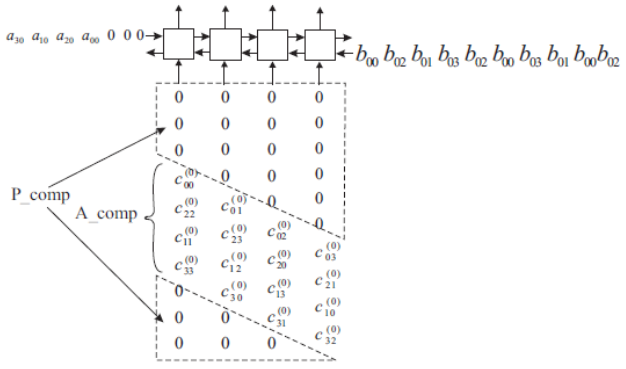

Fig. 2. Data pattern in the BLSA during the first iteration.

The algorithm of type I is executed in n iterations. Each iteration is performed for 3n-2 instruction cycles. From data-flow point of view, each iteration consists of two phases: passive computation, P_comp, and active computation, A_comp, as shown in Fig. 2.

2.5 Address transformations during initial loading of memory modules

Initial loading of memory modules MM, MEM_A i MEM_B is accomplished through a HAT. The HAT transforms host address into accelerator address. The HAT logic consists of three address transformers: T_adr_C, T_Adr_A and T_Adr_B. In general, a host address of element x_{ij} ($x \in \{a, b, c\}$) is mapped into the accelerator address. In order to perform address transformation, a host executes the following sequence of operations

```
LOAD    R1, R0(Mem_h)
STORE   R0(Mem_Acc), R1
```

where R1 and R0 are host registers, with R0 set to zero, Mem_h is a memory location in the host address space, while Mem_Acc represents a memory location in the accelerator address space. The main goal of HAT address transformations is to obtain address pattern structures of logically adjacent elements to be stored in *MM*, MEM_A and MEM_B. In this way the manipulation with elements which drive inputs c_{in}, a_{in} and b_{in} of the PEs is simplified, i.e. elements are read sequentially from *MM*, MEM_A and MEM_B.

Assume, now, that in a host memory, elements of matrices *A*, *B* and *C* are stored in row-major ordering. Memory address Mem_h (Mem_Acc) is composed of three fields (see Fig. 3).

base address	*i*	*j*

Fig. 3. Address format.

The field "base address" points to a starting memory location of the array (matrices *A*, *B* or *C*), either in the host or in accelerator address space. Fields *i* and *j* correspond to the offset of an element x_{ij} ($x \in \{a, b, c\}$) with respect to the base address. The HAT transforms offset part of the address by mapping *i, j* into *i′, j′*. The base address of accelerator memory is fixed and defined by a design. Therefore we will not consider host-to-accelerator base address transformation. In the sequel, the term *address transformation* will refer to the transformation of the offset part of the address only. Further, we assume that $n = 2^k$. In that case, the size of fields *i* and *j* is *k* bits.

Address transformations during MEM_A, MEM_B, and MM loading can be described by the following formulas:

$$\mathrm{Adr\_A} = \left\lfloor \frac{i}{2} \right\rfloor * (i+1) \bmod 2 + \left(\left\lceil \frac{n}{2} \right\rceil + \left\lfloor \frac{i}{2} \right\rfloor \right) * i \bmod 2 + j * n$$

$$\mathrm{Adr\_B} = \left\lfloor \frac{j}{2} \right\rfloor * (j+1) \bmod 2 + \left(\left\lceil \frac{n}{2} \right\rceil + \left\lfloor \frac{j}{2} \right\rfloor \right) * j \bmod 2 + i * n$$

$$(i, j) \mapsto (i'\, j') = ((n - i + j) \bmod n, \mathrm{shuffle}(i)) \qquad (1)$$

$$i, j = 0, 1, \ldots, n-1$$

The address transformations performed by T_Adr_A, T_Adr_B, and T_Adr_C, according to Equation (1), are given in Fig. 4, 5, and 6. As can be seen from Fig. 4 and 5 addresses transformations performed by T_Adr_A and T_Adr_B are implemented by cross-wiring, only, i.e. there is not hardware logic blocks.

