ARCHITECTURE AND IMPLEMENTATION OF THE 2D MEMORY WITH MULTI-PATTERN PARALLEL ACCESSES

Technical report

Supervisors: *G. Gaydadjiev,*
G. Kuzmanov

Arseniy Vitkovskiy, PhD student
ARCES - Advanced Research Center for Electronic Systems,
University of Bologna,
Viale Pepoli 3/2, 40123,
Bologna, ITALY
avitkovski@arces.unibo.it

December, 2007

Contents

ABSTRACT 3

1 INTRODUCTION 4

1.1 Research context and goal description 4

1.2 Related work 5

2 THEORETICAL BASIS 6

3 PROPOSED MEMORY ACCESS SCHEME 11

3.1 Module assignment function 11

3.2 Row address function 14

3.3 Memory access latencies 14

4 DESIGN IMPLEMENTATION AND COMPLEXITY EVALUATION 15

4.1 Mode select 16

4.2 Address generator 16

4.3 Row address generator 18

4.4 Module assignment unit 18

4.5 Shuffle unit 19

4.6 De-Shuffle unit 19

5 RESULTS AND ANALYSIS 20

5.1 ASIC synthesis 20

5.2 FPGA synthesis 21

5.3 Comparison with the related work 21

6 CONCLUSION 23

APPENDIX A. VHDL SOURCES. 24

REFERENCES 32

Abstract

This report presents a novel multi-pattern parallel addressing scheme in two-dimensional (2D) addressing space and the corresponding 2D interleaved memory organization with run-time reconfiguration features. The proposed architecture targets mainly multimedia and scientific applications with block cyclic data organization running on computing systems with high memory bandwidth demands, such as vector processors, multimedia accelerators, etc. The prior research on 2D addressing schemes is substantially extended introducing additional parameters, which define a large variety of 2D data patterns. The proposed scheme guarantees minimum memory latency and efficient bandwidth utilization for arbitrary configuration parameters of the data pattern. The presented mathematical descriptions prove the correctness of the proposed addressing schemes. The design and wire complexities, as well as the critical paths are evaluated using technology independent methodology and confirm the scalability of the memory organization. These theoretical results are confirmed by the synthesis for both ASIC and FPGA technologies. Comparison with the related works shows the advantages of reported addressing scheme. The RTL implementation of the memory organization represents the complete platform-independent IP and can be integrated in any architecture.

1 Introduction

1.1 Research context and goal description

With modern increase of technology development, the performance of memory subsystems lags more and more behind the processing units. This trend becomes increasingly evident for architectures with massively parallel data processing, such as multimedia accelerators, vector processors, SIMD-based machines, etc. There are several techniques developed to reduce the processor versus memory performance gap, including various caching mechanisms, memories advanced with extra wide data word or multiple ports. But most of all, the parallelism phenomenon is utilized in *parallel memory* organizations, where the storage subsystem consists of a set of memory modules working in parallel. The main advantages of this organization are: relatively small overheads, low latency, efficient interconnection usage and possibility of accessing specific *data patterns*. The data patterns depend very much on the target application and might have various shapes, sizes and *strides* (distances between the successive elements).

The design challenge is to ensure *conflict-free* parallel data access to all (or maximum possible number of) memory modules for a set of different data patterns. This is obtained by means of a *module assignment function*. According to the data pattern format (in other words template), various module assignments can be implemented, such as linear functions [4], XOR-schemes [6], rectangular addressable memories [11], periodic schemes [15] and others. *Row address function* specifies physical address inside a memory module. Together, module assignment and row address functions form the class of *skewing schemes*.

However, there is no single skewing scheme which would support conflict-free access for all possible data patterns [1]. Two solutions that deal with such limitation are: *Configurable Parallel Memory Architecture* (CPMA) [10], [13] and *Dynamic Storage Scheme* (DSS) [1], [9], [8], [16]. CPMA provides access to a number of data templates using a single relatively complex hardware when the number of memory modules is arbitrary. A more dedicated DSS unifies multiple storage schemes within one system. The appropriate scheme is chosen dynamically according to the specific data pattern in use. DSS restricts the amount of memory modules to the power of two and considers only interleaved memory system [9], [8].

The goal of this research is to develop a memory hierarchy with dynamically adjustable regular 2D access patterns, which would improve the data throughput between the main memory and processing units. Our approach is to split the problem into six trivial sub-problems, which would require hardware implementation with rather low complexity and short critical path. We consider an exhaustive set of pattern definition parameters and propose a performance efficient, interleaved memory organization. More specifically, the main contributions of the current proposal are as follows:

- Extended set of 2D pattern access parameters: *base address; vertical and horizontal strides, group lengths, and block sizes*.
- Support for the complete set of the 2D data patterns described by the above parameters.
- Run-time programmability of the memory access pattern by means of Special-Purpose Registers (SPRs).
- Independency of the data pattern size from the number of the interleaved memory modules;
- Minimal memory access latency for arbitrary strides and group lengths.

- Modular implementation, which can be easily simplified to a restricted subset of 2D data patterns (if the target application does not require full flexibility), thus reducing the design complexity and critical path.
- High design scalability confirmed by hardware synthesis results.

The proposed memory organization targets highly data-parallel applications with 2D block cyclic data distribution [5]. The matter concerns mainly scientific operations on matrices for e.g. synthetic aperture radar (SAR) software [12] or applied aerodynamics (Actiflow), as well as multimedia applications such as audio/video compression (ADPCM, G721, GSM, MPEG4, JPEG).

1.2 Related work

A DSS for a strided vector access was presented in [9]. The stride value is detected by the compiler and sent directly to the pipelined address transformation hardware. In such a way, the scheme supports conflict-free accesses for non-restricted vectors with constant arbitrary stride.

In [8] the authors extended their scheme with block, multistride and FFT accesses. To decrease the latency of multistride vector access when conflict-free access is not achieved, it was proposed to use dedicated buffers to smooth out transient non-uniformities in module reference distribution. Block access supported only restricted set of blocks sizes equal to power of two. Finally, in order to improve a radix-2 FFT algorithm, authors proposed non-interleaved storage scheme and a constant geometry algorithm for which they identified three data patterns. For all three types of address transformations, the same hardware was used.

CPMA from [10] supports generate, crumbled rectangle, chessboard, vector, and free data patterns. Virtual address is used to read appropriate row address and access function from the page table which are further transformed into row and module addresses. The authors presented complexity, timing and area evaluations for their architecture.

A buffer memory system for a fast and high-resolution graphical display system was proposed in [13]. It provides parallel access to a block, horizontal, vertical, forward-diagonal, and backward-diagonal data pattern in a two-dimensional image array. All pattern sizes are limited to power of two. The address differences of those patterns are specifically prearranged and saved in two SRAMs so that later they can be added to the base address in order to obtain memory module addresses.

Other researches explore memory scheduling of DRAM chips by addressing locality characteristics within the 3D (bank, row, column) memory structure [14]. The solution consists of reordering memory operations in such a way that allows saving clock cycles on precharging banks and accessing successive rows and columns.

2 Theoretical basis

In parallel memories data can be referenced using predetermined patterns called memory access patterns or data patterns. The data distribution among the parallel memory modules is called a module assignment function m which is also known as a skewed scheme. The module assignment function determines data patterns that can be accessed conflict-free. A data element with a linear address a is assigned to a memory module according to $m(a)$. A row address function A determines the physical address of a data element inside a memory module.

Our task is to develop a memory hierarchy with dynamically programmable data patterns to minimize the main memory access latency from the vector processing units using interleaved memory modules organized in a two-dimensional matrix with parallel access. Fig. 1 depicts an example of a data pattern of six groups of size $VGL \times HGL = 2 \times 4$ and strides $(VS,HS)=(4,5)$. The parameters used to describe the data pattern are explained in Table 1.

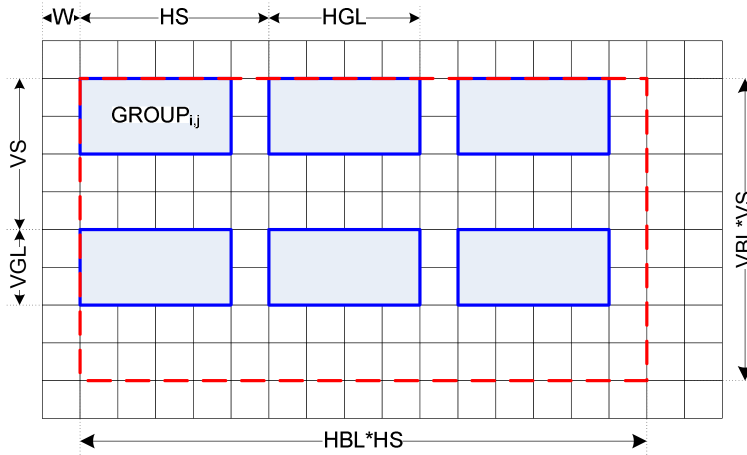

Fig. 1. Proposed memory access pattern.

Table 1. Memory access pattern parameters.

Parameter	*Description*
$W \in \mathbb{N}$[1]	- data word length in bytes;
$w_A \in \mathbb{N}$	- row address width in bits;
$b'(vb,hs) \in [0,\mathbb{N}]$	- base address of the accessed block;
$VS, HS \in \mathbb{N}$	- vertical and horizontal strides;
$VGL, HGL \in \mathbb{N}$	- vertical and horizontal group lengths – group size;
$VBL, HBL \in \mathbb{N}$	- vertical and horizontal block lengths – block size;
$M \times N \in \mathbb{N}$	- size of a data block stored in the memory;
$VD \times HD \in \mathbb{N}$	- size of the matrix of memory modules;

[1] Though the word length can have any natural value, on practice it usually has a value of power of two.

