

# Interface Synthesis for FPGA Based VLSI Processor Arrays

M. Bednara

Computer Engineering Laboratory  
Department of EE and IT  
University of Paderborn  
Paderborn, Germany

J. Teich\*

Computer Engineering Laboratory  
Department of EE and IT  
University of Paderborn  
Paderborn, Germany

## Abstract

*FPGA based VLSI processor arrays provide an enormous computation performance along with the flexibility of reconfigurable logic. In the future, hybrid processors will become available that consist of a RISC core and configurable logic area on the same die. This allows for the efficient usage of a VLSI array as a co-processor to the RISC core. Due to their high performance, VLSI array structures require interfaces with a large bandwidth, even under hard resource constraints as given in embedded systems. In this work, we present a generic interface hardware model that can be easily adapted to various performance and resource constraints.*

*Keywords:*

FPGA, Processor Arrays, Hardware Mapping, Interface

## 1 Introduction

VLSI processor arrays form a class of application specific and thus highly specialized architectures. For a wide area of applications, e.g., signal processing, linear algebra, combinatorial problems, and network processing, VLSI arrays provide the best performance since the inherent parallelism of the algorithm is exploited much better than on standard CPUs oder DSPs.

On the other hand, the high computation performance of these arrays causes a lack of flexibility when implemented in ASIC- or full custom technology, resulting in a poor cost-performance trade-off which makes them too expensive for consumer

market applications. Flexibility deficiency can be avoided by reconfigurable, i.e., FPGA based implementations. In [1] we have presented a methodology for automatic synthesis of FPGA based VLSI arrays for computation-intensive loop programs. This is part of our work on an automated continuous design flow for FPGA based VLSI arrays.

The massive internal parallelism of VLSI arrays requires for interfaces with a large bandwidth, while FPGAs cause strong resource limitations with respect to pin count, pad cell performance and memory elements.

**New contribution.** In this work, we show how resources of state-of-the-art FPGAs can efficiently be used for constructing host interfaces for VLSI arrays. We propose a generic interface architecture that can be adapted to various performance and resource constraints.

**Organization of the Paper.** Section 2 gives an overview of related work. In Section 3, some general interface requirements of VLSI arrays are given. We define four types of array ports with different bandwidth requirements. In Section 4, we present the array interface architecture that consists of hardware structures for the previously defined array port types. In Section 5, we focus on the port type called *random* type which requires the most complicated interface logic and show how this interface type can be optimized. Section 6 summarizes the paper.

---

\*This work is supported by DFG Sonderforschungsbereich 376 "Massive Parallelität."

## 2 Related Work

In [1], an automated design flow for mapping nested loop programs to regular arrays is given. More detailed information about partitioning, mapping theory and algorithm transformation is given in [4], [5], and [6]. The problem of interface synthesis for processor arrays is also discussed in [3] but here for the special case of a linear array of processing stages. A theoretical analysis of the data transfer between processor array and global memory along with an energy estimation is given in [2]. Here a hierarchical memory structure is presumed. However, an interface architecture is not proposed in that work.

## 3 Regular Array Interface Requirements

Throughout this paper, we assume that a regular array is a linear or two-dimensional array of identical processor elements (PE) along with a regular interconnection structure<sup>1</sup>. Input data can only be passed to border PEs (BPE) as well as output data is only available on BPEs.

Depending on the application context where the array is embedded, we can distinguish between several types of input and output data:

- Inputs of *constant* type are once initialized and hold their value during the whole computation. Before starting another computation, the value of the input may be changed. This is used, e.g., for digital filter coefficients.
- Values of *cyclic* type inputs are not constant but the sequence of values required on this input is predefined before computation starts. This type of inputs may be used, e.g., in a matrix multiplier that multiplies two matrices  $A$  and  $B$  where  $B$  is constant for a long time. Thus, the coefficients of  $B$  must be passed cyclic to the array.
- *random* type. These inputs receive arbitrary data from the array periphery (e.g. a host processor), like matrix  $A$  of the matrix multiplier.