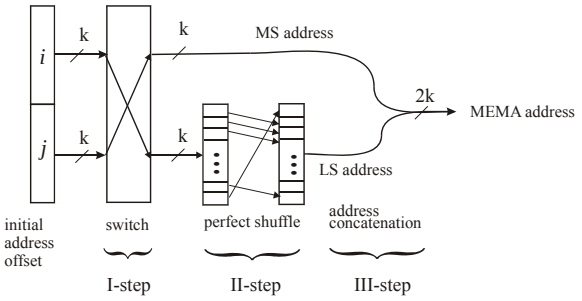

Fig. 4. Address transformation performed by T_adr_A.

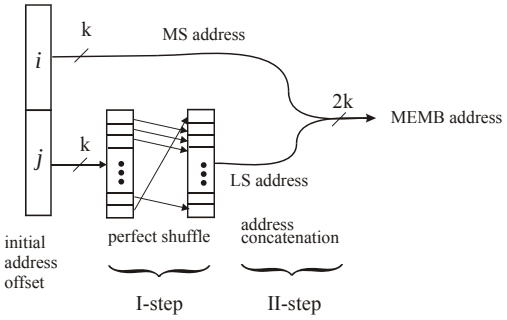

Fig. 5. Address transformation performed by T_adr_B.

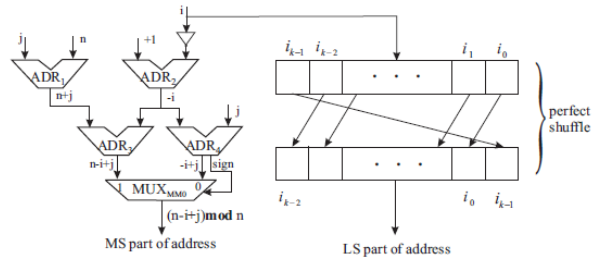

Fig. 6. Address transformation performed by T_adr_C.

2.6 Address generator units for algorithms of type I

Structures of AGUs for accessing memory modules *MM*, MEM_A and MEM_B, are given in Fig. 7, 8 and 9. For algorithms of type I, input operands c_{in}, a_{in} and b_{in} are fetched from *MM*, MEM_A and MEM_B, respectively. Results are stored in *MM*. A single address generator unit AGU C*i* is accompanied to each processing element PE_i, $i = 0, 1, ..., n-1$. The AGU C*i* generates addresses for accessing *Mi*. During each iteration, AGU_C*i* generates addresses for the following data sequence

$$\underbrace{00\ldots0}_{n-i-1}\underbrace{cc\ldots c}_{n}\underbrace{00\ldots0}_{n+i-1}$$

where *c* points to the element of matrix *C* fetched from memory block M_i, which drives input c_{in} of PE_i, $i = 0, 1, ..., n-1$. Sequences denoted as *S*1 and *S*3 are part of the P_com phase, while *S*2 corresponds to A_com phase (see Fig. 2).

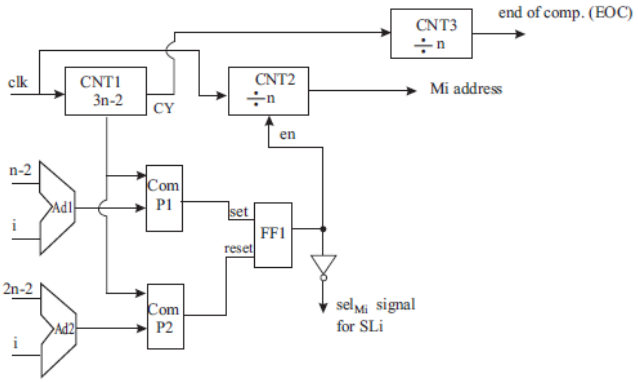

Fig. 7. The structure of AGU_C.

The address generator unit *AGU_A* (*AGU_B*) is constituent of a selector logic $SL_{A/B}$. In essence, there are separate address generator units: *AGU_A* for accessing MEM_A, and *AGU_B* for accessing MEM_B. During each iteration *AGU_A* generates addresses for the following data sequence

$$\underbrace{00\ldots0}_{n-1}\underbrace{aa\ldots a}_{n}\underbrace{00\ldots0}_{n-1}$$

where *a* denotes an element of matrix *A* fetched from MEM_A which drives the input a_{in} of PE_0. Let us note again, that the algorithm is executed in *n* iterations. For each iteration 3*n*-2 instruction cycles are used.