$0 \le i = \left\lfloor \frac{id_V}{VGL} \right\rfloor < VBL$, $0 \le id_V < VBL \cdot VGL$, $id_V \in \mathbb{N}$. $0 \le j = \left\lfloor \frac{id_H}{HGL} \right\rfloor < HBL$, $0 \le id_H < HBL \cdot HGL$, $id_H \in \mathbb{N}$.	- vertical i and horizontal j group indices;
$0 \le k = id_V \bmod VGL < VGL$ $0 \le l = id_H \bmod HGL < HGL$	- vertical k and horizontal l element indices.

The first step is to translate the linear address of the main memory into a two-dimensional one. The transformation equation (1) is used in order to translate linear base address b' into a two-dimensional one (vb, hb) with the vertical and horizontal constituents.

$$b' = vb \cdot N + hb, \quad vb = \lfloor b'/N \rfloor, \quad hb = b' \bmod N \tag{1}$$

Any data element with linear address a' belonging to an accessed data block has the following two-dimensional address (va, ha):

$$a' = va \cdot N + ha, \quad va = VB + i \cdot VS + k = a_{i,k}, \quad ha = HB + j \cdot HS + l = a_{j,l}. \tag{2}$$

where indices $i \in [0, VBL-1], k \in [0, VGL-1]$, and $j \in [0, HBL-1], l \in [0, HGL-1]$

As follows from (2), the two-dimensional address is completely separable, i.e. its vertical and horizontal constituents are independent from each other. Therefore, in the following discussion we consider the address constituent along only one dimension.

The *stride* parameter as any other natural number might be represented in the following expansion:

$$S = \sigma \cdot 2^s \in \mathbb{N}, \tag{3}$$

where $\sigma = 2x + 1$, $\forall x \in \mathbb{N}$ and $s \in [0, \mathbb{N}]$. Consequently, stride S is *odd* when $s = 0$, and it is *even* when $s \in \mathbb{N}$.

Before proceeding to the description of our solution we would need to consider the following theorems.

Theorem 1: No single skewing scheme can be found that allows conflict-free access for all the constant strides and group lengths when the data pattern can be unrestrictedly placed. The theorem is valid for arbitrary number of memory modules, when at least two data elements are accessed concurrently[1].

[1] This theorem follows the theorem 1 from [1] but in our case it has wider application since it considers the group length together with the stride.

Proof: Let $m(a)$ be the module assignment function and $a_{id} = b + j \cdot S + l$ the first accessed data element. Then, the next accessed data element is

$$\begin{cases} a_{id+1} = b + j \cdot S + l + 1 = a_{id} + 1, & \text{if } l < GL - 1; \\ a_{id+1} = b + (j+1) \cdot S = a_{id} + S - l, & \text{if } l = GL - 1 \end{cases},$$

where GL is a group length, and S is a stride. The two elements can not be accessed conflict-free if there are stride S and group length GL such that $m(a_{id}) = m(a_{id+1})$. □

Theorem 2: All the *odd strides* can be accessed conflict-free with the low-order interleaved scheme if the number of memory modules equal to power of two: $D = 2^d$, $d \in \mathbb{N}$.

Proof: The proof is presented by M. Valero et al. in [16]. □

Theorem 3: Let stride S', group length GL, and number of memory modules D along one dimension equal to power of two, i.e. $S' = 2^s$, $GL = 2^{gl}$, and $D = 2^d$, where $s, gl, d \in \mathbb{N}$. Also let $s \geq d$ which means that the strides have less than one access per row. Then the module assignment function defined by

$$m(a) = \left(a + GL \cdot \left\lfloor \frac{\lfloor a/D \rfloor}{2^{s-d}} \right\rfloor \right) \bmod D, \tag{4}$$

allows conflict-free parallel accesses to D memory modules.

Proof: First we find a period P of function (4). If function $m(a)$ maps address a to its module address for a stride S' and group length GL, then $m(a) = m(a + P \cdot S')$, $\forall\, a$. According to (4), this corresponds to

$$\left(a + GL \cdot \left\lfloor \frac{\lfloor a/D \rfloor}{2^{s-d}} \right\rfloor \right) \bmod D = \left((a + P \cdot S') + GL \cdot \left\lfloor \frac{\lfloor (a + P \cdot S')/D \rfloor}{2^{s-d}} \right\rfloor \right) \bmod D;$$

$$\left(a + GL \cdot \left\lfloor \frac{\lfloor a/D \rfloor}{2^{s-d}} \right\rfloor \right) \bmod D = \left(a + P \cdot S' + GL \cdot \left\lfloor \frac{\lfloor a/D + P \cdot S'/D \rfloor}{2^{s-d}} \right\rfloor \right) \bmod D;$$

$$\left(a + GL \cdot \left\lfloor \frac{\lfloor a/D \rfloor}{2^{s-d}} \right\rfloor \right) \bmod D = \left(a + P \cdot S' + GL \cdot \left\lfloor \frac{\lfloor a/D \rfloor + P \cdot 2^{s-d}}{2^{s-d}} \right\rfloor \right) \bmod D;$$

$$\left(a + GL \cdot \left\lfloor \frac{\lfloor a/D \rfloor}{2^{s-d}} \right\rfloor \right) \bmod D = \left(a + P \cdot S' + GL \cdot \left\lfloor \frac{\lfloor a/D \rfloor}{2^{s-d}} \right\rfloor + P \cdot GL \right) \bmod D;$$

$$\left(a + GL \cdot \left\lfloor \frac{\lfloor a/D \rfloor}{2^{s-d}} \right\rfloor \right) \bmod D = \left(a + GL \cdot \left\lfloor \frac{\lfloor a/D \rfloor}{2^{s-d}} \right\rfloor + P \cdot (S' + GL) \right) \bmod D.$$

According to the properties of modulo operation, this equality holds when $(P \cdot (S' + GL)) \bmod D = 0$. The last formula corresponds to the following set of periods:

$$\left[\begin{array}{l} P \cdot (S' + GL) = x \cdot D; \\ P \cdot S' = y \cdot D; \\ P \cdot GL = z \cdot D, \end{array} \right. \Rightarrow \left[\begin{array}{l} P = x \cdot \dfrac{D}{S + GL}; \\ P = y \cdot \dfrac{D}{S'}; \\ P = z \cdot \dfrac{D}{GL}, \end{array} \right.$$

where $x, y, z \in \mathbb{N}$ and $x, y, z : P \in \mathbb{N}$.

Now we will find the minimum period $P_{\min} > 1$. This means to solve the next set of minimization problems:

$$\begin{cases} P_{\min} = \min\left(x \cdot \dfrac{D}{S' + GL} \right); \\ P_{\min} = \min\left(y \cdot \dfrac{D}{S'} \right); \qquad \text{where } P_{\min} \in \mathbb{N}. \\ P_{\min} = \min\left(z \cdot \dfrac{D}{GL} \right), \end{cases}$$

$$\begin{cases} x = \dfrac{S' + GL}{GCD(D, S' + GL)} = \dfrac{S' + GL}{GCD(2^d, 2^s + GL)} \overset{\substack{GCD \\ property}}{\underset{s \geq d}{=}} \dfrac{S' + GL}{GCD(D, GL)}; \\ y = \dfrac{S'}{GCD(D, S')} = \dfrac{2^s}{GCD(2^d, 2^s)} \overset{s \geq d}{=} \dfrac{2^s}{2^d} = \dfrac{S'}{D}; \\ z = \dfrac{GL}{GCD(D, GL)}. \end{cases}$$

$$\begin{cases} P_{\min} = \dfrac{S' + GL}{GCD(D, GL)} \cdot \dfrac{D}{S' + GL} = \dfrac{D}{GCD(D, GL)}; \\ P_{\min} = \dfrac{S'}{D} \cdot \dfrac{D}{S'} = 1; \\ P_{\min} = \dfrac{GL}{GCD(D, GL)} \cdot \dfrac{D}{GL} = \dfrac{D}{GCD(D, GL)}. \end{cases}$$

Thus, the minimum period equal to $P_{\min} = \dfrac{D}{GCD(D, GL)}$.

Now we will find number of accesses performed to the distinct memory modules. Harper and Jump showed in [7] that for a basic skewing storage scheme the number of distinct modules referenced during a vector access is $A' = \min(P, D)$. For our case with group length presented, the number of distinct modules is $A = \min(P \cdot GL, D)$ and $GL = 2^{gl}$, $gl \in \mathbb{N}$.

$$A = \min(P \cdot GL, D) = \min\left(\frac{D \cdot GL}{GCD(D, GL)}, D \right) = \min\big(LCM(D, GL), D\big) = D.$$

Thus, any vector of length D, having the form of Fig. 1 and (2), inside the sequence of module addresses generated by $m(a)$ has exactly D distinct addresses. This is the definition of conflict-free accesses. □

Theorem 4: Let stride S', group length GL, and number of memory modules D along one dimension equal to power of two, i.e. $S' = 2^s$, $GL = 2^{gl}$, and $D = 2^d$, where $s, gl, d \in \mathbb{N}$. Also let $s < d$ which means that the strides have at least one access per row. Then the module assignment function defined by

$$m(a) = (a + \left(GL \cdot \left\lfloor \frac{a}{D} \right\rfloor \right) \bmod S') \bmod D, \qquad \textbf{(5)}$$

allows conflict-free parallel accesses to D memory modules.