---

<sup>1</sup>The results of this work, however, can generally applied also to inhomogenous VLSI arrays

- *stream* type data is a subtype of *random* but with another type of data source. This is typically used for passing a continuous data stream to the array like sensor data or audio/video streams.

For array outputs, only *random* and *stream* types are required. Due to FPGA resource limitations and the enormous bandwidth requirements of VLSI arrays, the complete array inputs cannot be supplied with input data from a host processor in each iteration period (for a definition of *iteration period* refer to Section 5.1). Our interface proposal which is presented in the next section offers a method for reducing the required maximum interface bandwidth.

Data of type *constant* and *cyclic* can be stored in an appropriate memory structure on the FPGA chip during an initialization phase. *Random* and *stream* type data must be passed to the array during run time from some data source outside the FPGA, e.g., a standard processor or a sensor group. Since the bandwidth requirements are the highest for these data types, we focus on the bandwidth minimization in Section 5.

All optimizations are done under the assumption that the interface circuitry operates on the same clock frequency as the processor array.

## 4 Interface Architecture

This section gives an overview of the array interface architecture. For each port type introduced in Section 3, we propose an interface logic. The structure of the complete array interface depends on the number of array input and output ports of each type. The modular design of the interface allows for an easy adaption to various arrays.

### 4.1 Port Types

**Constant and cycling.** Data words of type *constant* are stored in registers that are loaded during an initialization phase prior to the first computation of the array. Data words of type *cycling* are stored in a feed back FIFO memory also initialized before the first computation starts. Since this initialization phase is not time critical, all registers and FIFOs can be loaded via a serial bus. The architecture is shown in Fig. 1 The address decoder

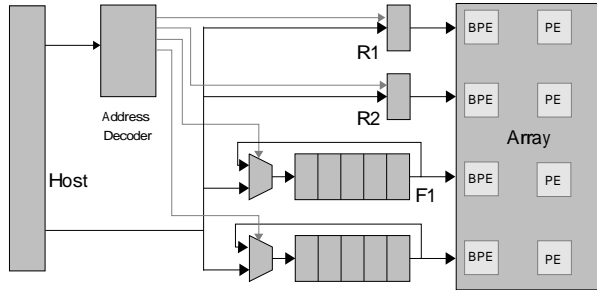


Figure 1: Interface structure for constant type and cycling type inputs

generates appropriate control signals for all registers, FIFOs, and multiplexers according to the host address. Registers and FIFOs can be mapped into the host processor’s address space or connected to an output channel of the host. In this case, the address decoder generates the control signals from an internal counter state instead of the host address.

Depending on the amount of *cycling* type data, FIFO buffers require a large area. On certain FPGA types, however, FIFOs can be implemented very efficiently by using LUT memories or dedicated block RAMs for FIFO buffers instead of flipflops.[10]

**Random.** Each BPE reads each of its inputs once per iteration period, thus, each *random* type input must be provided with a data word once per iteration period. We use a dual port memory (DPRAM) for buffering the host data before passed to the BPE. State-of-the-art FPGAs provide on-chip dedicated block memory elements that can be configured as DPRAMs of various depths and widths [7]. Even the input and output data width may be different which allows for one port to match the host interface bus width and the other port to match the required array input width. In most cases, however, the number of data words read by the array in parallel is larger than realistic DPRAM port widths. Thus, we have to provide the array input data words serially and use additional buffer registers. The resulting architecture is shown in Fig. 2. In Section 5, we analyze the minimum required DPRAM port width for a given array and show how to optimize the required number of buffer registers.

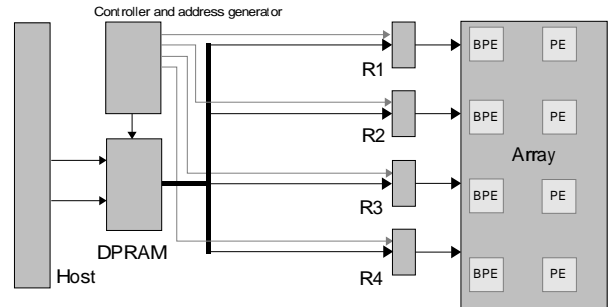


Figure 2: DPRAM based interface structure for random type input data

**Stream.** Input data words of type stream must also be passed to the array once per iteration period but are not provided by a host system but e.g. a sensor group. If the number of *stream* type inputs is small, the data words can be passed directly to the array via a FIFO buffer avoiding the bottleneck of the host interface.

