J Sign Process Syst
DOI 10.1007/s11265-008-0174-x

OpenAccess

High-bandwidth Address Generation Unit

Carlo Galuzzi · Chunyang Gou · Humberto Calderón · Georgi N. Gaydadjiev · Stamatis Vassiliadis

Received: 22 October 2007 / Accepted: 4 March 2008
© The Author(s) 2008

Abstract In this paper we present an efficient data fetch circuitry to retrieve several operands from a *n*-way parallel memory system in a single machine cycle. The proposed address generation unit operates with an improved version of the low-order parallel memory access approach. Our design supports data structures of arbitrary lengths and different odd strides. The experimental results show that our address generation unit is capable of generating eight $32 - bit$ addresses every 6 ns for different strides when implemented on a VIRTEX-II PRO xc2vp30-7ff1696 FPGA device using only trivial hardware resources.

Keywords Address generation unit · Parallel memory · Stride

1 Introduction

Vector architectures were originally adopted for building supercomputers in 1970s [1]. Since then, they have played a central role in conventional supercomputers and have flourished for almost two decades till the appearance of the "vector aware" microprocessors in the early 1990s. The most successful ones include the Cray series [2] and the NEC SX series [3].

Nowadays, with the advance of the integration technology, more and more transistors can be integrated on a single chip. Consequently, the SIMD processing paradigm (including vector execution model) regains its position in the organization of on-chip computation resources, such as the General Purpose Processors (GPP) SIMD enhancements [4–6] and vector accelerators in more specific application domains such as home entertainment [7] and computer graphics [8]. In order to provide sufficient and sustained bandwidth and to reduce memory access latency, parallel (multibank) memories are widely used in these SIMD processing systems [2, 3, 8]. A fast and efficient address generation for the memory subsystem is one of the key issues in system design when many memory banks work in parallel. This paper deals with the address generation problem of on-chip vector accelerators. More specifically, the main contributions of this paper are the following:

- the design of an address generation unit (AGEN) capable of generating eight $32 - bit$ addresses in a single machine cycle. Arbitrary memory sequences are supported by using only one *special purpose instruction*;
- an organization that uses optimized Boolean equations to generate the offsets instead of including an additional adder stage;
- an FPGA implementation of the proposed design able to fetch 1.33 Giga operands per second from an

Prof. Dr. Stamatis Vassiliadis has passed away.

C. Galuzzi (✉) · C. Gou · H. Calderón · G. N. Gaydadjiev · S. Vassiliadis
Computer Engineering Laboratory, Electrical Engineering, Mathematics and Computer Science Faculty, TU Delft, Delft, The Netherlands
e-mail: C.Galuzzi@tudelft.nl

C. Gou
e-mail: Chunyang@tudelft.nl

H. Calderón
e-mail: H.Calderon@tudelft.nl

G. N. Gaydadjiev
e-mail: G.N.Gaydadjiev@tudelft.nl

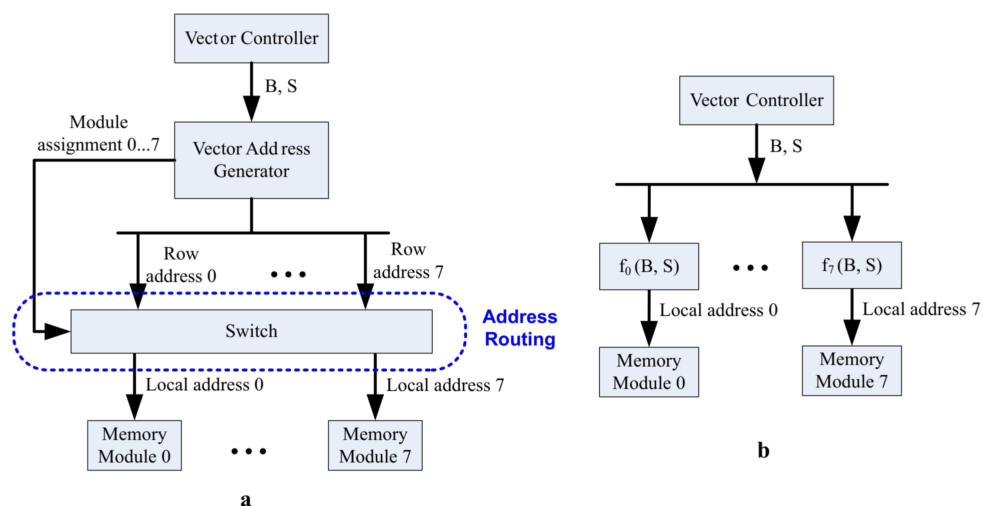

Figure 1 Two paradigms of vector address generation: **a** centralized, **b** distributed.

8-way-parallel memory system using only 3% of the slices and only 4% of the lookup tables (LUTs) of the targeted device, the VIRTEX-II PRO xc2vp30-7ff1696 FPGA.

The paper is organized as follows. In Sections 2 background information and related works are provided. In Section 3 and 4, the basic definitions and the design of the proposed AGEN are presented. An outline of the experimental results is given in Section 5. Concluding remarks and an outline of conducted research are given in Section 6.

2 Background and Related Work

One of the main issues in vector processing systems with multi-bank or multi-module memories is the address generation. There are two major paradigms for address generation: centralized and distributed. Whenever a vector memory access command comes in centralized address generation, the base address as well as the stride[1] are sent to the AGEN, just as shown in Fig. 1a. The centralized address generation paradigm is widely used in traditional vector supercomputers. In centralized AGEN, local addresses of all memory modules are calculated and distributed to the corresponding modules using the address routing circuit (e.g. a crossbar) under the guidance of the corresponding module assignments. In the distributed AGEN, instead of sending individual addresses to each memory module, Corbal et al. proposed a Command Vector Memory (CVM) System [9] that broadcasts the vector access command to all memory modules simultaneously. The local addresses of each memory module are then calculated independently by the local module controller, as illustrated in Fig. 1b. Note that in Fig. 1, $f_i(B, S)(i = 0, 1, \ldots, 7)$ are the local address generation functions where B and S denote the vector base and stride, respectively. It is shown in Fig. 1 that the address routing circuitry is successfully removed by the distributed address generation scheme. Additionally, in [9] the authors have demonstrated the feasibility of this memory organization with simulations at the architectural level of a CVM coupled with an out-of-order vector processor.

