Address Generators for Linear Processor Array

M. K. Stojčev, I. Ž. Milovanović, E. I. Milovanović, T. R. Nikolić

Abstract – **In processor arrays, the memory subsystem represents a major cost and performance bottleneck. To optimize the system performance we use address generation unit which performs host-to-processor array address transformation in hardware. The aim of initial loading is to provide sequential access to data elements stored in processor array memory modules. The performance of the proposed solution are estimated by the speedup, which is defined as a ratio of the time needed to perform address transformation in software and in hardware. Proposed hardware implementation of address transformation gives a speedup of 2.3, with low hardware overhead. Most of address transformations are performed by cross-wiring**

Keywords – **Processor array, Memory subsystem, Address transformation.**

I. INTRODUCTION

The large number of data and the repetitive nature of the algorithms involved in digital signal processing, numerical analysis, database and dictionary machines, graph algorithms, etc. mean that fast solution to such problems become important, especially for real-time applications. The performance improvement of these algorithms has always been of interest to researchers. [1-3]. As these algorithms are usually loop oriented and data oblivious, they are suitable for hardware acceleration. Processor arrays are good candidate accelerator architectures that can be used in Multiprocessor System on Chip (MPSoC) designs with aim to achieve high computational and communication performance. One of the crucial components that determine the success of the MPSoC based architectures is its memory system. Address calculations in such systems often involve linear and polynomial arithmetic expressions which have to be calculated during program execution under strict timing constraints. Hence, it is very important to carry out these accesses and related address computation in an efficient way.

Some current MPSoC architectures [1] have addresses this problem including a dedicated unit that works in parallel with the processing elements ensuring efficient feed and storage of the data to/from the processor array. These units are referred to as Address Generation Units (AGU).

In many applications the sequence of storage and retrieval for particular blocks of data is strongly patterned. In these cases it is often useful to arrange memory allocation in one of the following ways [2]:

1. The incoming data is written to consecutive locations and the consumed data is read in the required pattern;
2. The incoming data is written in a pre-determined pattern so that reading can proceed from sequential location.

All authors are with the Faculty of Electronic Engineering, A. Medvedeva 14, P.O. Box 73, 1800 Niš, Serbia

In this paper we will use the second approach for implementing address generation unit, which acts as an interface logic between the host and processor array composed of a linear chain of identical processing elements (PEs). In our case, processor array is a bidirectional linear systolic array (BLSA).

II. PROBLEM FORMULATION

Target Algorithms

A broad class of problems that can be solved on the BLSA has a form of nested loops such as [7]:

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

where $A=(a_{ik})$, $B=(b_{kj})$ and $C^{(0)}=(c_{ij}^{(0)})$ are given matrices, are given matrices, while $C^{(n)}=(c_{ij}^{(n)})$ is a resulting matrix. "⊕" and "⊗" are operations from the set {+, -, *, min , max ,=, ∧, ∨, ¬, NOP}.

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

- Case1. If $c_{ij}^{(0)}=0$ and (⊕,⊗) = (+, *) 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 (⊕,⊗)=(∨, ∧), Algorithm_1 relates to Boolean product of matrices [4].
- Case 3. If $c_{ij}^{(0)}=1$ and (⊕,⊗)=(∧, =), Algorithm_1 deals with two-dimensional topple comparison [5].
- Case 4. If $c_{ij}^{(0)}=+\infty$ and (⊕,⊗)=(min ,+), Algorithm_1 covers distance matrix multiplication algorithm [6].

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;

978-1-4244-4383-3/09/$25.00 ©2009 IEEE

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.

Target Architecture

Our target architecture is shown Fig. 1. The BLSA, which acts as a hardware accelerator, is composed of n PEs, denoted as PE_0 to PE_{n-1}, memory modules M_i, i=0,1,…, n-1, connected to each PE_i, and memory modules M_A and M_B connected to border PEs. Memory modules and BLSA form a linear processor array (PA). PA is connected to a host via AGU. AGU performs three address transformations. The first and the second relate to address generation for storing data in M_A and M_B, respectively. The third one deals with address generation for storing data into memory modules M_0 to M_{n-1}.