**Array Outputs.** For array outputs, only *random* and *stream* types are used. The same architecture as for inputs can be used.

## 4.2 FPGA Interface

Ports of *random* and *stream* type may require large transfer rates to the external data source/sink. Generally, FPGA pins are subject to hard limitations with respect to transfer rates. FPGAs of the newest generation, however, support LVDS interfaces (Low Voltage Differential Signaling, [8]) which provide transfer rates up to 622 Mbit/sec on a serial line. As shown in [9], transfer rates of more than 1.2GByte/sec from/to a Xilinx Virtex-E FPGA can be obtained.

## 5 Bandwidth Optimization

In this section, we focus on the bandwidth optimization for *random* type inputs. Our goals are:

- Determine a lower bound for the required DPRAM port width
- Minimize the DPRAM port width

- Minimize the number of additionally required registers.

## 5.1 Linear Mapping and Scheduling

The mapping of nodes of regular dependence graphs to processor elements  $p$  is described in detail in [1], here we only give a short summary. Index vectors are denoted with  $I$  and the domain of all index vectors is the index space  $\mathbf{I}$ . The mapping is performed by a linear mapping function  $T = (Q \ \lambda)^T$ , where  $\lambda$  is a schedule vector and  $Q$  is a matrix that must fulfill  $Qu = 0$  where  $u$  is a projection vector. This leads to

$$\begin{pmatrix} p \\ t \end{pmatrix} = \begin{pmatrix} Q \\ \lambda \end{pmatrix} I = TI \quad (1)$$

$I$  is the index vector of an quantified equation,  $T$  is the transformation matrix,  $p$  the PE that the index vector  $I$  is mapped to, and  $t$  the time step when the computation of the quantified equation at  $I$  starts. Generally, the computation of an indexed equation requires more than one time step. The time between the startings of two subsequent computations on the same PE is called *iteration period*  $P$  and can be expressed as  $P = |\lambda u|$ . We consider only homogeneous algorithms here, so  $P$  is equal for all PEs.

As shown in [1], the computation of an indexed equation can be decomposed into a number of atomic arithmetic operations performed on the resources of a PE. Besides the schedule vector  $\lambda$  we also specify a start time  $\gamma$  for each atomic operation that denotes the number of time steps from the beginning of the iteration period until the operation starts.  $\gamma$  denotes the start time of an *operation*, but here the time steps where input data of a PE must be available is more interesting.

**Definition 5.1** ( $\omega$ -steps) *Let  $a_0, a_1, \dots, a_{n-1}$  the inputs of a PE. Then  $\omega(a_i), (0 \leq i \leq n-1)$  denote the number of time steps from the beginning of the iteration period until input  $a_i$  is read by the PE.*

For an operation  $op$  that depends directly on the PE inputs  $a_i, \dots, a_j$  and that starts at  $\gamma(op)$ , obviously holds  $\omega(a_i), \dots, \omega(a_j) = \gamma(op)$ , since all input data are read exactly when  $op$  starts.

## 5.2 Definition of Border PE

A crucial parameter for the interface design is the number of BPEs in the array. We formally define the subsets  $\mathcal{BPE}_{in}$  and  $\mathcal{BPE}_{out}$  of the processor space  $\mathcal{P}$ .  $\mathcal{BPE}_{in}$  (input border processor elements) is the set of PE where index vectors  $I \in \mathbf{I}$  are mapped to that have a data dependency to another index vector  $J \notin \mathbf{I}$  outside the index space. Let  $\mathcal{D}(I)$  the set of dependence vectors of all indexed equations computed at index point  $I$ .

$$\begin{aligned} \mathcal{BPE}_{in} = \{ & p \in \mathcal{P} \mid \exists I \in \mathbf{I}, \exists t \in \mathbb{N} : (pt)^T = TI \\ & \wedge \exists d \in \mathcal{D}(I) : I - d = J, J \notin \mathbf{I} \} \end{aligned}$$

$T$  is the linear mapping as defined in Section 5.1. The definition of  $\mathcal{BPE}_{out}$  (output border processor elements) is more complicated since for the array outputs no dependence vectors exist. However,  $\mathcal{BPE}_{out}$  is not considered here and we use  $\mathcal{BPE}$  and  $\mathcal{BPE}_{in}$  synonymously.