There are two clear advantages using the distributed address generation. First, the local address routing circuit can be substantially simplified (usually from a crossbar to a bus), that potentially allows for shorter data access latencies. Second, since there can be more than one memory address seen by the local module controller, a distributed address generation can exploit in a better way the internal features of the module to improve performance. This is the case, for example, when there are more than one access going to the same memory module on a long vector access command or when multiple vector commands are queued in the module controller. In the CVM system, each memory module is a commodity SDRAM module with internal banks and row buffers. The performance of the memory system can therefore be enhanced by properly and independently scheduling of the address sequences on each module to increase the row buffer hit rate.

An extension of the work presented in [9] is proposed by Mathew et al. in [10, 11] where the authors

[1] A formal definition of stride is presented in Section 3. Anyhow, the stride is the address distance between two consecutive data elements stored in a memory system.

proposed a Parallel Vector Access unit (PVA). This unit is a vector memory subsystem that provided strided data accesses with high efficiency on a multi-module SDRAM memory. The PVA unit adopts the same idea of the CVM system when it transmits vector commands instead of individual addresses to memory modules, and lets the modules independently determine which addresses are requested from them and how to provide these requests. The authors, not only provided architectural simulation and the hardware design of the memory control unit but they also validated the design through gate-level simulation.

In order to perform PVA in command-based memory system, one of the key points is the use of distributed high-bandwidth address generation mechanisms capable of inferring which addresses are requested from the local memory module. However, there exist no progress work reported in the literature that focuses on the study of address generation circuitries for distributed address generation. The high-bandwidth AGEN proposed in this paper unravels the design issues by providing a case study for the distributed address generation.

Besides memory address generation, another critical design consideration in multi-bank memory systems is the distribution of the data within the different memory banks to ensure maximum benefits from multi-bank memory organization. One simple low-order interleaving scheme which maps address a to bank (a *mod* m) at local address (offset) (a *div* m) is easy to implement, when m is a power of two, and the stride equals one.[2] Unfortunately, when the stride is different than one, there is a large performance degradation due to bank conflicts [12]. To cope with this problem, many researchers focused their work on parallel memory schemes. Two categories are identified: memory systems with redundancy and memory system without redundancy [13]. An illustrative example of the first category is represented by the *prime memory systems*. These systems are used by many researchers and research projects [14–17] and are built upon previous work on prime numbers. The drawback of these systems is the modulo operation on prime numbers which is difficult to implement efficiently in hardware. An example of the second category is represented by the *skewing memory systems* including the widely used row rotation schemes and XOR schemes [18–24].

The focus of our paper is on the high-bandwidth address generation circuitry for multi-bank memory systems with low-order interleaving scheme. Although the details of vector address generation is closely related to the memory schemes that the vector processing systems adopt, we believe that the methodologies presented in this paper may also be of interest for address generation circuitry under other parallel memory schemes.

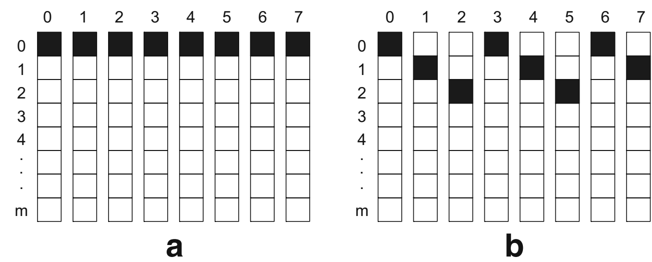

Figure 2 An example of a memory system composed by 8 banks of memory, each one holding the same number of memory cells, where 8 data are stored in two different ways: **a** in a consecutive way, **b** leaving a fix number of cells between the data.

3 Theoretical Background

In this section, we begin by introducing a motivating example to informally outline the main concepts of the proposed method. Thereafter we present the theoretical foundation of our approach.

3.1 Motivating Example

Let us consider a memory system composed by n banks of memory each holding m memory cells. The $m \times n$ memory cells can be represented by a matrix $[m \times n]$ where each column corresponds to a memory bank. Let us consider the data $a_0, ..., a_{n-1}$ to store in the main memory system. Let us assume the element a_0 is identified by the pair of indexes (i_0, j_0) in the matrix representing the memory. The data are stored per row in a circular way: when a data a_α is stored in position $(i_\alpha, n-1)$ then $a_{\alpha+1}$ is stored in position $(i_\alpha + 1, 0)$. Let us assume to store the data leaving a certain number of cells S between two consecutive data elements. Then a_α is identified by the pair of indexes: $i_\alpha = i_{\alpha-1} + (j_{\alpha-1} + S) \text{ div } n$ and $j_\alpha = (j_{\alpha-1} + S) \mod n$.

In Fig. 2, we present two examples of storing 8 data elements in a memory system composed by 8 memory banks, each one holding the same number of memory cells. In Fig. 2a the data are stored consecutively whereas in Fig. 2b the data are stored leaving a fixed number of cells between two consecutive elements. As the figure shows, the 8 elements belong to different

[2]Given two integers a and b we have $a = bq + r$ where a div $b = q$ is the quotient and $a \mod b = r$ is the rest, with $a, b, q, r \in \mathbb{N}$ and q, r univocally determined.

memory banks. This means that 8 data can be read in parallel without problem of conflict in a single cycle if we are able to design an AGEN capable of generating the eight addresses.

3.2 Theoretical Foundation of the AGEN Design

Definition 1 We call **stride** the address distance between two consecutive data elements stored in a memory system.

Roughly speaking, the address distance between two consecutive data is represented by the number of cells separating the two elements per row as described above. The stride is clearly an integer number.