Proof: The proof mainly repeats the one of *Theorem 3*. First we find a period P basing on the fact that $m(a) = m(a + P \cdot S')$, $\forall\, a$.

$$(a + \left(GL \cdot \left\lfloor \frac{a}{D} \right\rfloor \right) \bmod S') \bmod D = (a + P \cdot S + \left(GL \cdot \left\lfloor \frac{a + P \cdot S'}{D} \right\rfloor \right) \bmod S') \bmod D .$$

Using properties of modulo operation we derive the following combined equations:

$$\begin{cases} (P \cdot S') \bmod D = 0; \\ (a + \left(GL \cdot \left\lfloor \frac{a}{D} \right\rfloor \right) \bmod S') \bmod D = (a + \left(GL \cdot \left\lfloor \frac{a}{D} + \frac{P \cdot S'}{D} \right\rfloor \right) \bmod S') \bmod D. \end{cases}$$

The first equation gives us the set of periods equal to $P = x \cdot \frac{D}{S'}$, where $x \in \mathbb{N}$: $P \in \mathbb{N}$. Now we substitute period P in the second equation with its value derived from the first equation:

$$(a + \left(GL \cdot \left\lfloor \frac{a}{D} \right\rfloor \right) \bmod S') \bmod D = (a + \left(GL \cdot \left\lfloor \frac{a}{D} + x \cdot \frac{D}{S'} \cdot \frac{S'}{D} \right\rfloor \right) \bmod S') \bmod D ;$$

$$(a + \left(GL \cdot \left\lfloor \frac{a}{D} \right\rfloor \right) \bmod S') \bmod D = (a + \left(GL \cdot \left\lfloor \frac{a}{D} \right\rfloor + GL \cdot x \right) \bmod S') \bmod D .$$

Using the same property of modulo operator we obtain the following equation: $(x \cdot GL) \bmod S' = 0 \;\Rightarrow\; x = y \cdot \frac{S'}{GL}$, where $y \in \mathbb{N}$: $x \in \mathbb{N}$.

Substitute x in the equation which gives the set of periods P:

$$P = x \cdot \frac{D}{S'} = y \cdot \frac{S'}{GL} \cdot \frac{D}{S'} = y \cdot \frac{D}{GL},\; y \in \mathbb{N}:\; P \in \mathbb{N}.$$

This set of periods exactly repeats the one from the proof of *Theorem 3*. Therefore, the minimum period equal to $P_{\min} = \frac{D}{GCD(D, GL)}$ and, as follows from the same proof, the number of distinct accessed addresses equals to D. Thus, module assignment function (5) is conflict-free. □

Theorem 5: If a vector is to be accessed with even stride $S = \sigma \cdot 2^s$, where $\sigma = 2x + 1$, $\forall x \in \mathbb{N}$, $s \in \mathbb{N}$ and $s \neq 0$, and its elements are arranged in memory according to the storage scheme appropriate for a stride $S' = 2^s$ access the accesses are conflict-free [9].

Proof: The proof repeats the one from [9] with only difference that in our case we should examine the sequence of groups of module addresses instead of the sequence of single addresses. □

Now we are ready to present the proposed solution.

3 Proposed memory access scheme

Since there is no any single scheme for all the strides and group lengths according to *Theorem 1*, we propose to partition the problem in a number of cases, thus reducing the problem to a set of trivial sub-problems, and examine each of them independently.

We propose to partition the problem according to the *stride oddness* criterion on two subtasks. The following partition is done according to the theorems in section 2. According to *Theorem 2* odd strides can be accessed conflict-free using a basic skewing scheme [1], [11]. Further splitting on cases I and II is made for the purpose of memory latency minimization. Cases V and VI refer to *Theorem 3* and *Theorem 4* respectively. The remaining situations with even stride refer to cases III and IV and the skewing scheme from [9] is used. The problem partitioning is depicted on Fig. 2.

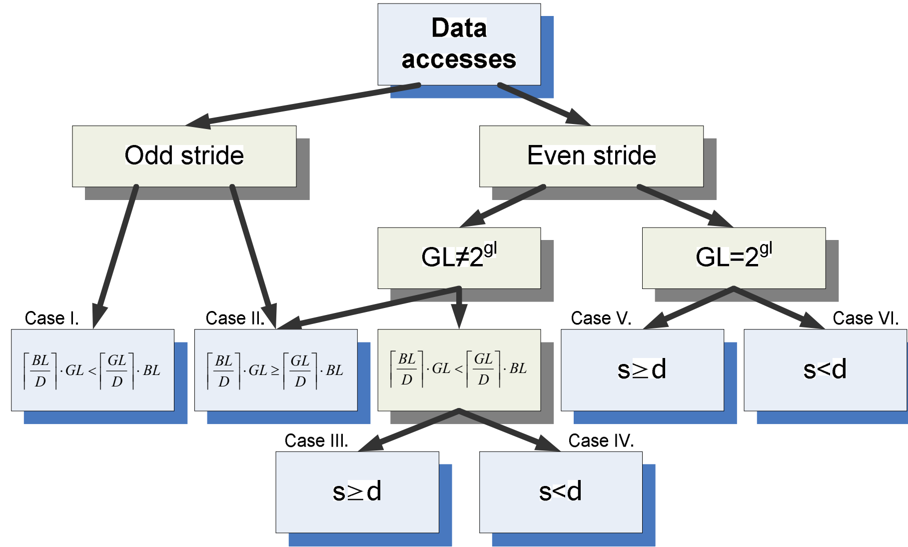

Fig. 2. Problem partition.

Now we need to build the module assignment and row address functions for all the cases.

3.1 Module assignment function

We partition the design problem, imposed by the multiplicity of access patterns, into trivial subtasks. A module assignment function is devised for each of six different cases with respect to particular initial conditions.

3.1.1 Case I

Initial conditions:

$$S = \{\sigma \cdot 2^s \mid \sigma = 2x+1; s = 0\},$$

$$\left\lceil \frac{BL}{D} \right\rceil \cdot GL < \left\lceil \frac{GL}{D} \right\rceil \cdot BL. \tag{6}$$

An access to the data pattern is performed on a basis of sets of elements; i.e. ab init, the set of the first elements of all groups is accessed followed by the set of the second elements of all groups and so on. When $BL > D$ then more than one access is required to reed/write a whole group of data. If $(BL \bmod D) \neq 0$ then the remaining memory modules stay unused. The relation $\left\lceil \frac{BL}{D} \right\rceil \cdot GL < \left\lceil \frac{GL}{D} \right\rceil \cdot BL$ again guarantees that the number of accesses required to access the

whole pattern is minimal. This case represents a conventional interleaved scheme with stride access which can be implemented conflict-free according to *Theorem 2*.

The module assignment function has the same representation as for a common interleaved scheme:

$$m(a) = a \bmod D\,. \tag{7}$$

The indices iterate according to the following sequence:

$$\begin{aligned}(i,k) = \; & ((0,0);(1,0);\ldots;(VBL\text{-}1,0);\\ & (0,1);(1,1);\ldots;(VBL\text{-}1,1);\\ & \ldots;\\ & (0,VGL\text{-}1);(1,VGL\text{-}1);\ldots;(VBL\text{-}1,VGL\text{-}1)).\end{aligned} \tag{8}$$

The number of the required accesses to read/write the whole data pattern in this case is equal to:

$$t = \left\lceil \frac{BL}{D} \right\rceil \cdot GL\,. \tag{9}$$

3.1.2 Case II

Initial conditions:

$$\begin{gathered} S = \{\sigma \cdot 2^s \mid \sigma = 2x+1; s = 0\}\,;\\ S = \{\sigma \cdot 2^s \mid \sigma = 2x+1; s \neq 0\} \;\&\; GL \neq 2^{gl}\,,\; gl \in \mathbb{N};\\ \left\lceil \frac{BL}{D} \right\rceil \cdot GL \geq \left\lceil \frac{GL}{D} \right\rceil \cdot BL\,. \end{gathered} \tag{10}$$

An access to the data pattern is performed group-wise, i.e. one group is accessed at a time. When $GL > D$ then more than one access is required to reed/write a whole group of data. If $(GL \bmod D) \neq 0$ then the remaining memory modules stay unused. The relation $\left\lceil \frac{BL}{D} \right\rceil \cdot GL \geq \left\lceil \frac{GL}{D} \right\rceil \cdot BL$ guarantees that the number of accesses required to access the whole pattern is minimal. The fact that any separate group inside the block can be accessed conflict-free is shown by G. Kuzmanov et al in [11].

The module assignment function has the same representation as for the case I (7):

$$m(a) = a \bmod D\,,$$

but the indices iterate according to the different sequence:

$$\begin{aligned}(i,k) = \; & ((0,0);(0,1);\ldots;(0,VGL\text{-}1);\\ & (1,0);(1,1);\ldots;(1,VGL\text{-}1);\\ & \ldots;\\ & (VBL\text{-}1,0);(VBL\text{-}1,1);\ldots;(VBL\text{-}1,VGL\text{-}1)).\end{aligned} \tag{11}$$

The number of the required accesses to read/write the whole data pattern in this case is equal to:

$$t = \left\lceil \frac{GL}{D} \right\rceil \cdot BL\,. \tag{12}$$

3.1.3 Case III

Initial conditions:

$$S = \{\sigma \cdot 2^s \mid \sigma = 2x+1; s \neq 0\} \;\&\; GL \neq 2^{gl}\,, \tag{13}$$

$$\left\lceil \frac{BL}{D} \right\rceil \cdot GL < \left\lceil \frac{GL}{D} \right\rceil \cdot BL \,,$$

$$s \geq d \,.$$

In this case, the indices iterate as in (8) but here we use module assignment function from [9]:

$$m(a) = \left(a + \left\lfloor \frac{\lfloor a/D \rfloor}{2^{s-d}} \right\rfloor \right) \bmod D \,, \tag{14}$$

In fact, all initial conditions for this case exactly repeat the ones presented in [9].