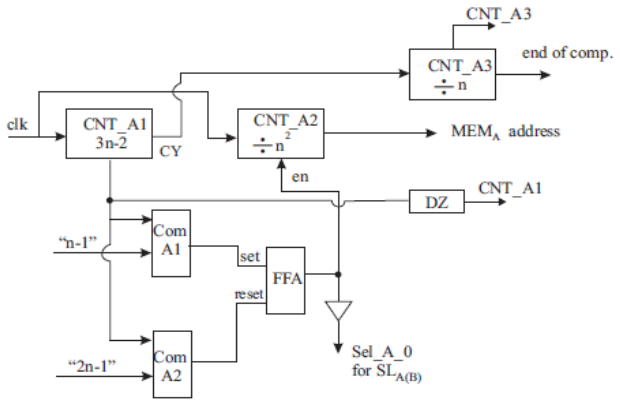

Fig. 8. The structure of AGU_A.

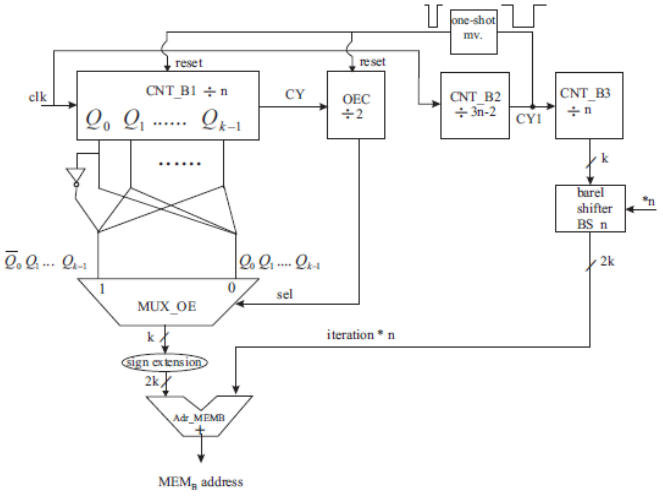

Fig. 9. The structure of AGU_B.

The structure of address generator unit AGU_B is sketched in Fig. 9. The AGU_B consists of four counter blocks, CNT_B1, CNT_B2, CNT_B3, CNT_B4, multiplexor MUX_OE, barrel shifter BS_n, and adder Adr_MEMB. The BS_n shifts left the output of CNT_B3 for *k* bit positions.

3. PERFORMANCE EVALUATION

To evaluate benefits of hardware implementation of AGUs, we will use a quality factor, *Q*, as a metric. We define it as

$$Q = \frac{T_{SW}}{T_{AGU}} \tag{2}$$

where T_{SW} and T_{AGU} correspond total execution time of the algorithm implemented on the BLSA in the case of software and hardware realization of address transformations, respectively. We will consider the amount of quality factor during initial loading of memory modules, execution phase, and a result transfer phase.

During initial loading data are transferred from host to accelerator address space. At the end of the computation data are transferred from memory module *MM* to a host memory. These tasks are performed by the HAT. Illustration only program sequences performed by the host during initial loading of memory modules, that correspond to hardware and software implementations of address transformations are given in Fig. 10 and 11. Without affecting generality, we assume that base address is 16-bits long, while fields *i* and *j* are 8-bits each. In order to simplify the analysis we assume that execution time of all instructions is the same, i.e. equal to a single time unit T_U.

```
  LEA R1, MEM_h              /* Load host address into R1 */
  LEA R2, MEM_acc            /* load acceler. base addr. into R2 */
  ADDI R3, R0, #counter      /* defines number of elements */
Loop:  LD   R4, R0(R1)       /* Load element from host memory */
  SD    R0(R2), R4          /* Store element into acceler. memory */
  ADDI R1, R1, #4           /* increment address */
  SUBI R3, R3, #1           /* Decrement counter */
  BNEZ R3, Loop             /* Test for the end of the loop */
```