Let us consider n banks of memory each one holding m memory cells. The $m \times n$ memory cells can be represented as a matrix $[m \times n]$ where each column corresponds to a memory bank. In addition, the memory cell i of the memory bank j corresponds to the matrix element with indexes (i, j). We denote this matrix as A and consider $n = 2^h$ and m for its dimensions, with $h, m \in \mathbb{N}$. Additionally, the stride is an integer $Str = 2q + 1, q \in \mathbb{N}$. From now on, the data stored in the memory banks will be considered as matrix A elements. Let the n consecutive data placed in the memory banks be denoted by:

$$a_0, ..., a_{n-1}. \tag{1}$$

Remark 1 Every element a_α, with $\alpha = 0, ..., n-1$, is identified in the matrix by its row-index i, with $i = 0, 1, ..., m-1$, and its column-index j, with $j = 0, 1, ..., n-1$. Additionally, the pair of indexes (i_α, j_α) can be used to represent a_α as a number in base n, obtainable as juxtaposition of i_α as most significant digit and j_α as least significant digit. The two indexes can also be used in a base 10 representation. Therefore, we have the following chain of equivalent representations for a_α:

$$a_\alpha \leftrightarrow (i_\alpha, j_\alpha) \leftrightarrow (i_\alpha j_\alpha)_{|n} \leftrightarrow (ni_\alpha + j_\alpha)_{|10}. \tag{2}$$

The usefulness of this result will be more clear in the proof of Theorem 1. As an example, Table 1 shows the pair of indexes identifying the data in the matrix for $n = 8$ and $Str = 3$ (Fig. 2b), assuming $(i_0, j_0) = (0, 0)$.

Remark 2 Without loss of generality, we can assume that the first element a_0 stored in the matrix remains at position $(i_0, j_0) = (0, 0)$.

Lemma 1 *The number of rows necessary to hold n elements with stride $Str = 2q + 1, q \in \mathbb{N}$ is Str.*

Table 1 Correspondence $a_\alpha \leftrightarrow (i_\alpha, j_\alpha) \leftrightarrow a_{\alpha|n} \leftrightarrow a_{\alpha|10}$ for $n = 8$ and $Str = 3$.

Element a_α	Row-index i_α	Column-index j_α	$a_{\alpha\|8}$	$a_{\alpha\|10}$
a_0	0	0	00	0
a_1	0	3	03	3
a_2	0	6	06	6
a_3	1	1	11	9
a_4	1	4	14	12
a_5	1	7	17	15
a_6	2	2	22	18
a_7	2	5	25	21

Proof The number of cells ($\sharp_{cell}$) necessary to store n elements with stride Str is $\sharp_{cell} = n + (Str - 1)\, n = n(2q + 1)$. Therefore, the number of rows is:

$$\sharp_{cell} \mod n = n(Str) \mod n = Str. \tag{3}$$

□

Remark 2 and Lemma 1 imply that the necessary rows to store the n elements with stride Str are:

$$\{0, 1, ..., Str - 1\} \tag{4}$$

As an example, in Fig. 2b it is possible to see that the eight data are stored in 3 rows ($Str = 3$).

The n data a_α can be defined recursively. If $a_0 = (i_0, j_0)$ the elements $a_2, ..., a_{n-1}$ can be recursively defined as follows:

$$a_\alpha = a_{\alpha-1} + Str. \tag{5}$$

This is equivalent to the following: given $a_{\alpha-1}$ identified by the pair of indexes $(i_{\alpha-1}, j_{\alpha-1})$, a_α is identified by the pair of indexes:

$$i_\alpha = i_{\alpha-1} + (j_{\alpha-1} + Str) \text{ div } n$$
$$j_\alpha = (j_{\alpha-1} + Str) \mod n \tag{6}$$

Theorem 1 *Let n be the number of elements a_α, with $\alpha = 0, ..., n-1$, stored in a matrix A, $m \times n$, with $n = 2^h$. Let the stride be the integer $Str \in \mathbb{N}$. If (i_α, j_α) and (i_β, j_β) are the index pairs identifying a_α and a_β in the matrix and $\gcd(n, Str) = 1$, we have:*

$$j_\alpha \neq j_\beta \qquad \forall \alpha, \beta \in [0, ..., n-1]. \tag{7}$$

Proof Without loss of generality, by Remark 2, we can assume $(i_0, j_0) = (0, 0)$. By contradiction let $j_\alpha = j_\beta$. We have two possible cases: (1) $i_\alpha = i_\beta$ and (2) $i_\alpha \neq i_\beta$.

The first case does not occur: more precisely, if $i_\alpha = i_\beta$, the assumption $j_\alpha = j_\beta$ leads to $a_\alpha = a_\beta$ (see Remark

1). In the second case: $i_\alpha \neq i_\beta$. Firstly, by Eq. 4, it follows:

$$i_\beta - i_\alpha \in [0, Str - 1]. \quad (8)$$

Without loss of generality we can assume $\beta > \alpha$. By Eq. 5 we have:

$$a_\beta = a_{\beta-1} + Str = a_{\beta-2} + 2Str = \ldots = a_\alpha + xStr, \quad (9)$$

with $x \in \mathbb{N}$ and $x < n$; it is straightforward to show that $x = \beta - \alpha$. By using the representation in base 10 of a_α and a_β (see Eq. 2), Eq. 9 becomes:

$$ni_\beta + j_\beta = ni_\alpha + j_\alpha + x\ Str, \quad (10)$$

taking into account the assumption $j_\alpha = j_\beta$ we can rewrite Eq. 10 as

$$n(i_\beta - i_\alpha) = x\ Str. \quad (11)$$

Since $gcd(n, Str) = 1$ and n divides the product $x\ Str$, it follows that n is a non trivial divisor of x (i.e. a divisor different from 1 and n). This implies that: $x = r\,n$, with $r \in \mathbb{N}$. It follows $x > n$ which contradicts the original hypothesis. As a consequence, it must be that $j_\alpha \neq j_\beta$, for all $\alpha, \beta \in [0, \ldots, n-1]$. □

The previous theorem implies the following:

(S): *n data stored in n memory banks can be accessed in parallel in a single machine cycle if n is coprime with the stride.*

This because by Theorem 1 each data is stored in a different memory bank. This holds, in particular, if $n = 2^h$ and Str is an odd integer and viceversa, n an odd integer and $Str = 2^h$.

Example 1 Let us consider Fig. 2b and Table 1. In this case $n = 8$ and Str= 3. In Column 3 of Table 1, it is possible to see that each data stored in the system is allocated in a different memory bank. This follows by Theorem 1. If there exist two elements a_α, a_β with the same column index then there exists $x < 8$ such that: $n(i_\beta - i_\alpha) = x(2q+1)$ ($q = 1$ in this case). Considering that $n = 8$ in our example, $n(i_\beta - i_\alpha)$ can be either 8 or 16. The difference cannot be 0 since in that case $i_\alpha = i_\beta$ and therefore $a_\alpha = a_\beta$. As a consequence, we have two cases $8 = 3x$ or $16 = 3x$ and both equations do not have an integer solution for x.