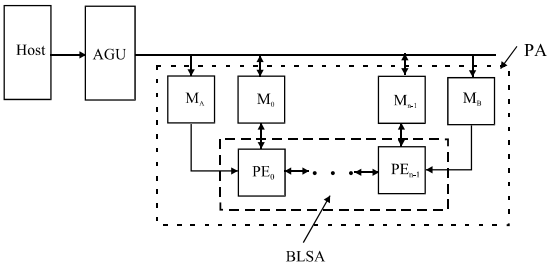

Fig. 1. Global structure of the system

Data pattern needed to feed in BLSA during one iteration is shown in Fig. 2 for the case n=4.

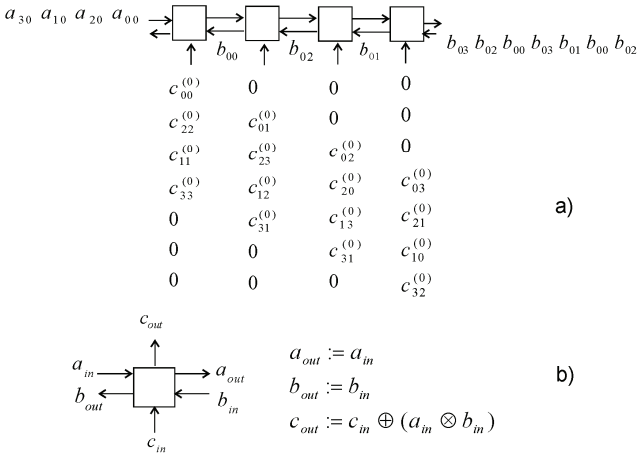

Fig. 2. a) Initial data schedule in the BLSA during first iteration
b) Functional property of PE

Memory Allocation

Memory allocation is a task that assigns data to memory modules. Our goal was to provide initial loading of memory modules such that during algorithm execution data are read from successive memory addresses. We assume that matrices are stored in row-major ordering in host memory. Memory address is composed of three fields as presented in Fig. 3.

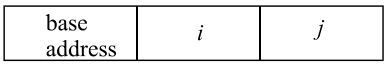

Fig. 3. Memory address format

A field "base address" points to the starting memory location of the array (A, B or C), either in the host or in the accelerator address space. Fields i and j correspond to the offset of the element $x_{ij} \in \{a,b,c\}$, with respect to the base address. The AGU transforms offset part of the address by mapping (i,j) into (i',j'). Accelerator memory base address is defined by the design. We assume that $n=2^k$. In that case the size of fields i and j is k bits. Fig. 4 shows an example of memory allocation for the matrices of size 4x4.