According to this definition, the BPEs do not necessarily need to be located at the border of the processor array. Depending on the index space  $\mathbf{I}$  and the mapping  $T$ , any PE in the array may be a BPE resulting in an irregular wiring as well as a large number of array inputs. This should be avoided by appropriate index space transformations to  $\mathbf{I}$  before mapping. Here, we assume all BPEs are located at the array borders:

$$\forall p \in \mathcal{BPE} : \exists e \in E : p + e \notin \mathcal{P} \vee p - e \notin \mathcal{P} \quad (2)$$

where  $E$  is the set of unit vectors (i.e.  $E = \{(0 \ 1)^T, (1 \ 0)^T\}$  for a processor array of dimension  $d = 2$ ).

## 5.3 Bandwidth Bounds

Without loss of generality, we make the following assumptions for the rest of the paper:

- We have only one type of BPE, i.e., all BPEs have the same inputs (which is the case for homogeneous arrays)
- The *first* iteration intervals of all BPEs overlap, i.e. all BPEs start the first computation within one iteration interval
- All array inputs have the same bit width

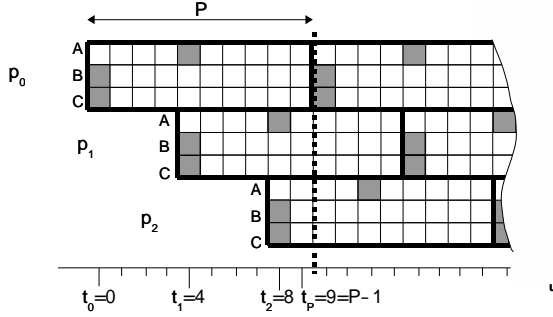


Figure 3: First iteration intervals of three BPEs.

**Example 5.1** Consider Fig. 3. We have  $\mathcal{BPE} = \{p_0, p_1, p_2\}$ . Each BPE has the inputs A, B, and C. The grey boxes mark the  $\omega$ -steps when the inputs are read:  $\omega(B) = \omega(C) = 0, \omega(A) = 4$ . The length of the iteration period is  $P = 10$ .

At  $t = t_1$ , three data words are read in parallel, while in other time steps no inputs are read. This results in an unbalanced interface bandwidth with peaks at certain time steps.

The DPRAM output bus width  $b$  must be sufficient to serve the bandwidth peaks (i.e. 3 data words in Example 5.1:  $b_{max} = 3$ ). Our goal is to avoid the bandwidth peaks by distributing the  $\omega$ -steps uniformly over the whole iteration period, which consequently facilitates a reduction of the DPRAM output bus width.

Let  $B = |\mathcal{BPE}|$  the number of BPEs and  $k$  the number of inputs of each BPE. Each input is read once per iteration period  $P$  which leads to a minimum bus width of

$$b_{min} = \lceil \frac{kB}{P} \rceil \quad (3)$$

data words which is achieved when the  $\omega$ -steps are distributed uniformly over  $P$ . In Example 5.1 we have  $b_{min} = 1$ .

## 5.4 Definitions

In this section, we define functions and matrices that are used for the optimization algorithm given in Section 5.5.