Remark 3 Let the stride $Str = 2q + 1$ be and odd integer. Let a_α and a_β be two consecutive data located in the same memory bank. Then $|i_\beta - i_\alpha| = Str$.

Proof Let a_α and a_β be identified by the pairs of indexes (i_α, j_α) and (i_β, j_β) respectively with $j_\alpha = j_\beta$. Without loss of generality we can assume $i_\beta > i_\alpha$. By the proof of Theorem 1, we know that if $j_\alpha = j_\beta$, then:

$$n(i_\beta - i_\alpha) = x\ Str. \quad (12)$$

Therefore

$$(i_\beta - i_\alpha) = \frac{x}{n}\ Str. \quad (13)$$

Since $i_\beta - i_\alpha \in \mathbb{N}$, it means that $x = pn$ with $p \in \mathbb{N}$. Since a_α and a_β are two consecutive data located in the same memory bank, it means that $x = \min_{p\in\mathbb{N}} pn = n$. Then

$$(i_\beta - i_\alpha) = \frac{n}{n}\ Str = \ Str. \quad (14)$$

□

Previous remark can be reformulated saying that if the stride is an odd integer coprime with the number of memory banks, the distance between two consecutive data belonging to the same banks of memory is equal to the stride and independent by i and j.

Theorem 3 *Let a_0 be identified by the pair of indexes (i_0, j_0) with $j_0 = 0$ and let Str be an odd stride. For an element a_α we have that*

$$i_\alpha = BS + A_\alpha \quad (15)$$

where $BS = i_0 + k\ Str$ with $k \in \mathbb{N}$ and $A_\alpha = \phi(n, j_\alpha, Str)$ is a function of n, j_α and Str.

Proof By Eq. 6 we can easily obtain the following:

$$i_\alpha = i_0 + (j_0 + \alpha Str) \text{ div } n$$
$$j_\alpha = (j_0 + \alpha Str) \mod n \quad (16)$$

Additionally, we know that from the algorithm of division: given two integers a and b we have $a = bq + r$ where a div $b = q$ is the quotient and $a \mod b = r$ is the rest, with $a, b, q, r \in \mathbb{N}$ and q, r univocally determined. Combining these two results, if we consider $a = j_0 + \alpha Str$ and $b = n$, we obtain the following:

$$j_0 + \alpha Str \ = \ n(i_\alpha - i_0) + j_\alpha, \quad (17)$$

and therefore we can obtain i_α as follows:

$$i_\alpha \ = i_0 + \frac{j_0 - j_\alpha + \alpha Str}{n}. \quad (18)$$

Adding and subtracting the quantity $k\ Str$, since $BS = i_0 + k\ Str$, we have the following:

$$\begin{aligned} i_\alpha &= i_0 + k\ Str + \frac{j_0 - j_\alpha + \alpha Str}{n} - k\ Str \\ &= BS + \frac{j_0 - j_\alpha + \alpha Str}{n} - k\ Str. \end{aligned} \quad (19)$$

From Eq. 16, we have that:

$$j_\alpha = (j_0 + \alpha Str) \mod n. \quad (20)$$

This means that there exists $k_1 \in \mathbb{N}$ such that:

$$j_0 + \alpha Str = j_\alpha + k_1 n$$

$$\alpha = \frac{j_\alpha + k_1 n - j_0}{Str}. \quad (21)$$

This means that there exists a function ψ such that $\alpha = \psi(n, j_\alpha, Str) \in \mathbb{N}$. As a consequence, we have:

$$\begin{aligned} i_\alpha &= BS + \frac{j_0 - j_\alpha + \alpha Str}{n} - k\ Str \\ &= BS + \frac{j_0 - j_\alpha + Str\psi(n, j_\alpha, Str)}{n} - k\ Str \\ &= BS + \phi(n, j_\alpha, Str) = BS + A_\alpha. \end{aligned} \quad (22)$$

where $\phi : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$. □

Remark 4 By the previous Theorem 3 we can see that knowing the bank index j_α and the Stride Str, it is possible to identify the row index i_α of an element. More precisely, it is possible to calculate the local address (row index) in a **distributed** manner, with only the knowledge of the local bank index j_α, the address of the first element a_0 (or B in Fig. 1b) and the stride Str. We note that the structure of $\phi(n, j_\alpha, Str)$ is completely determined by the bank assignment and row assignment functions. From a hardware point of view, it can be either a SRAM-based lookup table or hardwired. In the next Section 4 we will show the hardwired A_α (i.e. the $\phi(n, j_\alpha, Str)$ functions). Additionally we can note that BS, called base-stride, represents the combination of the base i_0 and a certain number $k \in \mathbb{N}$ of times Str, which depends on the number of accesses necessary to collect the data. Clearly A_α depends on the memory bank and it is variable. For example, let us consider the example depicted in Fig. 3. In the example, the vector to be accessed has $2n$ elements. This means that 2 parallel accesses are required to collect the data. This means that k can assume the value 0 or 1. For instance, to calculate the local address (row index) of bank 2 ($j_\alpha = 2$) during the second memory access ($k = 1$), using Theorem 3, we have $i_\alpha = BS + A_\alpha = i_0 + 1 \cdot Str + A_\alpha = 1 + 3 + 2 = 6$. In this case, since $k = 1, \alpha \in [8, 15]$ and therefore $k_1 = 5$ univocally determined.

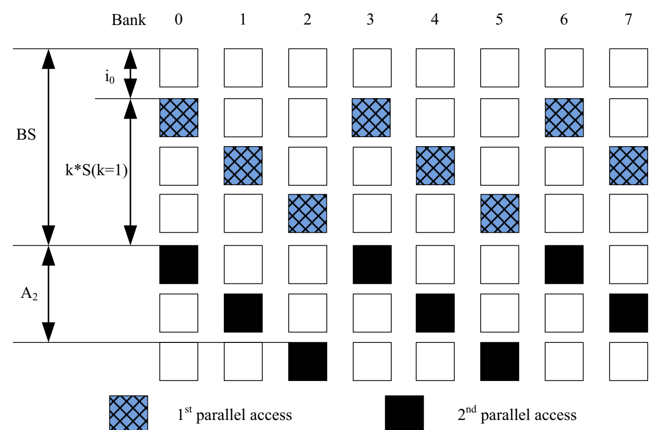

Figure 3 Distributed local address (row index) calculation.

4 The Address Generation Unit

In this section, we will describe the design of our distributed AGEN scheme.

