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 amount 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 slowdown 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 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 for implementing address calculation 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]:

\[
\text{Algorithm}_1 \quad \forall k := 1 \text{ to } n \text{ do } \forall i := 1 \text{ to } n \text{ do } \forall j := 1 \text{ to } n \text{ do } c_{ij}^{(k)} := c_{ij}^{(k-1)} \oplus (a_{ik} \otimes b_{kj})
\]

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

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 \((\oplus, \otimes) = (\lor, \land)\), Algorithm_1 relates to Boolean product of matrices.
- **Case_3.** If \( c_{ij}^{(0)} = 1 \) and \((\oplus, \otimes) = (\land, \lor)\), Algorithm_1 deals with two-dimensional topple comparison.
• Case 4. If \( c_{ij}^{(0)} = +\infty \) and \( \langle \ominus, \odot \rangle = (\min, +) \), 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, \( k=1,2,\ldots, n \), 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 \( c \odot (a \ominus b) \).

Memory module MM is composed of \( n \) single port memory blocks, \( M_0, M_1, \ldots, M_{n-1} \). 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 \( M_i \), \( i = 0, 1, \ldots, n-1 \). 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 \( M_i \), \( i = 0, 1, \ldots, n-1 \), can be accessed by a host via the HAT, and by PEs through \( SL_i \), \( i = 0, 1, \ldots, n-1 \), \( SL_A \) 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 al., 2008].
2.3 Constituents of selector logic

The selector logic, SLi, acts as an interface between the BLSA, memory modules and a host. There are three types of selector logic blocks, called HAT, SL_A and SL_AB.

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 SLi acts as an interface between the accelerator-bus protocol and PE-component protocol. The SLi is active during the following data transfers: a) Host memory to Mi memory module via the accelerator bus - typical for initial loading of Mi; b) Mi to host memory - result transfer at the end of the computation; c) Mi tocin input of PEi - typical for the start of each Pi machine cycle; d) PEi to Mi - at the end of PEi machine cycle.

The selector logic block SLi is composed of the following three entities: a) DATAi/O - selects data that are written/read to/from memory block Mi; b) ADDR - generates the Mi's address for read/write operation; c) CTRL - generates control signals R/W, CS and OE for memory blocks Mi.

The selector logic SL_AB is used as an interface between boundary PEs (PE0 and PEm-1) and memory modules MEMA, MEMB. The hardware structure and principles of operation of SL_A and SL_B are identical. The SL_AB 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 cin of PE0 (MEM_B to input cin of PEm-1) - at the beginning of each machine cycle. The SL_AB is composed of three entities: 1) DATA_M_AB - passes data that are written/read to/from memory module MEM_A (MEM_B); 2) ADDR_AB - selects the address for access to MEM_A (MEM_B) location; 3) CTRL_AB - generates control signals R/W, CS and OE for memory module MEM_A (MEM_B).

2.4 Passive and active computations

![Diagram](image)

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-

2.5 Address transformations during initial loading of memory modules

Initial loading of memory modules MM, MEM_A and 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 ∈ {a, b, c}) is mapped into the accelerator address. In order to perform address transformation, a host executes the following sequence of operations:

\[
\text{LOAD}\ R1, R0(\text{Mem}_h) \\
\text{STORE}\ R0(\text{Mem}_\text{Acc}), R1
\]

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 cin, ain and bin 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).

![Table](image)

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 ∈ {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:

\[
\text{ADR}_A = \left[\frac{i}{2}\right] \times (i+1) \text{mod}2 + \left[\frac{n}{2}\right] + \left[\frac{i}{2}\right] \times i \text{mod}2 + j \times n
\]

\[
\text{ADR}_B = \left[\frac{j}{2}\right] \times (j+1) \text{mod}2 + \left[\frac{n}{2}\right] + \left[\frac{j}{2}\right] \times j \text{mod}2 + i \times n
\]

\[(i, j) \rightarrow (i', j') = ((n \cdot i + j) \text{mod} n, \text{shuffle}(i))\]  

(1)

i, j = 0, 1, ..., n - 1
The address transformations performed by $T_{\text{adr}_A}$, $T_{\text{adr}_B}$, and $T_{\text{adr}_C}$, according to Equation (1), are given in Figs. 4, 5, and 6. As can be seen from Fig. 4 and 5 addresses transformations performed by $T_{\text{adr}_A}$ and $T_{\text{adr}_B}$ are implemented by cross-wiring, only, i.e. there is not hardware logic blocks.

Fig. 4. Address transformation performed by $T_{\text{adr}_A}$.

Fig. 5. Address transformation performed by $T_{\text{adr}_B}$.

Fig. 6. Address transformation performed by $T_{\text{adr}_C}$.

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 $AGU_Ci$ is accompanied to each processing element $PE_i$, $i = 0, 1, \ldots, n-1$. The $AGU_Ci$ generates addresses for accessing $Mi$. During each iteration, $AGU_Ci$ generates addresses for the following data sequence

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 $3n-2$ instruction cycles are used.

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

where $c$ points to the element of matrix $C$ fetched from memory block $M_i$, which drives input $c_{in}$ of $PE_i$, $i = 0, 1, \ldots, n-1$. Sequences denoted as $S1$ and $S3$ are part of the $P_{\text{com}}$ phase, while $S2$ corresponds to $A_{\text{com}}$ phase (see Fig. 2).
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, multiplexer 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

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

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 $T_U$.

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

**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

$$Q_{HAT_{ini}} = \frac{Q_{T_{Adr\_A}} + Q_{T_{Adr\_B}} + Q_{T_{Adr\_C}}}{3} \approx 2.73$$

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

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

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

$$Q_H = \frac{Q_{HAT_{ini}} + Q_{HAT_{end}}}{2} \approx 3$$

The quality factor of the AGUs during BLSA operation is

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

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

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

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

<table>
<thead>
<tr>
<th>FPGA family</th>
<th>Number of PEs</th>
<th>Gate count</th>
<th>PE</th>
<th>Overhead [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Gate count</td>
<td>8-bit</td>
<td>16-bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>5122</td>
<td>10250</td>
</tr>
<tr>
<td>Spartan 3E</td>
<td>8</td>
<td>410 379</td>
<td>389</td>
<td>201</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>503 517</td>
<td>527</td>
<td>284</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>725 608</td>
<td>737</td>
<td>388</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>884 702</td>
<td>923</td>
<td>465</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>1046 733</td>
<td>1073</td>
<td>551</td>
</tr>
<tr>
<td></td>
<td>256</td>
<td>1070 815</td>
<td>1259</td>
<td>634</td>
</tr>
<tr>
<td>Virtex</td>
<td>8</td>
<td>422 346</td>
<td>374</td>
<td>198</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>593 605</td>
<td>542</td>
<td>281</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>791 812</td>
<td>743</td>
<td>385</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>1060 1042</td>
<td>881</td>
<td>462</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>1379 1444</td>
<td>1352</td>
<td>542</td>
</tr>
<tr>
<td></td>
<td>256</td>
<td>1609 1758</td>
<td>1493</td>
<td>631</td>
</tr>
</tbody>
</table>

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 effective.

REFERENCES

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