We define a function  $\tau : \mathcal{BPE} \mapsto \mathbb{N}$  where  $\tau(p)$  is the time step when the first iteration period starts on  $p$ :

$$\tau(p) = \min\{t \in \mathbb{N} | \exists I \in \mathbf{I} : TI = (p, t)^T\} \quad (4)$$

$T$  is again the mapping function. We sort all BPEs in ascending order with respect to  $\tau$ :

$$\begin{aligned} \forall p_i, p_j \in \mathcal{BPE} : \\ i < j \Rightarrow \tau(p_i) \leq \tau(p_j), \quad 0 \leq i, j \leq B-1 \end{aligned}$$

with  $B = |\mathcal{BPE}|$ . Without loss of generality we assume  $\tau(p_0) = 0$ . If  $k$  is the number of the inputs of each BPE, we arrange the inputs  $a_0, a_1, \dots, a_{k-1}$  of a BPE in an arbitrary but fixed order.

Now we define a function  $\tilde{\omega} : \mathbb{N}^2 \mapsto \mathbb{N}$ , where  $\tilde{\omega}(i, j)$  is the time step when input  $a_j$  of BPE  $p_i$  is read, modulo the iteration period  $P$ :

$$\tilde{\omega}(i, j) = (\omega(a_j) + \tau(p_i)) \bmod P \quad (5)$$

Note that  $\tilde{\omega}$  denotes absolute time steps and not time steps from the beginning of an iteration period as  $\omega$  does.

With  $\tilde{\omega}$  we construct a matrix  $\Delta = (\delta_{i,j}) \in \{0, 1\}^{P \times kB}$  which is the basic structure for our bandwidth optimization:

$$\delta_{i,j} = \begin{cases} 1 & : j = \tilde{\omega}(\lfloor \frac{i}{k} \rfloor, i \bmod k) \\ 0 & : \text{else} \end{cases}$$

$$0 \leq i < kP, \quad 0 \leq j < P$$

Each column in  $\Delta$  denotes a time step in the iteration period of  $p_0$  and each line represents one input of one BPE. (Note that the indexing of the matrix elements starts at  $(0, 0)$ ).

**Example 5.2** Consider again Example 5.1. We have  $\tau(p_0) = 0$ ,  $\tau(p_1) = 4$ , and  $\tau(p_2) = 8$ . We sort the PE inputs as  $a_0 = A$ ,  $a_1 = B$ , and  $a_2 = C$ . Eq. 5 leads to  $\tilde{\omega}(0, 0) = 4$ ,  $\tilde{\omega}(0, 1) = \tilde{\omega}(0, 2) = 0$ ,  $\tilde{\omega}(1, 0) = 8$ ,  $\tilde{\omega}(1, 1) = \tilde{\omega}(1, 2) = 4$ ,  $\tilde{\omega}(2, 0) = 2$ , and  $\tilde{\omega}(2, 1) = \tilde{\omega}(2, 2) = 8$ .

The  $\Delta$ -Matrix is

$$\Delta = \begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

$\Delta$  can be derived from Figure 3 by wrapping around at the dashed line ( $t = P - 1$ ) and writing a '1' for the grey boxes.

The 1's in  $\Delta$  mark the time steps when the array inputs are read.

For minimizing the DPRAM bus width, we use additional buffer registers. On the other hand, buffer registers are not required on every array input since the DPRAM output latches can also be used for buffering. Totally  $kB - b_{min}$  additional buffers are required at most. All additional registers must be loaded not later than one clock cycle before read, i.e., the DPRAM output data must be available one clock cycle earlier except for those array inputs that use the DPRAM latches for buffering. This is reflected in  $\Delta$  by selecting a column  $j$  with at least  $b_{min}$  1's and tagging  $b_{min}$  lines that have a 1 in column  $j$ . All other line vectors must be rotated left by one. We formally define the line rotation as follows:

$$rot(\Delta, l, d) = \tilde{\Delta}, \quad \tilde{\delta}_{i,j} = \begin{cases} \delta_{i,(j+d) \bmod P} & : i = l \\ \delta_{i,j} & : i \neq l \end{cases}$$

$$0 \leq i < kP, \quad 0 \leq j < P$$

The function  $rot(\Delta, l, d)$  provides a matrix  $\tilde{\Delta}$  which is equal to  $\Delta$  but with line  $l$  rotated to left by  $d$ .