In Fig. 4 a simple vector based SIMD organization is shown. The shaded part depicts the distributed AGEN unit proposed in this paper. There are i memory modules and processing elements connected by a shuffling switch. The base address, stride and index are used by the distributed address generation scheme to provide with the current addresses. Please note the difference between the simple 1-to-1 data permutation in space performed by the switch (between the memory modules and the processing elements) and the operation of the memory control and data unit functionality, e.g. reordering DRAM memory accesses, hiding bus turnaround penalties and more. In addition, when proper mapping of data onto the memory banks is

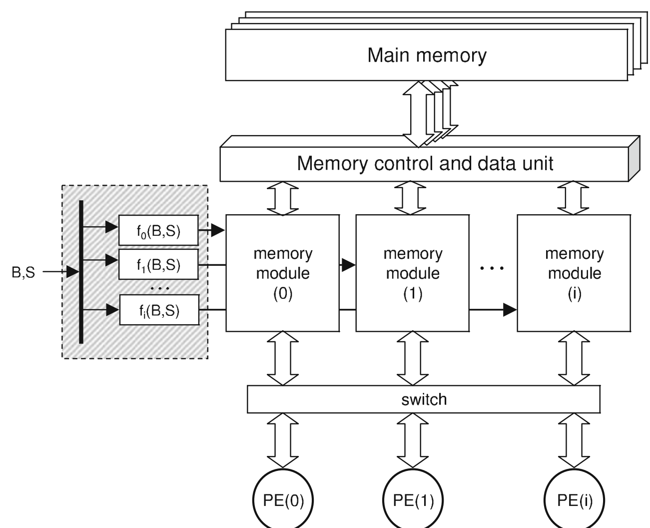

Figure 4 Vector processor based organization.

0	8	12	16	20	24	28–31
	Reg1	Reg2	Reg3	Reg4	Reg5	Reg6
Opcode	Base 1	Base2	Base3	Length	Stride	Index

Figure 5 The special-purpose-instruction [31].

applied, the switch can be avoided at all. This will additionally slightly increase the system performance. Organization like this can be used for proposals with strict demands on the timing of the address generation process as the ones presented in [25, 26].

Such organization can also be applied for polymorphic computing systems as the one proposed in [27]. The arbiter used to distribute the instructions between the general purpose processor and the reconfigurable vector co-processor will provide the required control instructions at the exact moment in the instruction flow. Such an organization is assumed in the reminder of this section.

We consider an 8-way parallel memory where the memory banks are built using dual ported memories, such as BRAMs [28], which in case of an implementation on an FPGA are shared by the core processor and the vector co-processor (composed by the PE's in the figure). The 8 memory banks are arranged as a linear array memory with *low-order interleaving* address mapping [29] from the general purpose processor and the memory controller side. The second BRAM port is used by the vector processing elements that see the memory organized as a 2D rectangular array [13]. Since the two memory accesses are decoupled in nature and the address generation is critical for the SIMD performance, we will concentrate on the vector co-processor side.

When the vector co-processor needs to access the memory banks in parallel, the Effective Address (EA) computation impacts performance [30]. Our AGEN unit (different from the one embedded into the main memory controller and the general purpose processor) which generates the addresses for the data stored in the parallel memory in the correct sequence plays an important role. Considering an 8-way parallel memory, 8 different addresses for each memory access are necessary. The proposed AGEN generates those 8 addresses needed to fetch the data simultaneously from the memory at high rate. More specifically, as we will see in Section 5, the proposed AGEN is able to generate eight 32-bit addresses every 6 ns for different odd strides when implemented in Xilinx FPGA technology.

Our AGEN unit is designed to work with single or multiple groups of streamed data (with the same stride) using a special-purpose-instruction like in [31]. This instruction configures the base addresses, the stride and the length of the particular streaming data format. Additionally, the memory accesses can be performed in parallel with the execution phase of a previous iteration using a decoupled approach as presented in [32] and as shown in Fig. 4.

Figure 5 depicts an example of the special-purpose-instruction for operations with multiple indices such as Sum of Absolute Differences operation (SAD) and/or Multiply-ACcumulate operation (MAC). More precisely, the special-purpose-instruction contains the following information:

- $Base_i$ ($i = 1, 2, 3$). These three fields contain the memory addresses of the data to read or write in the memory. For example, these fields can be used to store the minuend and subtrahend of the SAD operation or the multiplicand, multiplier and addendum in the MAC operation.
- *Length.* This field contains the number of n-tuples (cycles) needed to fetch n elements from the memory. For example, if $Length = 10$ and $n = 8$, in 10 memory cycles 80 elements will be fetched.
- *Stride.* This field contains the address distance between two consecutive data elements stored in an n-way parallel memory. In this paper we address only the odd strides between 1 and 15. Future research will include the analysis of the remaining cases.

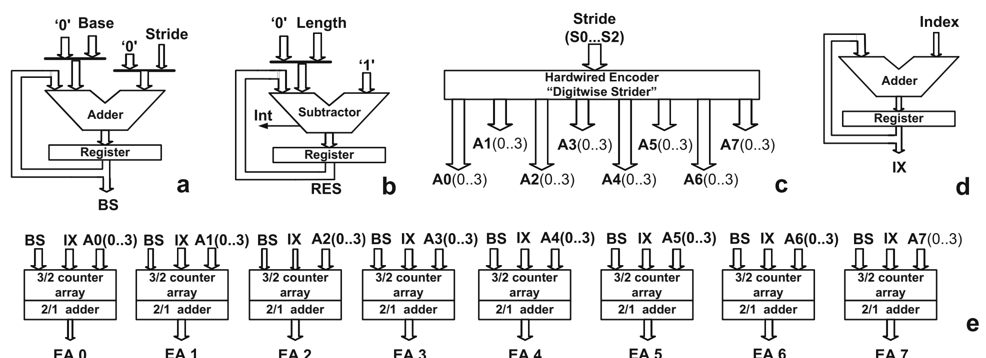

Figure 6 AGEN: **a** accumulator for BS computing, **b** accumulator for loop control, **c** hardwired encoder, **d** index accumulator, **e** final addition EA computing.