The number of the required accesses to read/write the whole data pattern is described by formula (12).

3.1.4 Case IV

Initial conditions:

$$S = \{\sigma \cdot 2^s \mid \sigma = 2x+1; s \neq 0\} \,\&\, GL \neq 2^{gl} \,,\ gl \in \mathbb{N};$$

$$\left\lceil \frac{BL}{D} \right\rceil \cdot GL < \left\lceil \frac{GL}{D} \right\rceil \cdot BL \,; \tag{15}$$

$$s < d \,.$$

Again, memory access repeats the sequence (8), and the module assignment function from [9] is appropriate to the initial conditions:

$$m(a) = \left(a + \left\lfloor \frac{a}{D} \right\rfloor \bmod S' \right) \bmod D \,. \tag{16}$$

The number of the required accesses to read/write the whole data pattern is described by formula (12).

3.1.5 Case V

Initial conditions:

$$S = \{\sigma \cdot 2^s \mid \sigma = 2x+1; s \neq 0\} \,\&\, GL = 2^{gl} \,,\ gl \in \mathbb{N};$$

$$s \geq d \,. \tag{17}$$

An access to the data pattern is performed element-wise, that allows the maximum utilization of the memory modules. Power of stride is not smaller than power of the array size: $s \geq d$.

The module assignment function has the following representation:

$$m(a) = \left(a + GL \cdot \left\lfloor \frac{\lfloor a/D \rfloor}{2^{s-d}} \right\rfloor \right) \bmod D \,, \tag{18}$$

The sequence of indices *(i, k)* is not important in this case since all memory modules are accessed conflict-free (refer to *Theorem 3*).

The number of the required accesses to read/write the whole data pattern in this case is equal to:

$$t = \left\lceil \frac{GL \cdot BL}{D} \right\rceil . \tag{19}$$

3.1.6 Case VI.

Initial conditions:

$$S = \{\sigma \cdot 2^s \mid \sigma = 2x+1; s \neq 0\} \;\&\; GL = 2^{gl},\ gl \in \mathbb{N};$$
$$s < d\,. \tag{20}$$

An access to the data pattern is performed element-wise, as in case V. Power of stride is smaller than power of the array size: $s < d$.

The module assignment function has the following representation:

$$m(a) = \left(a + \left(GL \cdot \left\lfloor \frac{a}{D} \right\rfloor\right) \bmod S'\right) \bmod D\,, \tag{21}$$

The sequence of indices *(i, k)* is not important again since all memory modules are accessed conflict-free (refer to *Theorem 4*).

Number of the required accesses to read/write the whole data pattern in this case, as in case V, is equal to:

$$t = \left\lceil \frac{GL \cdot BL}{D} \right\rceil. \tag{22}$$

3.2 Row address function

The row address function determines the linear address inside a memory module. An important characteristic of the proposed solution is that, in spite of having four different representations of the module assignment function, there is only one row address function which is valid for all cases described above. This feature enables large hardware design simplification as well as reduces the design time. The row address function is described by the following formula:

$$A(va, ha) = \left\lfloor \frac{va}{VD} \right\rfloor \cdot \left(\frac{N}{HD}\right) + \left\lfloor \frac{ha}{HD} \right\rfloor. \tag{23}$$

Equation (23) shows that the function is completely separable, which means that we are still able to examine vertical and horizontal constituents independently.

3.3 Memory access latencies

Number of accesses that are needed to read/write the whole data pattern is described by the following equations, representing the best and the worst cases:

$$t_m^{best} = \left\lceil \frac{VGL \cdot VBL}{VD} \right\rceil \times \left\lceil \frac{HGL \cdot HBL}{HD} \right\rceil, \tag{24}$$

$$t_m^{worst} = \left(\min(VGL, VBL) \cdot \left\lceil \frac{\max(VGL, VBL)}{VD} \right\rceil\right) \times \left(\min(HGL, HBL) \cdot \left\lceil \frac{\max(HGL, HBL)}{HD} \right\rceil\right). \tag{25}$$

From the equations above we can derive an optimal choice for the matrix size:

$$D_{opt} = \{D : D \mid \max(GL, BL)\}. \tag{26}$$

4 Design implementation and complexity evaluation

In order to evaluate our scheme, we have verified it using a MatLab model, implemented it in VHDL and performed technology independent complexity evaluation in terms of wire complexity and logic complexity. The parallel memory controller which exploits the proposed scheme is implemented between the main memory and the processing unit (see Fig. 3). It aims at shadowing the high-latency channel to the main memory by means of a wide bus connection to the processing unit which performs parallel transmissions of data. The concurrently accessed matrix of memory modules is placed inside the memory controller. The pattern parameters are transmitted to the memory controller via programmable Special Purpose Registers (SPRs).

Fig. 3. Integration of parallel memory controller.

Structurally, the memory controller consists of an address generation part, a data routing part and a matrix of memory modules (see Fig. 4). The address generation part is split in vertical and horizontal sides that completely mirror each other. It includes the following blocks: mode select, address generator, set of row address generators, module assignment and address shuffle. The data routing part consists of a number of input shuffles and output de-shuffles. Refer to Appendix A for the VHDL source code of each sub-module of the design.

The critical path passes through the address generator, module assignment unit and decoding part of the shuffle. Hereafter the memory controller blocks are described in details.

Fig. 4. Parallel memory controller block diagram.

4.1 Mode select

The mode select unit sets the address generation logic to a mode, corresponding to the six cases of the problem partitioning (see Fig. 2). The pattern parameters stride S, group length GL and block length BL are read from the programmable SPRs.

The block implements stride oddness check and resolving of two inequalities: $\left\lceil \frac{BL}{D} \right\rceil \cdot GL \geq \left\lceil \frac{GL}{D} \right\rceil \cdot BL$ and $s \geq d$ (see Fig. 5). The *Counter** includes logic for power of two equality check, i.e. it counts number of logic '1' in the input signal: if there is only one logic '1', then the input is equal to power of two and the output is set to logic '1', otherwise the output is set to logic '0'. The *Coder* block performs coding of four 1-bit signals according to the problem partition diagram (Fig. 2) in order to create 3-bit *Mode* signal. The correspondence between cases and *Mode* signal is outlined in Table 2.

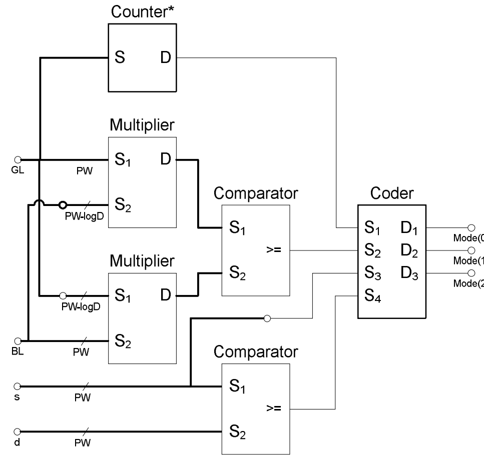

Fig. 5. Mode select block diagram.

The hardware complexity almost does not depend on the size of the matrix of memory modules $VD \times HD$, nor on the data word length W since the width of the input signal is constant and equals to 16 bits in our implementation. The wire complexity is constant and does not depend on the quantity of memory modules, data or address widths.

Table 2. Correspondence table.

Case #	*Mode signal*		
	bit 2	bit 1	bit 0
Case I.	0	0	0
Case II.	0	0	1
Case III.	0	1	0
Case IV.	0	1	1
Case V.	1	0	0
Case VI.	1	0	1
Reserved	1	1	0
Reserved	1	1	1

4.2 Address generator

The address generator produces vertical/horizontal constituents of two-dimensional addresses of the accessed data pattern according to formula (2). Data pattern parameters are read from SPRs

and an address mode is loaded from the mode select block. The address generator consists of two double parallel counters and one single parallel counter (see Fig. 6) that generate the sequence of pairs of indices *(i, k)* or *(j, l)* (refer to Table 1, section 3.1 and equations (8),(11)). The double counters generate group and element indices separately (for cases I-IV), and the single counter generates group and element indices on the base of a common index by implementing respectively division and modulo by group length (for cases V-VI). Note that the group length is equal to power of two for cases V-VI therefore division and modulo operations become possible. One side of a parallel double counter is presented on Fig. 7.

The complexity of the address generator block is $O(w_A \cdot D)$ because of the multiplier with input signal width depending on the size of the matrix of memory modules.

The critical path passes from the register inside a double counter and goes through the double counter, one multiplexer, one multiplier[1] and one adder to the address output. The critical path is equivalent to $O(\log D)$.

The wire complexity is equal to $O(w_A \cdot D)$ since the address generator produces D addresses of width w_A for each of the two dimensions.