Fig. 10. Program sequence for initial loading of memory modules performed by the host when address transformations are performed by HAT.

```
  LEA R1, MEM_A_h            /* Load host base address into R1 */
  LEA R2, MEM_A_acc          /* load acceler. base addr. into R2 */
  ADDI R3, R0, #counter      /* defines number of elements – n^2 */
Loop:  LD  R4, R0(R1)        /* Load element from host mem. */
  ANDI R11, R1, 0x0000FF00          /* Extract i part of offset */
  ANDI R22, R1, 0x000000FF          /* Extract j part of offset */
  SHL R22, R22, #8                  / * swap i and j part
  SHR  R11, R11, #8                    of offset */
  SHR  R11, R11, #1                 /* Performs
  JNC   R11, Lab                       perfect shuffle
  ANDI  R11, R11, #0x00000080          on i */
Lab:  OR   R22, R22, R11       / * concatenate two parts of offset */
  SD   R2(R22), R4           /* Store element into acceler. mem. */
  ADDI R1, R1, #4            /* increment address */
  SUBI R3, R3, #1            /* decrement counter */
  BNEZ R3, Loop              /* Test for the end of the loop */
```

Fig. 11. Program sequence for initial loading of MEM_A when address transformation is performed by the host.

The quality factor of the HAT during the initialization is

$$Q_{HAT\_ini} = \frac{Q_{T\_Adr\_A} + Q_{T\_Adr\_B} + Q_{T\_Adr\_C}}{3} \approx 2.73 \tag{3}$$

The quality factor of the HAT during result transfer phase is given by

$$Q_{HAT\_end} = \frac{\left(4 + 16n^2\right)T_U}{\left(3 + 5n^2\right)T_U} \approx 3.2 \tag{4}$$

The average quality factor of the HAT during initialization and result transfer is obtained as

$$Q_{HAT} = \frac{Q_{HAT\_ini} + Q_{HAT\_end}}{2} \approx 3 \tag{5}$$

The quality factor of the AGUs during BLSA operation is

$$Q_{AGU} = \frac{23n^2 - 10n + 2}{15n^2 - 8n + 2} \approx 1.5 \tag{6}$$

The overall quality factor is obtained as ratio between total execution time of the algorithm when address transformations are performed in software and hardware, respectively. The total execution time of the algorithm includes time for initial loading of MEM_A, MEM_B and *MM*, active execution time of the BLSA, and time needed to transfer results from *MM* to host memory. The overall quality factor *Q* is given by the following formula

$$Q = \frac{\left(70n^2 - 10n + 16\right)T_U}{\left(35n^2 - 8n + 14\right)T_U} \approx 2 \tag{7}$$

This means that by involving hardware AGUs the total execution time of algorithms is decreased by a factor of two. The penalty is paid at increased hardware complexity. The overhead in term of equivalent gate count due to hardware implementation of AGUs for various complexity of PE (16-, 24-, 32-bit) are given in Table 1. At behavioral level a BLSA structure was described using VHDL code. For synthesis, routing and technology mapping a Xilinx development CAD tool ISE WebPack 9.1 was used. The BLSA was implemented on FPGA devices from Xilinx Spartan 3E and Virtex series. We define hardware overhead as a ratio of total number of equivalent gates in the BLSA with and without hardware AGUs.

From Table 1 we can conclude the following: i) The overhead is relatively low (within the range from 1.2% up to 24.9%); ii) As complexity of PE increases, the overhead decreases; iii) As the BLSA size increases, the overhead increases. For example, for the BLSA with 128 16-bit PEs for Virtex series the overhead is 6.4%, while for the BLSA with 256 16-bit PEs it is 7.6%. This means that doubling the array size results in 1.2% overhead increase, only.

On the other hand, the overall execution time of the BLSA is two times shorter. This, in great deal, justifies the usage of hardware AGUs.