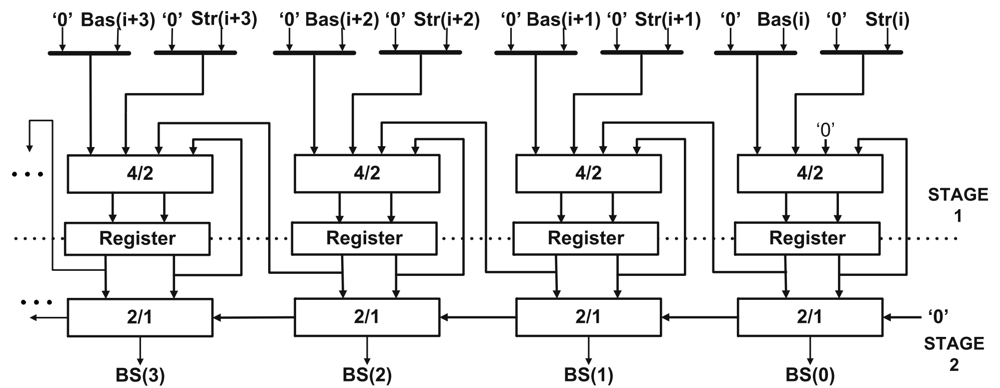

Figure 7 Main accumulator circuitry.

These 8 different strides are encoded using three bits, even though 4 bits are reserved for this field.

– *Index*. This field can contain two different information field:
  - The field contains the vertical distance between two consecutive groups of n elements. As an example, in Fig. 2a the index, or vertical stride, is equal to 9.
  - The field contains an offset address used by the AGEN to retrieve a single data word.

Equation 23 shows the EA (i.e. the row address of memory bank j on $k-th$ parallel memory access) computation:

$$EA_j = BS + A_j(0...3) + IX \quad \forall\ 0 \leq j \leq 7 \ \wedge \ RES \geq 0 \tag{23}$$

where BS is a base-stride combination explained in Theorem 3, IX is the index and $A_j(0...3)$ is the memory-bank offset, which is also exactly in accordance with in Theorem 3. Figure 6e depicts the eight EA generators for the targeted 8-way parallel memory system. BS is calculated as follows: $BS = i_0 + k \cdot Str$, where i_0 is the row index (i.e. local address) of the first element of the vector access (a_0). During the first cycle, BS is equal to i_0 ($k = 0$). After that, at iteration k, BS is augmented of $k \cdot Str$.

Figure 6a shows the accumulator structure. This structure is composed by two accumulator stages partially (4-bits only) shown in Fig. 7. The first stage consists of a 4/2 counter which receives the SUM and the carry signals of the previously computed value and the mux-es outputs used to select the appropriate operands (base and stride values) as previously explained. The second stage consists of a 2/1 adder which produces the BS values.

Figure 6b depicts the subtractor used for counting the number of memory accesses. At each clock cycle the subtractor value is decremented by one unit until it reaches zero. A negative value of the subtractor result (underflow) indicates the end of address generation process.

The stride values considered in our implementation are encoded using 3 bits and represented by $S_2S_1S_0$. The pattern range $000_2..111_2$ encodes the odd stride

Table 2 Hardwired encoder: set up table of equations for the address generation.

Bank	A_0	A_1	A_2	A_3
0	0	0	0	0
1	$S_2 \cdot \overline{S_1} \cdot \overline{S_0} + \overline{S_2} \cdot \overline{S_1} \cdot S_0 + S_2 \cdot S_1 \cdot S_0 + \overline{S_2} \cdot S_1 \cdot \overline{S_0}$	$S_2 \cdot \overline{S_1}$	$S_2 \cdot S_0 + S_1 \cdot S_2$	$S_2 \cdot S_1$
2	S_1	$\overline{S_2} \cdot \overline{S_1} \cdot S_0 + S_2 \cdot \overline{S_0} + S_2 \cdot S_1$	$\overline{S_2} \cdot S_1 \cdot S_0$	$S_2 \cdot S_0$
3	S_2	$S_2 \cdot \overline{S_0}$	$\overline{S_2} \cdot S_1$	$S_2 \cdot S_1$
4	S_0	S_1	S_2	0
5	S_2	$\overline{S_2} \cdot S_0$	$S_2 \cdot \overline{S_1} \cdot \overline{S_0} + S_2 \cdot S_1 \cdot S_0$	$S_2 \cdot \overline{S_1} \cdot S_0$
6	S_1	$S_2 \cdot S_1 + S_2 \cdot S_0 + \overline{S_2} \cdot S_1 \cdot \overline{S_0}$	$S_2 \cdot \overline{S_1} \cdot \overline{S_0}$	$S_2 S_1 \overline{S_0}$
7	$S_2 \cdot \overline{S_1} \cdot \overline{S_0} + S_2 \cdot S_1 \cdot S_0 + \overline{S_2} \cdot S_1 \cdot \overline{S_0}$	$S_2 \cdot \overline{S_1}$	$S_2 \cdot \overline{S_1} + S_2 \cdot S_0$	0

For example the address bit A_2 of bank 1 will be: $A_2 = S_2 \cdot S_0 + S_1 \cdot S_2$. This value (offset) is added to the current Base address value to calculate EA_2 (see Eq. 23).

Table 3 Time delay and hardware usage of the AGEN unit and the embedded arithmetic units.

Unit	Time delay (ns)			Hardware used	
	Logic delay	Wire delay	Total delay	Slices	LUTs
AGEN	4.5	1.4	6.0	673	1072
Hardwired encoder (Digitwise)[a]	0.3	–	0.3	9	16
4:2 counter[a]	0.5	0.5	1.0	72	126
3:2 counter[a]	0.3	–	0.3	37	64
32-bit CPA (2/1) adder[a]	2.2	0.7	2.9	54	99

[a]Embedded circuitry inside the AGEN unit presented without I/O buffers delays.

values ranging from 1 to 15. A hardwired encoder is used to transform the encoded stride values into the corresponding $A_0(0...3)$,.., $A_7(0...3)$ address offsets using a *memory-bank-wise operation*. A memory-bank-wise address is created based on the stride value. For example, Fig. 2b shows the case when the stride is 3. In this case, it is possible to see that for the memory banks 1 and 2 it is required an offset value of 1 and 2 respectively. These values are generated by the hardwired encoder presented in Fig. 6c (see Table 2).

5 Experimental Results

The proposed AGEN unit has been implemented in VHDL, synthesized and functionally validated using the ISE 7.1i Xilinx environment [33]. The target device used was a VIRTEX-II PRO xc2vp30-7ff1696 FPGA. Table 3 summarizes the performance results in terms of delay time and hardware utilization. The results are for the complete AGEN unit as well as for the major sub-units used in our design.