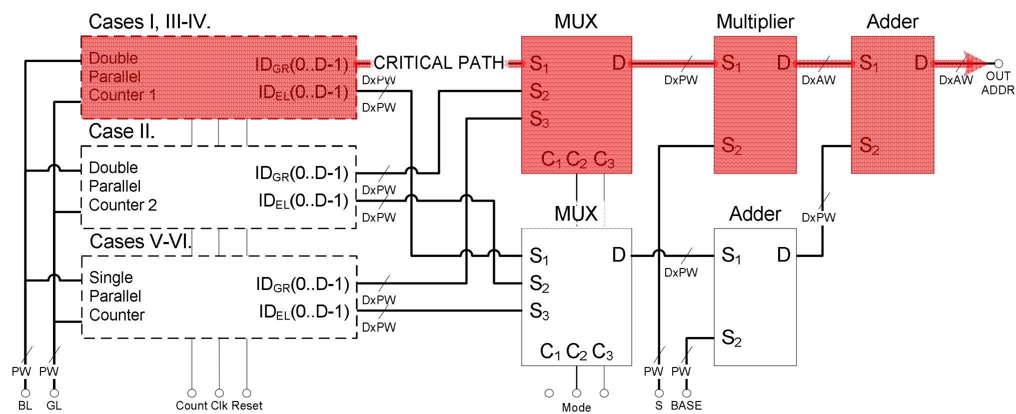

Fig. 6. Address generator block diagram.

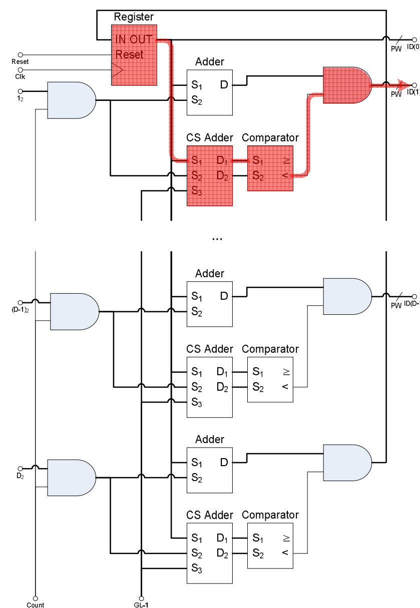

Fig. 7. Parallel counter block diagram.

---

[1] In the actual VHDL implementation, this multiplier was unrolled in a set of adders and pre-calculated in parallel with the counters. This allowed to reduce the critical path though its length still equals to $O(\log D)$.

4.3 Row address generator

The row address generator translates vertical/horizontal constituents of two-dimensional addresses into the physical addresses inside memory modules according to equation (23). Since formula (23) is completely separable, vertical and horizontal row address generators are implemented in separable blocks (see Fig. 4). Consequently, vertical blocks generate upper bits of the row address, and horizontal blocks generate the lower bits. No additional logic is needed to implement this block. The wire complexity is equivalent to the address width: $O(w_A)$.

4.4 Module assignment unit

The module assignment unit translates vertical/horizontal constituents of two-dimensional addresses into memory module addresses inside the matrix of memory modules according to equations (7), (14), (16), (18), and (21). Data pattern parameters are read from SPRs and an address mode is loaded from the mode select block. The equations are implemented in parallel and their outputs are multiplexed according to the address mode (see Fig. 8).

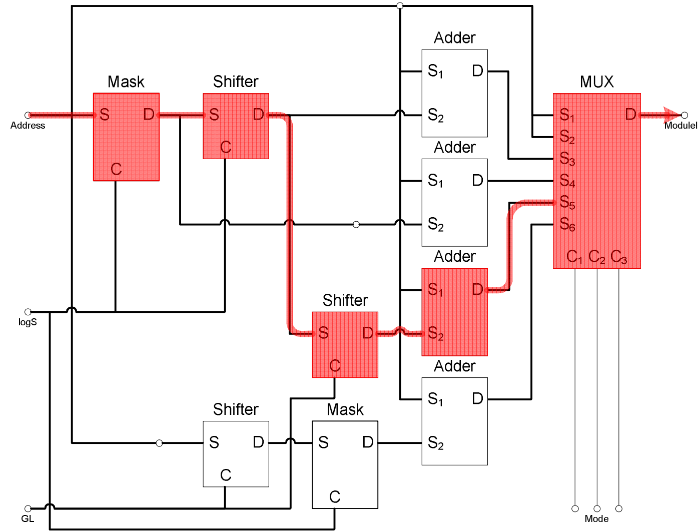

Fig. 8. Module assignment unit block diagram.

Complexity of the module assignment function for all cases (refer to equations (7), (14), (16), (18), and (21)) is presented in Table 3. The notation $x_{ms:ls}$ represents the bit interval from the least significant bit ls to the most significant bit ms. The complexity of the complete block is proportional to $O(\log D)$ because of the adders with input signals of the maximum width equal to $\log D$.

The wire complexity is equal to $O(w_A \cdot D)$ since the module assignment unit produces D results basing on the input address of width w_A.

The critical path passes through the mask unit, shifters, one adder and one multiplexer. Its length is mostly influenced by the adder of width $\log D$ and is equivalent to $O(\log(\log D))$.

Table 3. Module assignment function complexity for different cases.

Case #	*Complexity*	
Cases I-II.	$m(a) = a_{d-1:0}$	**(27)**
Case III.	$m(a) = (a_{d-1:0} + a_{s+d-1:s})_{d-1:0}$	**(28)**
Case IV.	$m(a) = (a_{d-1:0} + a_{s+d-1:d})_{d-1:0}$	**(29)**

Case V.	$m(a) = \left(a_{d-1:0} + a_{s+d-1:s} \cdot 2^{gl}\right)_{d-1:0}$	**(30)**
Case VI.	$m(a) = \left(a_{d-1:0} + \left(a_{8\cdot W-1:d} \cdot 2^{gl}\right)_{s-1:0}\right)_{d-1:0}$	**(31)**

4.5 Shuffle unit

The shuffle unit is used to reorder row addresses, received from the row address generators, according to the module assignment function. It consists of a parallel set of de-multiplexers and output OR-gates (see Fig. 9).

Its complexity is $O(w_A \cdot D)$. The biggest shuffle in the design is the Data IN shuffle (see Fig. 4). Its wire complexity is equal to $O(D^2 \cdot W)$ since it has D^2 inputs and the same amount of outputs of width W.

The critical path passes through a multiplexer via its select port and does not depend on D.

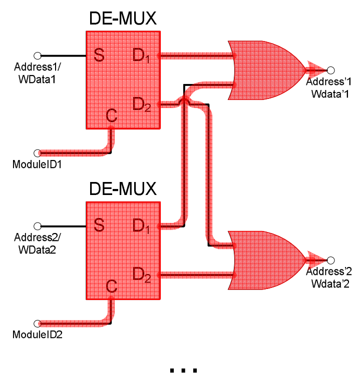

Fig. 9. Shuffle unit block diagram.

4.6 De-Shuffle unit

The de-shuffle unit is needed to reorder the data from memory modules back to the initial sequence. It consists of a set of parallel multiplexers (see Fig. 10).

The complexity of the shuffle unit is $O(w_A \cdot D)$. The widest de-shuffle is situated at the Data OUT. Its wire complexity is similar to the shuffle's one and equals to $O(D^2 \cdot W)$.

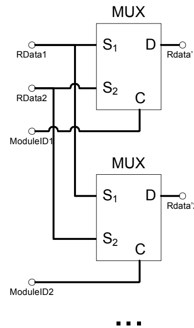

Fig. 10. De-Shuffle unit block diagram.

5 Results and analysis

The technology independent complexity estimations from Table 4 indicate that the critical path complexity is weakly sensitive to the size of the memory matrix and thus the design is well scalable to any matrix size. In fact, the throughput is directly proportional to the matrix size *VD*×*HD*, and inversely proportional to the critical path, i.e. $throughput \propto W \cdot D^2 / \log D$.

Table 4. Summary of the technology independent design complexity evaluation.

Design unit	*Logic complexity*	*Wire complexity*	*Critical path*
Mode select	$O(const)$	$O(const)$	-
Address generator	$O(w_A \cdot D)$	$O(w_A \cdot D)$	$O(\log D)$
Row address generator	0	$O(w_A)$	-
Module assignment unit	$O(\log D)$	$O(w_A \cdot D)$	$O(\log(\log D))$
Shuffle	$O(w_A \cdot D)$	$O(D^2 \cdot W)$	$O(const)$
De-shuffle	$O(w_A \cdot D)$	$O(D^2 \cdot W)$	-
Total	$O(w_A \cdot D)$	$O(D^2 \cdot W)$	$O(\log D)$

5.1 ASIC synthesis

The synthesis was performed for an ASIC 90 nm CMOS technology. The results for six different matrix sizes, word widths *W* of 32 and 64 bits, and 12-bit addresses are presented in Table 5 and **Ошибка! Источник ссылки не найден.**. In fact, data word width of 8 Bytes corresponds to utilization of two concurrently coupled 32-bit wide memory modules. Generally speaking, the address width ranging from 8 till 16 bits is enough for the most of practical applications, which would give the complexity range of 45.5-53.9 Kgates for 4×4 matrix with *W* = 32 bits. Considering that the memory modules used in the design have 4096×32bits size and occupy 43.8 Kgates, the logic complexity overheads vary from 14.5% for 2×2 32-bits matrix to 3.8% for 8×8 64-bits matrix with respect to the total hardware complexity. The presented synthesis results confirm the linear increase of the design complexity and the quadratic increase of the throughput, derived from our theoretical estimations. As it was expected, the critical path is proportional to the logarithm of the matrix size along one dimension and the design complexity depends linearly on it.

Table 5. Synthesis results for ASIC 90 nm.