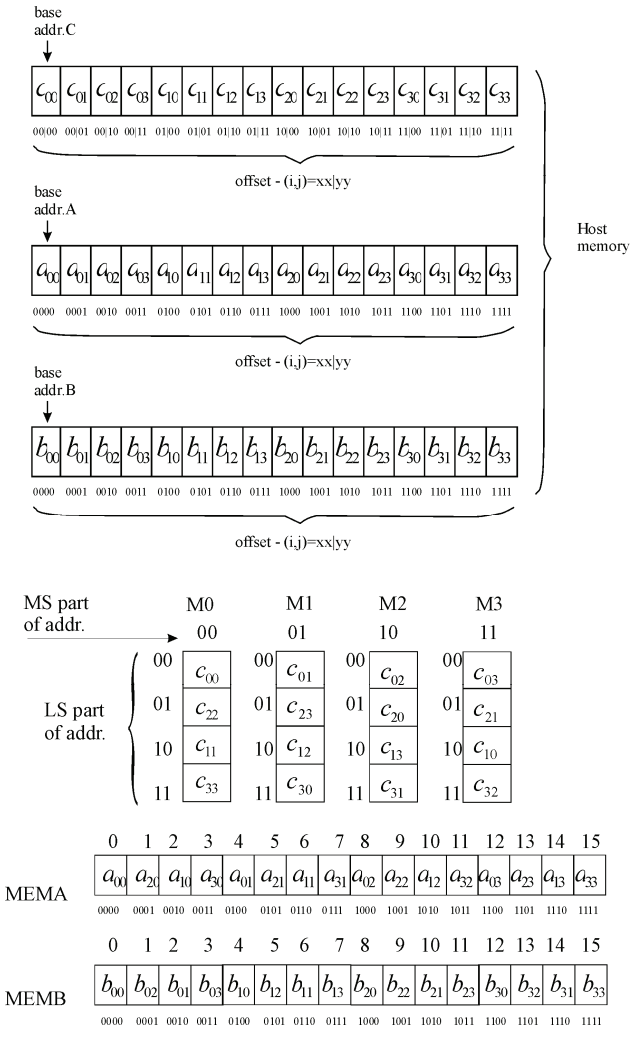

Fig. 4. An Example of memory allocation for matrices of order 4x4

III. ADDRESS GENERATION UNITS

A task of AGU is to perform mapping from host to accelerator address space. Three different types of address transformations are performed during initial loading of memory modules:

- Host memory to M_A address transformation
- Host memory to M_B address transformation

- Host memory to M_i, i=0,1,...,n-1, address transformation

Accordingly, AGU is composed of three parts: AGU_A, AGU_B and AGU_C.

Initial Loading of M_A

Host memory to M_A address transformation, performed by AGU_A, can be described by the following equation

$$Adr\_A = \overbrace{\left\lfloor \frac{i}{2} \right\rfloor * (i+1) \bmod 2 + \left(\left\lfloor \frac{n}{2} \right\rfloor + \left\lfloor \frac{i}{2} \right\rfloor \right) * i \bmod 2}^{S} + j * n,$$

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

The term denoted by S in the above equation represents a perfect shuffle of i, while $j*n$ corresponds to shifting j for k bit positions left (note that $n=2^k$). The structure of AGU_A is depicted in Fig. 5. In the first step i and j part of the address are interchanged by cross-wiring. In the second step, a perfect shuffle on k bits is performed, also by cross-wiring. In the third step a concatenation of MS and LS part of the address is performed and memory address for accessing MA is obtained.

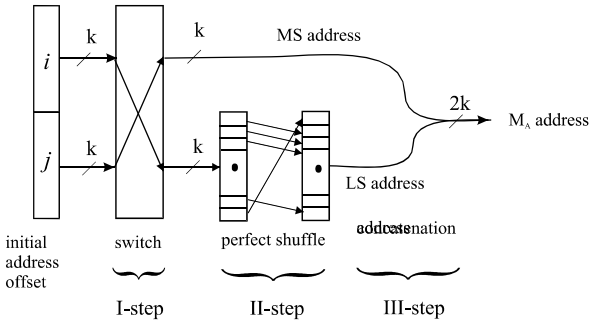

Fig. 5. The structure of AGU_A

Let us note that address transformation is realized without processing. The propagation delay through AGU_A is determined by wiring delay.

Initial Loading of M_B

The AGU_B performs address transformation during initial loading of matrix B into MB. The process of address transformation can be described by the following equation

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

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

The structure of AGU_B is depicted in Fig. 6. It consists of two steps: a perfect shuffle of the lower part of the offset, followed by the concatenation with the upper part. Again, for address transformation only cross wiring is used. The propagation delay through AGU_B is determined by wiring delay.