From the data presented in Table 3 it is possible to conclude that the 32-bit CPA adder used is the most expensive component in terms of delay. However, this delay can be additionally reduced by using a deeper pipeline organization, as shown in [34], with an increase in the overall performance. This becomes important when the target technology has a lower memory latency like the Virtex 4 and the Virtex 5 platforms [35]. The AGEN unit proposed in this paper has a 3-stage pipeline. Stage 1 and stage 2 are used by the accumulator for generating the BS values (see Fig. 6a). Stage number 3 is used for the 3/2 counter array and the final 2/1 adder which represents the critical path of our implementation.

The proposed AGEN reaches an operation frequency of 166 MHz which means that the proposed AGEN is capable of generating 1.33 Giga addresses of 32-bits, a total of 43.5 Gbps, to fetch the data from the 8-way parallel memory system. The proposed AGEN makes use of very few hardware resources: only 3% of the slices and only 4% of the LUTs of the target device (VIRTEX-II PRO xc2vp30-7ff1696 FPGA).

6 Conclusions

In this paper we have presented the design of an efficient AGEN unit for fetching several data from an *n*-way parallel memory system in a single machine cycle. The proposed AGEN can reach an operation frequency of 166 MHz, which means that the proposed AGEN is capable of generating 1.33 Giga addresses of 32-bits, a total of 43.5 Gbps, to fetch the data from the parallel memory system. The proposed AGEN makes use of a limited amount of hardware resources: only 3% of the total slices and only 4% of the total LUTs of the target platform, the VIRTEX-II PRO xc2vp30-7ff1696 FPGA.

The proposed AGEN unit reduces the number of cycles required for data transfers within the memory system, providing a high bandwidth utilization and a short critical path in hardware.

Acknowledgements This work was supported by the European Union in the context of the MORPHEUS project #027342, the SARC integrated project #27648 and the foundation for the advancement of the field of computer engineering, Delft, The Netherlands.

Open Access This article is distributed under the terms of the Creative Commons Attribution Noncommercial License which permits any noncommercial use, distribution, and reproduction in any medium, provided the original author(s) and source are credited.

References

1. Espasa, R., Valero, M., & Smith, J. E. (1998). Vector architectures: Past, present and future. In *ICS '98: Proceedings of the 12th international conference on supercomputing* (pp. 425–432). New York, NY: ACM.
2. Russell, R. M. (2000). The CRAY-1 computer system. In M. D. Hill, N. P. Jouppi, & G. S. Sohi (Eds.), *Readings in computer architecture* (pp. 40–49). San Francisco: Morgan Kaufmann.

3. Hammond, S. W., Loft, R. D., & Tannenbaum, P. D. (1996). Architectural and application: the performance of the NEC SX-4 on the NCAR benchmark suite. In *Supercomputing '96: Proceedings of the 1996 ACM/IEEE conference on supercomputing* (*CDROM*) (pp. 22). (ISSN:0-89791-854-1) Pittsburgh, Pennsylvania: IEEE Computer Society. doi:http://doi.acm.org/10.1145/369028.369076.
4. Advanced Micro Devices (2000). *3DNow technology manual*. Austin: Advanced Micro Devices.
5. Thakkar, S. T., & Huff, T. (1999). Internet streaming SIMD extensions. *Computer, 32*(12), 26–34.
6. Diefendorff, K., Dubey, P. K., Hochsprung, R., & Scales, H. (2000). Altivec extension to PowerPC accelerates media processing. *IEEE Micro, 20*(2), 85–95.
7. Kahle, J. A., Day, M. N., Hofstee, H. P., Johns, C. R., Maeurer, T. R., & Shippy, D. (2005). Introduction to the cell multiprocessor. *IBM Research Journal and Development, 49*(4/5).
8. Macedonia, M. (2003). The gpu enters computing's mainstream. *IEEE Computer Magazine, 36*(10), 106–108 (October).
9. Corbal, J., Espasa, R., & Valero, M. (1998). Command vector memory systems: high performance at low cost. In *PACT '98: Proceedings of the 1998 international conference on parallel architectures and compilation techniques* (p. 68). Washington, DC: IEEE Computer Society.
10. Mathew, B. K., McKee, S. A., Carter, J. B., & Davis, A. (2000). Design of a parallel vector access unit for SDRAM memory systems. In *Proceedings of the sixth international symposium on high-performance computer architecture* (*HPCA-6*) (pp. 39–48). citeseer.ist.psu.edu/mathew00design.html (January).
11. Mathew, B. K., McKee, S. A., Carter, J. B., & Davis, A. (2000). Algorithmic foundations for a parallel vector access memory system. In *Proc. 12th ACM symposium on parallel algorithms and architectures* (pp. 156–165). citeseer.ist.psu.edu/mathew00algorithmic.html (July).
12. Bailey, D. H. (1987). Vector computer memory bank contention. *IEEE Transactions on Computers, 36*(3), 293–298.
13. Kuzmanov, G., Gaydadjiev, G. N., & Vassiliadis, S. (2006). Multimedia rectangularly addressable memory. *IEEE Transactions on Multimedia, 8*(2), 315–322.
14. Kuck, D. J., & Stokes, R. A. (2000). The burroughs scientific processor (BSP). In M. D. Hill, N. P. Jouppi, & G. S. Sohi (Eds.), *Readings in computer architecture* (pp. 528–541). San Francisco: Morgan Kaufmann.
15. Lawrie, D. H., & Vora, C. R. (1982). The prime memory system for array access. *IEEE Transactions on Computers, 31*(5), 435–442.
16. Won Park, J. (2001). An efficient buffer memory system for subarray access. *IEEE Transactions on Parallel and Distributed Systems, 12*(3), 316–335.
17. Won Park, J. (2004). Multiaccess memory system for attached SIMD computer. *IEEE Transactions on Computers, 53*(4), 439–452.
18. Budnik, P., & Kuck, D. (1971). The organization and use of parallel memories. *IEEE Transactions on Computers, C-20*(12), 1566–1569 (December).
19. Harper III, D. (1991). Block, multistride vector, and FFT accesses in parallel memory systems. *IEEE Transactions on Parallel and Distributed Systems, 2*(1), 43–51.
20. Harper III, D. (1992). Increased memory performance during vector accesses through the use of linear address transformations. *IEEE Transactions on Computers, 41*(2), 227–230.
21. Harper III, D., & Linebarger, D. A. (1991). Conflict-free vector access using a dynamic storage scheme. *IEEE Transactions on Computers, 40*(3), 276–283.
22. Sohi, G. (1993). High-bandwidth interleaved memories for vector processors - a simulation study. *IEEE Transactions on Computers, 42*(1), 34–44.
23. Seznec, A., & Lenfant, J. (1992). Interleaved parallel schemes: improving memory throughput on supercomputers. In *ISCA '92: Proceedings of the 19th annual international symposium on computer architecture* (pp. 246–255). New York, NY: ACM.
24. Lang, T., Valero, M., Peiron, M., & Ayguadé, E. (1995). Conflict-free access for streams in multimodule memories. *IEEE Transactions on Computers, 44*(5), 634–646.
25. Calderón, H., & Vassiliadis, S. (2005). Reconfigurable multiple operation array. In *Proceedings of the embedded computer systems: Architectures, modeling, and simulation* (*SAMOS05*) (pp. 22–31) (July).
26. Calderón, H., & Vassiliadis, S. (2006). Reconfigurable fixed point dense and sparse matrix-vector multiply/add unit. In *ASAP '06: Proceedings of the IEEE 17th international conference on application-specific systems, architectures and processors* (*ASAP'06*) (pp. 311–316). Washington, DC: IEEE Computer Society.
27. Vassiliadis, S., Wong, S., Gaydadjiev, G., Bertels, K., Kuzmanov, G., & Panainte, E. M. (2004). The MOLEN polymorphic processor. *IEEE Transactions on Computers, 53*(11), 1363–1375.
28. Xilinx (2003). XILINX-LogiCore: Dual-Port block memory v7.0 - product specification. DS235 Xilinx (December).
29. Hwang, K., & Briggs, F. A. (1990). *Computer architecture and parallel processing*. New York, NY: McGraw-Hill.
30. Mathew, S., Anders, M., Krishnamurthy, R., & Borkar, S. (2002). A 4GHz 130nm address generation unit with 32-bit sparse-tree adder core. In *Proceedings of IEEE VLSI circuits symposium* (pp. 126–127). Honolulu (June).
31. Juurlink, B. H. H., Vassiliadis, S., Tcheressiz, D., & Wijshoff, H. A. G. (2001). Implementation and evaluation of the complex streamed instruction set. In *PACT '01: Proceedings of the 2001 international conference on parallel architectures and compilation techniques* (pp. 73–82). Washington, DC: IEEE Computer Society.
32. Espasa, R., & Valero, M. (1997). Exploiting instruction- and data-level parallelism. *IEEE Micro, 17*(5), 20–27.
33. Xilinx (2005). The XILINX software manuals. XILINX 7.1i. http://www.xilinx.com/support/swmanuals/xilinx7/.
34. Xilinx (2003). XILINX-LogiCore: Adder/Subtracter v7.0 - Product specification. DS214 Xilinx (December).
35. Memory Solutions (2007). http://www.xilinx.com/products/design resources/mem corner/.