Matrix size	*Complexity (KGates)*		*Frequency (MHz)*		*Throughput (Gbits/sec)*	
	W=4	*W*=8	*W*=4	*W*=8	*W*=4	*W*=8
2×2	25.34	26.73	377	371	44.94	88.45
2×4	33.81	39.11	341	336	81.30	160.21
2×8	58.48	70.19	314	321	149.72	306.12
4×4	46.60	53.27	336	333	160.21	317.57
4×8	88.07	101.57	321	314	306.12	598.90
8×8	176.83	211.06	313	310	597.00	1182.55

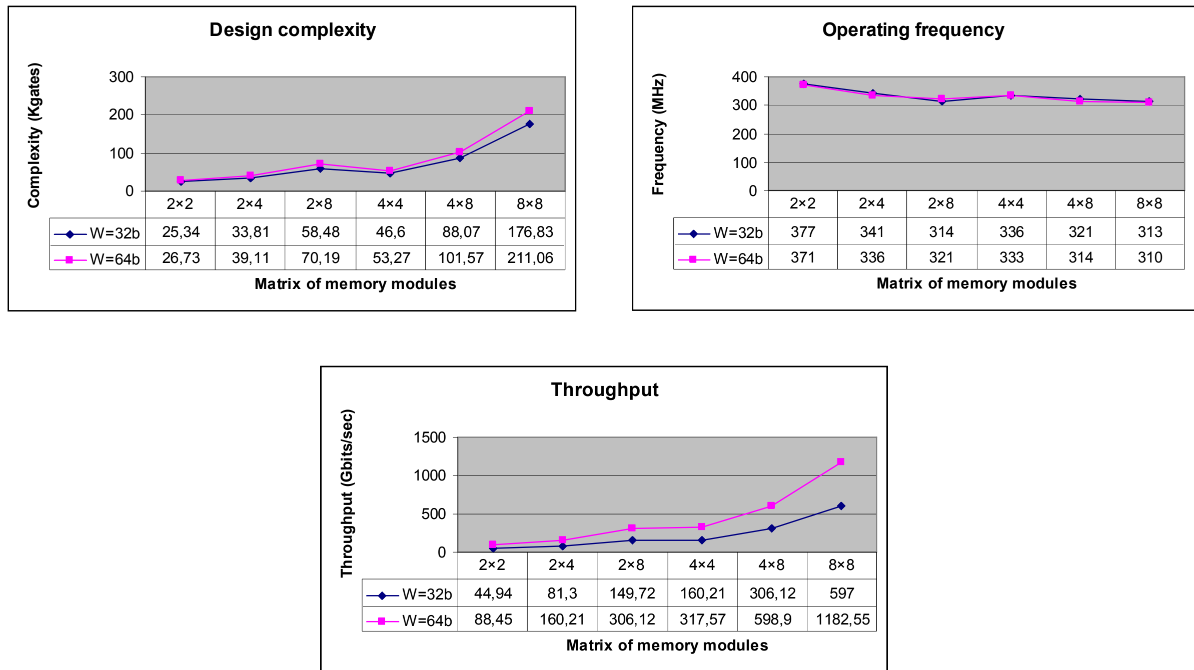

Fig. 11. Synthesis results for ASIC 90 nm: design complexity, frequency and throughput.

5.2 FPGA synthesis

The FPGA synthesis was performed with Xilinx ISE 8.2i toolset for Virtex2P xc2vp30-7ff896 device with speed coefficient -7. The results are presented in Table 6.

Table 6. FPGA synthesis results.

Matrix size	IO ports utilization (%)			Frequency (MHz)			Throughput (Gbits/sec)		
	W=1	W=2	W=4	W=1	W=2	W=4	W=1	W=2	W=4
2×2	16	22	58	128.8	128.8	108.4	4.12	8.24	13.87
2×4	22	34	-	123.3	123.3	-	7.89	15.78	-
2×8	35	58	-	134.4	134.4	-	17.20	34.40	-
4×4	35	58	-	124.6	124.6	-	15.94	31.89	-
4×8	61	-	-	133.2	-	-	34.09	-	-

In contrast to ASIC, design for FPGA is much more sensitive to the wire complexity which is significant for the parallel systems. Therefore, FPGA technology enables implementation of a restricted variety of configurations outlined in Table 6. On the other hand, FPGA technology allows mapping the design within a short timeframe and performing the experiments on real-applications to prove the concept of parallel memory access with configurable 2D data patterns.

5.3 Comparison with the related work

Table 7 presents the comparison with the related schemes from [3] and [2]. The memory access latency is presented for the case when the number of accessed elements is smaller than the number of memory modules. In our design it is calculated by formulas (24), (25) and is equal to one cycle for the best case and four cycles for the worst case. For instance, a data pattern with even stride and group length equal to the power of two will always have one cycle of latency. The large area occupied by our scheme is accounted for its higher flexibility. However, the modularity feature allows significant simplification of the design by reducing the set of the

supported data patterns. The critical path in our design scales according to $O(log(max\{VD,HD\}))$ while in the other schemes it scales linearly with the total number of memory modules $VD \times HD$.

Table 7. Comparison to the schemes with 8 memory modules and 8 bits data width.

Design	Memory latency (r/w cycles)	Complexity (Kgates)	Frequency (MHz)
CPMA [3]	11/8 pipelined	~27.0	~13 (0.25CMOS)
PMAS [2]	3/2 pipelined	5.5	256 (0.18CMOS)
This proposal	1/1 bc – 4/4 wc	26.9	393 (0.09CMOS)

6 Conclusion

High throughput memory accesses with flexible data patterns are widely used in many different areas such as multimedia, telecommunications, and scientific applications. We presented a parallel memory organization that accumulates the advantages of the previous solutions. In addition, it allows access to a data pattern with more complex structure and relaxes the limitations of the data pattern parameters. Runtime programmability by means of SPRs enables flexible data management. Our theoretical conclusions were proved, and additionally confirmed by mathematical modeling. The design implementation and synthesis showed expected results according to our theoretical estimations. As a result, our memory organization provides minimum latency between main memory and processing unit for a given type of schemes.

Appendix A. VHDL sources.

Mode select

```
library ieee;
use ieee.std_logic_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.STD_LOGIC_UNSIGNED.all;
use work.gsma_package.all;

entity mode_select is
  generic (
    logD : natural := 1);              -- Logarithm to the base 2 of number of
                                       -- modules along one dimention
  port (
    logS    : in  std_logic_vector(logParam_width - 1 downto 0);  -- Logarithm to
                                                                  -- the base 2 of
                                                                  -- S
    GL      : in  std_logic_vector(param_width - 1 downto 0);  -- Group length
    BL      : in  std_logic_vector(param_width - 1 downto 0);  -- Block length
    mode    : out std_logic_vector(2 downto 0)                 -- Case enable
    );
end mode_select;

architecture Behavioral of mode_select is
  signal stride_oddness             : std_logic;  -- '1' = odd stride
  signal GL_norm, BL_norm           : unsigned(2 * param_width - logD - 1 downto 0);
  signal pot                        : boolean;

begin  -- Behavioral
  stride_oddness <= '1' when conv_integer(unsigned(logS)) = 0 else  -- S(0);
                    '0';
  GL_norm <= conv_unsigned(unsigned(BL(param_width - 1 downto logD)) * unsigned(GL), 2 * param_width - logD) when
             conv_integer(unsigned(BL(logD - 1 downto 0))) = 0 else
             conv_unsigned((unsigned(BL(param_width - 1 downto logD)) + 1) * unsigned(GL), 2 * param_width - logD);
  BL_norm <= conv_unsigned(unsigned(GL(param_width - 1 downto logD)) * unsigned(BL), 2 * param_width - logD) when
             conv_integer(unsigned(GL(logD - 1 downto 0))) = 0 else
             conv_unsigned((unsigned(GL(param_width - 1 downto logD)) + 1) * unsigned(BL), 2 * param_width - logD);
  mode <= "000" when (stride_oddness = '1') and (GL_norm < BL_norm) else  -- Case I.
          "001" when ((stride_oddness = '1') and (GL_norm >= BL_norm)) or
          ((stride_oddness = '0') and (not po2(GL)) and (GL_norm >= BL_norm)) else  -- Case II.
          "010" when (stride_oddness = '0') and (not po2(GL)) and
          (GL_norm < BL_norm) and (unsigned(logS) >= logD) else  -- Case III.
          "011" when (stride_oddness = '0') and (not po2(GL)) and
          (GL_norm < BL_norm) and (unsigned(logS) < logD) else  -- Case IV.
          "100" when (stride_oddness = '0') and po2(GL) and (unsigned(logS) >= logD) else  -- Case V.
          "101" when (stride_oddness = '0') and po2(GL) and (unsigned(logS) < logD) else  -- Case VI.
          "XXX";
end Behavioral;
```