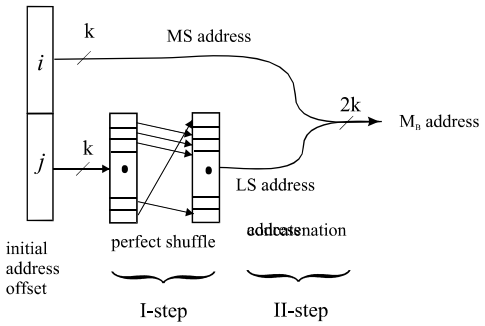

Fig. 6. Hardware structure of AGU_B

Initial Loading of M_i

AGU_C is used as a hardware block for transforming host address of matrix C element into M_i, i=0,1,...,n-1, address. The process of address transformation is described by the following relation

$$(i, j) \rightarrow (i', j') = ((n - i + j) \bmod n, \text{shuffle}(i)) =$$
$$(\text{MS\_part}, \text{LS\_part})$$

where MS_part represents MS part of the address which is used for a selection of memory module M_i, while LS_part defines address location within M_i. The hardware structure of AGU_C is given in Fig. 7. A propagation delay through AGU_C is determined by the delay through a chain of adders and a multiplexer.

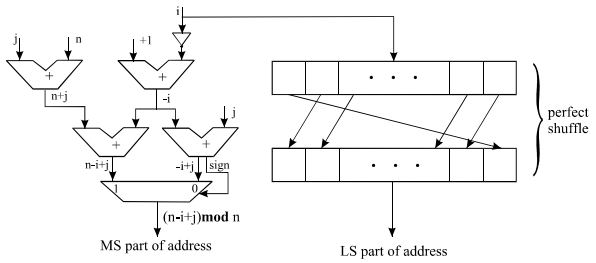

Fig. 7. Hardware structure of AGU_C

IV. Performance Evaluation

In order to evaluate benefits of hardware implementation of address generation units, we will use speedup as a metric. We define speedup as

$$S = \frac{T_{SW}}{T_{AGU}},$$

where T_{SW} and T_{AGU}, correspond to software and hardware implementation of address transformations. The program sequences performed by the host that correspond to hardware and software implementations of address transformations are given in Figs. 8-10. When address transformations are performed by AGU, a program sequence is identical for all matrices (i.e. for A, B and C) In accordance with Fig. 3 we assume that base address is 16-bits long, while fields i and j are 8-bits each. This implies that maximum array size is 256x256=64K elements.

```
LEA   R1, MEM_h          /* Load host base address into R1 */
LEA   R2, MEM_acc        /* load accelerator base address 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 accelerator memory */
ADDI R1, R1, #4           /* increment address to access next
ADDI R2, R2, #4               element of the array */
SUBI R3, R3, #1           /* Decrement counter */
BNEZ R3, Loop             /* Test for the end of the loop */
```

Fig. 8. Program sequence for initial loading of memory modules performed by the host when address transformations are performed by AGU

```
LEA   R1, MEM_A_h         /* Load host base address into R1 */
LEA   R2, MEM_A_acc       /* load accelerator base address into R2 */
ADDI R3, R0, #counter     /* defines number of elements  - n^2*/
ADDI  R6, R0, #k          /* k=ln n */
Loop:   LD    R4, R0(R1)          /* Load element from host memory */
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 paert
SHR  R11, R11, #8             of offset */
SHR  R11, R11, #1         /* Performs
JNC   R11, Lab                perfect shuffle
ANDI  R11, R11, R6            on  I */
Lab:    OR    R22, R22, R11       / * concatenate two parts of offset */
SD    R2(R22), R4         /* Store element into accelerator memory */
ADDI R1, R1, #4     /* increment address to  access next  element of array */
SUBI R3, R3, #1           /* decrement counter */
BNEZ R3, Loop             /* Test for the end of the loop */
```