Carlo Galuzzi received the M.Sc. in Mathematics (summa cum laude) from Università Degli Studi di Milano, Italy in 2003. He is currently at the final stage of his Ph.D. in Computer Engineering at TU Delft, The Netherlands. He is a reviewer for more than 20 international conferences and journals. He served as publication chair for many conferences, e.g. MICRO-41, SAMOS 2006-08, DTIS 2007. His research interests include instruction-set extension, hardware-software partitioning and graph theory. Carlo received the best paper award at ARC 2008.

Chunyang Gou was born in Sichuan, China in 1981. He received the Bachelor degree from University of Electronic Science and Technology of China (UESTC), Chengdu, China in 2003 and the MSc degree from Tsinghua University, Beijing, China in 2006. He is currently working towards the Ph.D. in Computer Engineering in the Delft University of Technology, The Netherlands. His research interests include computer architecture in general, with particular emphasis on high-performance memory hierarchies.

Humberto Calderón was born in La Paz, Bolivia, in 1964. He received the M.Sc. degree in Computer Sciences from the ITCR (Costa Rica) in 1997 and the Ph.D. degree in computer engineering from TU Delft, The Netherlands, in 2007. His current research interests include reconfigurable computing, multimedia embedded systems, computer arithmetic, intelligent control and robotics. He currently joined the “Istituto Italiano di Tecnologia in Genova, Italy, as a senior engineer and researcher.

Georgi N. Gaydadjiev was born in Plovdiv, Bulgaria, in 1964. He is currently assistant professor at the Computer Engineering Laboratory, Delft University of Technology, The Netherlands. His research and development industrial experience includes more than 15 years in hardware and software design at System Engineering Ltd. in Pravetz Bulgaria and Pijnenburg Microelectronics and Software B.V. in Vught, the Netherlands. His research interests include: embedded systems design, advanced computer architectures, hardware/software co-design, VLSI design, cryptographic systems and computer systems testing. Georgi has been a member of many conference program committees at different levels, e.g. ISC, ICS, Computing Frontiers, ICCD, HiPC and more. He was program chair of SAMOS in 2006 and was a general chair in 2007. Georgi received the best paper awards at Usenix/SAGE LISA 2006 and WiSTP 2007. He is IEEE and ACM member.

Stamatis Vassiliadis (M'86-SM'92-F'97) was born in Manolates, Samos, Greece 1951. Regrettably, Prof. Vassiliadis deceased in April 2007. He was a chair professor in the Electrical Engineering department of Delft University of Technology (TU Delft), The Netherlands. He had also served in the EE faculties of Cornell University, Ithaca, NY and the State University of New York (S.U.N.Y.), Binghamton, NY. He worked for a decade with IBM where he had been involved in a number of advanced research and development projects. For his work he received numerous awards including 24 publication awards, 15 invention awards and an outstanding innovation award for engineering/scientific hardware design. His 72 USA patents rank him as the top all time IBM inventor. Dr. Vassiliadis received an honorable mention Best Paper award at the ACM/IEEE MICRO25 in 1992 and Best Paper awards in the IEEE CAS (1998, 2001), IEEE ICCD (2001), PDCS (2002) and the best poster award in the IEEE NANO (2005). He is an IEEE and ACM fellow and a member of the Royal Dutch Academy of Science.