Address generator

```
library ieee;
use ieee.std_logic_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.STD_LOGIC_UNSIGNED.all;
use work.gsma_package.all;

entity AddrGen is
  generic (
    D    : natural;                    -- Number of memory modules along one dimension
    logD : natural
    );
```

```
  port (
    clk         : in  std_logic;
    RESETn      : in  std_logic;
    ready       : in  std_logic;      -- Enables addr generation process
    mode        : in  std_logic_vector(2 downto 0);
    B           : in  std_logic_vector(addr_width - 1 downto 0);  -- Linear base address
    S           : in  std_logic_vector(param_width - 1 downto 0);  -- Stride
    GL          : in  std_logic_vector(param_width - 1 downto 0);  -- Group length
    BL          : in  std_logic_vector(param_width - 1 downto 0);  -- Block length
    output_valid : out std_logic_vector(D - 1 downto 0);  -- Output velidness
    last_output  : out std_logic;      -- Signals the last output address
    a           : out addr_bus(D - 1 downto 0));  -- Output addresses
end AddrGen;

architecture Behavioral of AddrGen is
  type index_type is array (natural range <>) of std_logic_vector(param_width - 1 downto 0); --natural range 0 to
2**param_width - 1;

  signal i, k                           : index_type(D downto 0);
  signal ind                            : std_logic_vector(param_width - 1 downto 0); --natural range 0 to
2**param_width - 1;
  signal carry_i, carry_k, carry_ind    : std_logic_vector(param_width - 1 downto 0); --natural range 0 to
2**param_width - 1;
  signal carry_last_output, last_output_i : std_logic;
  signal output_valid_i                 : std_logic_vector(D - 1 downto 0);
  signal ibyS, iS_prec                  : index_type(D downto 0);
  signal carry_ibyS                     : std_logic_vector(param_width - 1 downto 0);

begin  -- Behavioral

  TPC_REGs: process (clk, RESETn)
  begin  -- process
    if RESETn = '0' then               -- asynchronous reset (active low)
      carry_i     <= (others => '0');
      carry_k     <= (others => '0');
      carry_ind   <= (others => '0');
      carry_ibyS  <= (others => '0');
    elsif clk'event and clk = '1' then  -- rising clock edge
      -- Counter indices
      if ready = '1' then
        carry_i   <= i(D);             -- i + 1
        carry_k   <= k(D);             -- k + 1
        carry_ind <= ind;              -- (i * k) + 1
        carry_ibyS <= ibyS(D);         -- (i * S) + S
      end if;
    end if;
  end process TPC_REGs;

  -- Preculculation of the (i * S) product
  iS_prec(0) <= carry_ibyS;
  ibyS(0) <= carry_ibyS;
  MULT_PREC: for id in 1 to D generate
    iS_prec(id) <= iS_prec(id - 1) + S;  -- precalculated value
    ibyS(id) <= iS_prec(id)  when i(id) > i(id - 1) else
                ibyS(id - 1) when i(id) = i(id - 1) else
                (others => '0');
  end generate MULT_PREC;

  -- Tripple Parallel Counter
  TPC: process (mode, GL, BL, carry_k, carry_i, carry_ind)
    variable i_id, k_id, ind_id, temp_id   : natural;
    variable save_i : natural;
    variable lst_output, output_valid_i_id : std_logic;
    variable temp : std_logic_vector(param_width - 1 downto 0);
```

```
begin

  k(0)              <= carry_k;
  i(0)              <= carry_i;
  output_valid_i(0) <= '1';

  case mode is

    when "001" =>                    -- Group-wise access
      -- Current indices
      for id in 1 to D - 1 loop
        if carry_k + id > GL - 1 then
          k(id) <= GL - 1;
          output_valid_i(id) <= '0';
        else
          k(id) <= carry_k + id;
          output_valid_i(id) <= '1';
        end if;
        i(id) <= carry_i;
      end loop;  --id
      -- Next carry indices
      if carry_k + D > GL - 1 then
        k(D) <= (others => '0');                  -- reset k syncronously
        if carry_i = BL - 1 then
          i(D) <= (others => '0');                -- reset i syncronously
          last_output_i <= '1';
        else
          i(D) <= carry_i + 1;
          last_output_i <= '0';
        end if;
      else
        k(D) <= carry_k + D;
        i(D) <= carry_i;
        last_output_i <= '0';
      end if;
      ind <= carry_ind;

    when "000" | "010" | "011" =>     -- Access based on the set of elements
      -- Current indices
      for id in 1 to D - 1 loop
        if carry_i + id > BL - 1 then
          i(id) <= BL - 1;
          output_valid_i(id) <= '0';
        else
          i(id) <= carry_i + id;
          output_valid_i(id) <= '1';
        end if;
        k(id) <= carry_k;
      end loop;  --id
      -- Next carry indices
      if carry_i + D > BL - 1 then
        i(D) <= (others => '0');
        if carry_k = GL - 1 then
          k(D) <= (others => '0');
          last_output_i <= '1';
        else
          k(D) <= carry_k + 1;
          last_output_i <= '0';
        end if;
      else
        i(D) <= carry_i + D;
        k(D) <= carry_k;
        last_output_i <= '0';
      end if;
```

```
        ind <= carry_ind;

      when "100" | "101" =>            -- Element-wise access
        -- Current indices
        for id in 1 to D - 1 loop
          if (carry_k + id = GL - 1) and (carry_i + id = BL - 1) then
            k(id) <= GL - 1;
            i(id) <= BL - 1;
            output_valid_i(id) <= '0';
          else
            k(id) <= carry_k + id; -- mod2(carry_ind + id, GL);
            i(id) <= div2(carry_ind + id, GL);
            output_valid_i(id) <= '1';
          end if;
        end loop;  -- id
        -- Next carry indices
        if (carry_k + D = GL - 1) and (carry_i + D = BL - 1) then
          ind  <= (others => '0');
          k(D) <= (others => '0');
          i(D) <= (others => '0');
          last_output_i <= '1';
        else
          ind <= carry_ind + D;
          k(D) <= carry_k + D; -- mod2(carry_ind + D, GL);
          i(D) <= div2(carry_ind + D, GL);
          last_output_i <= '0';
        end if;

      when others =>
        k              <= (others => (others => 'X'));
        i              <= (others => (others => 'X'));
        ind            <= (others => 'X');
        output_valid_i <= (others => 'X');
        last_output_i  <= 'X';

    end case;

  end process TPC;

  -- Outport connections
  ADDR_BUS : for m_id in 0 to D - 1 generate
    a(m_id) <= std_logic_vector(conv_unsigned(unsigned(B + ibyS(m_id) + k(m_id)), addr_width)); -- implicit
multiplication and base address
  end generate ADDR_BUS;
  last_output  <= last_output_i;
  output_valid <= output_valid_i;

end Behavioral;
```

Row address generator

```
library ieee;
use ieee.std_logic_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.STD_LOGIC_UNSIGNED.all;
use work.gsma_package.all;

entity RowGen is
  generic (
    logD :     natural := 1);           -- Logarithm to the base 2 of number of modules along one dimention
  port (
    a    : in  std_logic_vector(addr_width - 1 downto 0);   -- One dimention constituent of the input address
    row  : out std_logic_vector(addr_width - 1 downto 0));  -- One dimention constituent of the row address
end RowGen;
```

```
architecture Behavioral of RowGen is

  signal row_un, a_un : unsigned(addr_width - 1 downto 0);

begin  -- Behavioral

  -- Shifter is implemented afterwords as high and low memory module addresses (simplified)
  row(addr_width - logD - 1 downto 0)          <= a(addr_width - 1 downto logD);
  row(addr_width - 1 downto addr_width - logD) <= low_vector(logD - 1 downto 0);

end Behavioral;
```

Module assignment unit

```
library ieee;
use ieee.std_logic_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.STD_LOGIC_UNSIGNED.all;
use work.gsma_package.all;

entity ModuleAssign is

  generic (
    logD : natural := 1);
  port (
    a        : in  std_logic_vector(addr_width - 1 downto 0);  -- One dimention constituent of the input address
    logS     : in  std_logic_vector(logParam_width - 1 downto 0);  -- Logarithm to the base 2 of S
    GL       : in  std_logic_vector(param_width - 1 downto 0);  -- Group length
    mode     : in  std_logic_vector(2 downto 0);  -- Case enable
    ModuleID : out std_logic_vector(logD - 1 downto 0));  -- Module select
end ModuleAssign;

architecture Behavioral of ModuleAssign is

  signal GL_un                  : unsigned(param_width - 1 downto 0);
  signal m1, m2, m3, m4, m5, m6 : unsigned(logD - 1 downto 0);
  signal prod5, prod6           : unsigned(addr_width - 1 downto 0);
  signal a_msk3, a_msk6, a_sh3  : std_logic_vector(addr_width - 1 downto 0);
  signal logS_nat               : natural;
  signal ModuleID_tmp           : std_logic_vector(logD - 1 downto 0);

begin  -- Behavioral

  logS_nat <= conv_integer("0" & logS);

  -- CASE I.
  m1 <= unsigned(a(logD - 1 downto 0));  -- m1 = a[d - 1 : 0]

  -- CASE II.
  m2 <= unsigned(a(logD - 1 downto 0));  -- m2 = a[d - 1 : 0]

  -- CASE III.
  masking3 : for i in 0 to addr_width - 1 generate
    a_msk3(i) <= a(i) when i < logS_nat + logD else  -- a_msk3 = a[s + d - 1 : 0]
                 '0';
  end generate masking3;
  a_sh3       <= to_stdLogicVector(to_bitVector(a_msk3) srl logS_nat);  -- a_sh3 = a[s + d - 1 : s]
  m3          <= conv_unsigned(unsigned(a(logD - 1 downto 0)) + unsigned(a_sh3), logD);
                                     -- m3 = (a[d - 1 : 0] + a[s + d - 1 : s])[d - 1 : 0]

  -- CASE IV.
  m4 <= conv_unsigned(unsigned(a(logD - 1 downto 0)) + unsigned(a_msk3(addr_width - 1 downto logD)), logD);
                                     -- m4 = (a[d - 1 : 0] + a[s + d - 1 : d])[d - 1 : 0]
```

```
  -- CASE V.
  prod5 <= conv_unsigned(unsigned(to_stdLogicVector(to_bitVector(a_sh3) sll log2(GL))), addr_width);
                                        -- prod5 = a[s + d - 1 : s] * 2^gl
  m5    <= conv_unsigned(unsigned(a(logD - 1 downto 0)) + prod5, logD);
                                        -- m5 = (a[d - 1 : 0] + a[s + d - 1 : s] * 2^gl)[d - 1 : 0]

  -- CASE VI.
  prod6      <= conv_unsigned(unsigned(to_stdLogicVector(to_bitVector(a(addr_width - 1 downto logD)) sll log2(GL))),
addr_width);
                                        -- prod6 = a[addr_width - 1 : d] * 2^gl
  masking6 : for i in 0 to addr_width - 1 generate
    a_msk6(i) <= prod6(i) when i < logS_nat else  -- a_msk6 = (a[addr_width - 1 : d] * 2^gl)[s - 1 : 0]
                 '0';
  end generate masking6;
  m6         <= conv_unsigned(unsigned(a(logD - 1 downto 0)) + unsigned(a_msk6), logD);
                                        -- m6 = (a[d - 1 : 0] + (a[addr_width - 1 : d] * 2^gl)[s - 1 : 0])[d - 1 : 0]

  -- Case multiplexing
  ModuleID_tmp <= std_logic_vector(m1) when mode = "000" else
                  std_logic_vector(m2) when mode = "001" else
                  std_logic_vector(m3) when mode = "010" else
                  std_logic_vector(m4) when mode = "011" else
                  std_logic_vector(m5) when mode = "100" else
                  std_logic_vector(m6) when mode = "101" else
                  (others => 'X');

  ModuleID <= ModuleID_tmp;

end Behavioral;
```