**Example 5.3** (Consider Example 5.2).  $b_{min} = 1$ . We tag line 0 and transform all other lines by  $rot(\Delta, i, 1) (i = 1, \dots, kP - 1)$ :

$$\Delta = \begin{pmatrix} (*) & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 \end{pmatrix}$$

The tag (\*) in line 0 denotes that no additional input buffer is used at input  $a_0$  of  $p_0$  (because the DPRAM output latch is used here). The 1's in the other lines denote that the content of the corresponding input buffer will be used in the next time step.

## 5.5 Optimization Algorithm

The 1's in  $\Delta$  represent the *latest* time steps when the buffer registers must be loaded. Bandwidth peaks occur in these time steps where the sum of elements in the corresponding  $\Delta$ -column is higher than  $b_{min}$  (columns 3, 7, and 9 in Example 5.3).

The following algorithm avoids the bandwidth peaks by shifting the time steps where the buffers are loaded to earlier time steps. The idea is to create another Matrix  $\Delta^* \in \{0, 1\}^{P \times kB}$  that assigns each input buffer a new load-time step such that

$$\Sigma(\Delta^*, i) \leq b_{min} \quad \forall i = 0, \dots, P - 1$$

where  $\Sigma(\Delta^*, i)$  is the sum of the elements of column  $i$  of matrix  $\Delta^*$ . Each 1 in  $\Delta^*$  denotes a time step when a buffer is loaded and each 1 in  $\Delta$  denotes the last time step the buffer is used. Thus,  $\Delta$  and  $\Delta^*$  define an allocation interval for each buffer. Our algorithm keeps these intervals as short as possible in order to allow for resource sharing of buffers with non-overlapping allocation intervals.

In the following algorithm,  $update\_Sigma()$  computes  $\Sigma(\Delta^*, j)$  for all  $j = 0, \dots, P - 1$  and  $tagged(i)$  returns true iff line  $i$  of  $\Delta$  is tagged.

```

1   $\Delta^* = \Delta;$ 
2   $update\_Sigma();$ 
3  for  $j = P - 1$  downto 0
4  {  $k=0;$ 
5    while  $(\Sigma(\Delta^*, j) > b_{min})$ 
6      { while  $(\Sigma(\Delta^*, t) \geq b_{min})$   $t = (t - 1) \bmod P;$ 
7        while  $(\delta_{k,j}^* = 0 \vee tagged(k))$   $k = k + 1;$ 
8           $\delta_{k,t}^* = 1;$   $\delta_{k,j}^* = 0;$ 
9           $\Sigma(\Delta^*, j) = \Sigma(\Delta^*, j) - 1;$   $\Sigma(t) = \Sigma(t) + 1;$ 
10       }
11  }
```

Figure 4: Algorithm for computation of  $\Delta^*$

The for loop in line 3 processes all columns  $j$  of  $\Delta^*$ . While  $\Sigma(\Delta^*, j)$  is greater than  $b_{min}$ , in line 6 another column  $t$  is searched with  $\Sigma(\Delta^*, t)$  smaller than  $b_{min}$ . Then, in line 7, a non-tagged line vector  $k$  of  $\Delta$  is chosen that has a 1 in column  $j$ . This 1 is moved to column  $t$  (line 8) and  $\Sigma(\Delta^*, j)$  and  $\Sigma(\Delta^*, t)$  are updated (line 9). The loop is repeated until  $\Sigma(\Delta^*, j)$  is less or equal to  $b_{min}$ .

It should be noted that all operations concerning the column index are performed modulo  $P$ . This is admissible since  $\Delta^*$  describes a cyclic schedule and the time step when a register is loaded can be moved into the previous iteration period.

**Example 5.4** The following matrix results from the algorithm in Fig. 4 is applied to the matrix in

Example 5.3:

$$\Delta^* = \begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

Obviously, each sum of column elements is less or equal to  $b_{min} = 1$ . Line 0 remained unchanged since it was tagged in the original matrix  $\Delta$ .