Fig. 9. Program sequence for initial loading of MA when address transformation is performed by software

```
LEA   R1, MEM_C_h         /* Load host base address into R1 */
LEA   R2, MEM_C_acc       /* load accelerator base address into R2 */
ADDI R3, R0, #counter     /* defines number of elements  - n^2*/
ADDI  R5, R0, #n          /* n is dimension of matrix */
Loop:   LD    R4, R0(R1)          /* Load element from host memory */
ANDI R11, R1, 0x0000FF00  /*  Extract i part of offset */
ANDI R22, R1, 0x000000FF  /* Extract j part of offset */
SHR  R11, R11, #8         /* shift i for 8 positions to the right */
SHR  R33, R11, #1         /* Performs
JNC   R11, Lab                perfect shuffle
ANDI  R33, R33, #0x00000080       on  i  -  LS part of address is obtained*/
Lab:     ADD   R22, R22, R5           /* n+j is obtained */
SUB   R22, R22, R11       /* n+j-i is obtained */
AND   R22, R22, #0x000000FF  /* (n+j-i)mod n  is obtained */
SHL    R22, R22, #8       /* MS part of address is obtained */
OR    R22, R22, R33       / * concatenate MS and LS part of address */
SD    R2(R22), R4         /* Store element into accelerator memory */
ADDI R1, R1, #4     /* increment address to  access next  element of array */
SUBI R3, R3, #1           /* decrement counter */
BNEZ R3, Loop             /* Test for the end of the loop */
```

Fig. 10. Program sequence for initial loading of Mi, i=0,1,…,n-1, when address transformation is performed by software

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. According to Figs. 8 and 9, we have that speedup of AGU_A is

$$S_{AGU\_A} = \frac{T_{SW\_A}}{T_{AGU\_A}} = \frac{(3+13n^2)T_U}{(3+6n^2)T_U} \approx 2.2.$$

Similarly, for AGU_B and AGU_C we obtain

$$S_{AGU\_B} = \frac{(3+12n^2)T_U}{(3+6n^2)T_U} \approx 2., \; S_{AGU\_C} = \frac{(4+16n^2)T_U}{(3+6n^2)T_U} \approx 2.7$$

The total speedup is obtained as

$$S = \frac{T_{SW\_A} + T_{SW\_B} + T_{SW\_C}}{T_{AGU\_A} + T_{AGU\_B} + T_{AGU\_C}} = \frac{(10+41n^2)T_U}{(9+18n^2)T_U} \approx 2.3.$$

Having in mind that address transformations performed by AGU_A and AGU_B are completely realized by cross-wiring, while AGU_C is partially realized by cross-wiring and partially by hardware logic (composed of four adder blocks and a multiplexer), we can conclude that hardware overhead of AGU is very low. On the other hand hardware implementation of address transformations is more than two times faster than software implementation, what in great deal justifies the usage of AGU.

V. CONCLUSION

We have described an address generation unit (AGU) intended for initial loading of memory modules in a linear processor array which acts as a hardware accelerator for a class of problems which can be described in the form of three nested loops. AGU is partially implemented by cross-wiring and partially by hardware logic. The speedup achieved by hardware implementation of address transformation is three.

REFERENCES

[1] S. Pasricha, N. Dutt, "A Framework for Cosynthesis of Memory and Communication Architectures for MPSoC", *IEEE Trans. on CAD of Integrated Circuits and Systems*, Vol. 26, No 3, pp.408-420, 2007.

[2] M. Kandemir, N. Dutt, "Memory Systems and Compiler Support for MPSoC Architectures", in *Multiprocessor Systems-on-Chip*, edited by A. Jerraya and W. Wolf, Morgan Kaufmann Publishers, 2004, pp.251-281.

[3] D. Grant, P. Denyer, I. Finlay, "Synthesis of Address Generators", In *Proc. IEEE Int. Conf. Computer Aided Design*, pp.116-119, 1989.

[4] A.V. Aho, J.E. Hopcroft, J.D. Ullman, *The Design and Analysis of Computer Algorithm*, Addisom-Weslay Publ. Com., 1976.

[5] Li, B.W. Wah, "The Design of Optimal Systolic Arrays", *IEEE Trans. Comput.*, Vol. C-34, 1 (1985), pp.66-77.

[6] T. Tokaoka, "Subcubic Cost Algorithms for the All Pairs Shortest Path Problem", *Algorithmica*, 20 (1998), pp.309-318.

[7] E.I. Milovanović, I.Ž. Milovanović, M.K. Stojčev, "A Class of Low Power Linear Systolic Arrays with Reconfigurable Processing Elements", In: *Supercomputing Research Advances*, (Y. Huang, ed.), Nova Science Publishers, Inc., New York, 2008, pp.165-198.