Shuffle

```
library ieee;
use ieee.std_logic_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.STD_LOGIC_UNSIGNED.all;
use work.gsma_package.all;

entity Shuffle is
  generic (
    sub_bus_num     : natural := 2;
    log_sub_bus_num : natural := 1;
    sub_bus_width   : natural := 1);
  port (
    I_bus : in  std_logic_vector(sub_bus_num * sub_bus_width - 1 downto 0);  -- Input busses
    S_bus : in  std_logic_vector(sub_bus_num * log_sub_bus_num - 1 downto 0);  -- Select signal busses
    O_bus : out std_logic_vector(sub_bus_num * sub_bus_width - 1 downto 0));  -- Shuffled busses

end Shuffle;

architecture Behavioral of Shuffle is

  type inner_subtype is array (sub_bus_num - 1 downto 0) of std_logic_vector(sub_bus_width - 1 downto 0);
  type inner_type is array (sub_bus_num - 1 downto 0) of inner_subtype;
  type inner_nat_type is array (sub_bus_num - 1 downto 0) of natural;

  signal O_bus_tmp : std_logic_vector(sub_bus_num * sub_bus_width - 1 downto 0);
  signal inner_sig : inner_type;
  signal sel : inner_nat_type;

begin  -- Behavioral
```

```
  -- Set of deMUXs
  SET_OF_deMUXs : for demux_id in 0 to sub_bus_num - 1 generate
    -- Select signal decoder
    sel(demux_id) <= conv_integer(unsigned(S_bus((demux_id + 1) * log_sub_bus_num - 1 downto demux_id *
log_sub_bus_num)));
    -- DeMUX
    DeMUX : for output_id in 0 to sub_bus_num - 1 generate
      inner_sig(demux_id)(output_id)(sub_bus_width - 1 downto 0) <=
        I_bus((demux_id + 1) * sub_bus_width - 1 downto demux_id * sub_bus_width) when output_id = sel(demux_id) else
        low_vector(sub_bus_width - 1 downto 0); --(others => '0');
    end generate DeMUX;
  end generate SET_OF_deMUXs;

  -- Set of OR-gates
  OUTPUT_OR_GATES: process (inner_sig)
    variable O_bus_var : std_logic_vector(sub_bus_num * sub_bus_width - 1 downto 0);
  begin  -- process OUTPUT_OR_GATES
    for or_id in 0 to sub_bus_num - 1 loop
      -- OR-gate
      O_bus_var((or_id + 1) * sub_bus_width - 1 downto or_id * sub_bus_width) :=
        inner_sig(0)(or_id)(sub_bus_width - 1 downto 0);
      for input_id in 1 to sub_bus_num - 1 loop
        O_bus_var((or_id + 1) * sub_bus_width - 1 downto or_id * sub_bus_width) :=
          O_bus_var((or_id + 1) * sub_bus_width - 1 downto or_id * sub_bus_width) or
          inner_sig(input_id)(or_id)(sub_bus_width - 1 downto 0);
      end loop;  -- input_id
    end loop;  -- or_gate_id
    O_bus_tmp <= O_bus_var;
  end process OUTPUT_OR_GATES;
  O_bus <= O_bus_tmp;

end Behavioral;
```

De-shuffle

```
library ieee;
use ieee.std_logic_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.STD_LOGIC_UNSIGNED.all;
use work.gsma_package.all;

entity De_Shuffle is
  generic (
    sub_bus_num     : natural := 2;
    log_sub_bus_num : natural := 1;
    sub_bus_width   : natural := 1);
  port (
    I_bus : in  std_logic_vector(sub_bus_num * sub_bus_width - 1 downto 0);  -- Input busses
    S_bus : in  std_logic_vector(sub_bus_num * log_sub_bus_num - 1 downto 0);  -- Select signal busses
    O_bus : out std_logic_vector(sub_bus_num * sub_bus_width - 1 downto 0));  -- Shuffled busses

end De_Shuffle;

architecture Behavioral of De_Shuffle is

  signal O_bus_tmp : std_logic_vector(sub_bus_num * sub_bus_width - 1 downto 0);

begin  -- Behavioral

  MUXs           : process (I_bus, S_bus)
    variable sel : natural;             -- natural_vector(sub_bus_num - 1 downto 0);
  begin  -- process MUXes
    O_bus_tmp <= (others => '0');
    for mux_id in 0 to sub_bus_num - 1 loop  -- Block of MUXes
```

```
      sel := conv_integer(S_bus((mux_id + 1) * log_sub_bus_num - 1 downto mux_id * log_sub_bus_num));  -- Select
signal
      for input_id in 0 to sub_bus_num - 1 loop  -- MUX inputs
        if input_id = sel then
          O_bus_tmp((mux_id + 1) * sub_bus_width - 1 downto mux_id * sub_bus_width) <=
            I_bus((input_id + 1) * sub_bus_width - 1 downto input_id * sub_bus_width);
        end if;
      end loop;
    end loop;
  end process MUXs;

  O_bus <= O_bus_tmp;

end Behavioral;
```

References

[1] E. Aho, J. Vanne, and T.D. Hamalainen. Parallel memory architecture for arbitrary stride accesses. *Design and Diagnostics of Electronic Circuits and systems, 2006 IEEE*, pages 63–68, 2006.

[2] E. Aho, J. Vanne, and T.D. Hamalainen. Parallel memory implementation for arbitrary stride accesses. In *Embedded Computer Systems: Architectures, Modeling and Simulation, 2006 International Conference on*, pages 1–6, July 2006.

[3] E. Aho, J. Vanne, K. Kuusilinna, and T.D. Hamalainen. Address computation in configurable parallel memory architecture. *IEICE T. on Information and Systems*, E87-D(7):1674–1681, July 2004.

[4] D.J. Budnik, P. Kuck. The organization and use of parallel memories. *IEEE T. Comput.*, C-20:1566–1569, Dec. 1971.

[5] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. ScaLAPACK: A portable linear algebra library for distributed memory computers - design issues and performance. Technical report, Knoxville, TN 37996, USA, 1995.

[6] Jean Marc Frailong, William Jalby, and Jacques Lenfant. Xor-schemes: A flexible data organization in parallel memories. In *ICPP*, pages 276–283, 1985.

[7] D. Harper and J. Jump. Vector access performance in parallel memories using a skewed storage scheme. *IEEE Trans. on Comput.*, C-36:1440–1449, 1987.

[8] III Harper, D.T. Block, multistride vector, and fft accesses in parallel memory systems. *IEEE T. Parall. Distr.*, 2(1):43–51, 1991.

[9] III Harper, D.T. and D.A. Linebarger. Conflict-free vector access using a dynamic storage scheme. *IEEE T. Comput.*, 40(3):276–283, 1991.

[10] Kimmo Kuusilinna, Jarno Tanskanen, Timo HдmдIдinen, and Jarkko Niittylahti. Configurable parallel memory architecture for multimedia computers. *J. Syst. Archit.*, 47(14-15):1089–1115, 2002.

[11] G. Kuzmanov, G. Gaydadjiev, and S. Vassiliadis. Multimedia rectangularly addressable memory. *IEEE T. Multimedia.*, 8:315 – 322, April 2006.

[12] Coert Olmsted. Scientific sar user's guide. Technical Report asf-sd-003, AlaskaSatelliteFacility (ASF), July 1993.

[13] Jong Won Park. An efficient buffer memory system for subarray access. *IEEE T. Parall. Distr.*, 12(3):316–335, 2001.

[14] S. Rixner, W.J. Dally, U.J. Kapasi, P. Mattson, and J.D. Owens. Memory access scheduling. In *Computer Architecture, 2000. Proceedings of the 27th International Symposium on*, pages 128–138, 2000.

[15] H.D. Shapiro. Theoretical limitations on the efficient use of parallel memories. *Transactions on Computers*, C-27(5):421–428, May 1978.

[16] M. Valero, T. Lang, M. Peiron, and E. Ayguade. Conflict-free access for streams in multimodule memories. *IEEE T. Comput.*, 44(5):634–646, 1995.