## 5.6 Buffer Register Minimization

As shown in Section 5.4, the number of additional buffer registers can be decreased by exactly  $b_{min}$  if the DPRAM output latches are used for buffering. The number of buffer registers can further be optimized if we find non-overlapping allocation intervals for the buffers.

The non-tagged lines of  $\Delta$  and  $\Delta^*$  define allocation intervals of the buffer registers. We define a set  $V = \{I_0, I_1, \dots, I_{kB-b_{min}-1}\}$  that contains all  $kB - b_{min}$  allocation intervals. Furthermore, we define the sets  $Z_0, Z_1, \dots, Z_n (n \leq kB - b_{min})$  where each set  $Z_i$  contains the intervals that are disjunct and thus can be mapped to the same buffer register. The algorithm given in Fig. 5 generates the  $Z_i$ -sets.

```

1  k = 0;
2  while V ≠ ∅
3  {  Zk = ∅;
4     choose Ii ∈ V;
5     V = V - {Ii};
6     Zk = Zk ∪ {Ii};
7     for all Ij ∈ V
8     if Ii ∩ Ij = ∅
9     { V = V - {Ij};
10    Zk = Zk ∪ {Ij};
11    }
12  k = k + 1;
13 }
```

Figure 5: Algorithm for computation of the sets  $Z_i$

The matrices  $\Delta$  and  $\Delta^*$  and the  $Z$ -sets are now used for constructing a controller that generates the write-enable signals for all input buffers as well as the read addresses for the DPRAM.

## 6 Conclusion

VLSI processor arrays require a large interface bandwidth which often cannot be provided under hard resource constraints. We have presented a generic modular interface architecture for FPGA-implementations of regular processor arrays. For different types of array ports we have proposed an interface structure that matches the bandwidth requirements of this port type. Moreover, we have shown how the interface architecture can be optimized with respect to the required bus width and the number of buffer registers.

The proposed architecture is intended to be generated as part of our automated design flow for FPGA based VLSI processor arrays.

## References

- [1] M. Bednara and J. Teich. Synthesis of FPGA Implementations From Loop Algorithms. In *Proceedings of the Int. Conf. on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, June 2001.
- [2] U. Eckhardt. *Algorithmus-Architektur-Codesign für den Entwurf digitaler Systeme mit eingebettetem Prozessorarray und Speicherhierarchie*. PhD thesis, Fakultät Elektrotechnik, Technische Universität Dresden, Deutschland, 2001.
- [3] M.K. Stojcev, G. Djordjevic, E. Milovanovic, and I. Milovanovic. Data reordering converter: an interface block in a linear chain of processing arrays. *Microelectronics Journal*, (31):23–37, 2000.
- [4] J. Teich. *A Compiler for Application-Specific Processor Arrays*. PhD thesis, Institut für Mikroelektronik, Universität des Saarlandes, Saarbrücken, Deutschland, September 1993.
- [5] J. Teich, L. Thiele, and L. Zhang. Partitioning processor arrays under resource constraints. *Int. Journal of VLSI Signal Processing*, 17(1):5–20, September 1997.
- [6] L. Thiele. Scheduling of uniform algorithms with resource constraints. *Journal of VLSI Signal Processing*, 10:295–310, 1995.
- [7] Xilinx. Virtex-II 1.5V FPGAs, Detailed Description, <http://www.xilinx.com/partinfo/ds031-2.pdf>. Technical report, Xilinx, Inc., 2001.
- [8] Xilinx. Xilinx Application Note XAPP-230, The LVDS I/O Standard, <http://www.xilinx.com/xapp/xapp230.pdf>. Technical report, Xilinx, Inc., 2001.
- [9] Xilinx. Xilinx Application Note XAPP-233, Multi-Channel 622 Mb/s LVDS Data Transfer for Virtex-E Devices, <http://www.xilinx.com/xapp/xapp233.pdf>. Technical report, Xilinx, Inc., 2001.
- [10] Xilinx. Xilinx Application Note XAPP-256, FIFOs Using Virtex-II Shift Registers, <http://www.xilinx.com/xapp/xapp256.pdf>. Technical report, Xilinx, Inc., 2001.