Table 1. Hardware complexity of address generator and overhead

FPGA family	Number of PEs	Gate count									Overhead [%]		
								PE					
		AGU_A	AGU_B	AGU_C	T_Adr_C	AGU_MMA	AGU_MMB	8-bit	16-bit	32-bit	OV1	OV2	OV3
Spartan 3E	8	410	379	389	201	176	500	5122	10250	22756	11.7	5.8	2.6
	16	503	517	527	284	255	704				13.0	6.5	2.9
	32	725	608	737	388	328	942				16.2	8.1	3.6
	64	884	702	923	465	407	1301				19.2	9.6	4.3
	128	1046	733	1073	551	477	1426				21.6	10.8	4.9
	256	1070	815	1259	634	533	1776				24.9	12.5	5.6
Virtex	8	422	346	374	198	173	644	7256	18032	49550	8.2	3.3	1.2
	16	593	605	542	281	237	969				9.8	3.9	1.4
	32	791	812	743	385	328	1253				11.8	4.7	1.7
	64	1060	1042	881	462	425	1535				13.1	5.3	1.9
	128	1379	1444	1352	542	474	1901				15.9	6.4	2.3
	256	1609	1758	1493	631	595	2168				19.0	7.6	2.8

4. CONCLUSIONS

A wide variety of arithmetic intensive and scientific computing applications are characterized by a large number of data access. Such applications contain complex offset address manipulations. For most target SA architectures, these memory-intensive applications present significant bottlenecks for system designs in term of memory bandwidth and memory access latencies, which can result in poor utilization of SA computational logic. These time and space techniques require the design of optimized AGUs capable to deal with higher issue and execution rates, larger number of memory references, and demanding memory-bandwidth requirements. In this paper we described efficient AGUs intended for: a) retrieving operands from (n + 2) memory banks in a single machine cycle during SA operation; b) host-to-SA address space data transfer during loading and off-loading of SA, by involving complex address transformations. The results, presented in this paper, indicate that performance gains achievable by involving AGUs are high (the overall quality factor approximately 2.) In general, it is clear that increase in performance comes at the cost of area overhead, which in our case, is relatively minor (in average less than 10 %). For applications where the access patterns are regular, such as in scientific computations, the proposed approach is very executive.

REFERENCES

[Aho et all, 1976] Aho, A. V., Hopcroft, J. E., Ullman, J. D., *The design and analysis of computer algorithm*, Addisom-Weslay Publ. Com., 1976.

[Grant et all, 1989] Grant D., Denyer P., Finlay I., *Synthesis of address generators*, In Proc IEEE Int. Conf. Computer Aided Design, 116-119, 1989.

[Hertz et all, 2002] Hertz M., Hartenstein R., Miranda M., Brockmeyer E., Catthoor F., *Memory Addressing Organization for Stream-based Reconfigurable Computing*, Proc. IEEE ICECS 2002, Dubrovnik, Croatia, 2002.

[Milovanović et all, 2000] Milovanovic E. I., Stojcev M. K., Stojanovic N. M., Milovanovic I. Z., *Matrix-vector multiplication on fixed-size linear systolic array*, Comput. Math. Appl. Vol. 40 (2000), 1189-1203.

[Milovanovic et all, 2008] Milovanovic E. I., Milovanovic I. Z., Stojcev M. K., *A class of low power linear systolic arrays with recongurable pro- cessing elements, In: Supercomputing Research Advances, (Y. Huang, ed.)*, Nova Science Publishers, Inc., New York, 2008, 165-198.

[Milovanovic et all, 2009] Milovanovic E. I., Nikolic T. R., Stojcev M. K., Milovanovic I. Z., *Multi-functional systolic array with reconfigurable micro-power processing elements*, Microelectronics Reliability, (2009), doi:10.1016/j.microrel.2009.03.019

[Miranda et all, 1998] Miranda M. A., et all., *High-level address optimization and synthesis techniques for data-transfer intensive applications*, IEEE Trans. on VLSI Systems, Vol. 6, No 4, (1998), 677-686.

[Stojcev et all, 2000] Stojcev M. K., Djordjevic G. L., Milovanovic E. I., Milovanovic I. Z., *Data reordering converter:an interface block in a linear chain of processor arrays*, Microelectronics J., Vol. 31, 1 (2000), 23-37.

ACKNOWLEDGEMENTS

This work was supported by the Serbian Ministry of Science and Technological Development, project No. TR - 11020 - "Reconfigurable embedded systems".