6. KressArray Design Space

The KressArray design space exploration is aimed to assist the designer in finding a suitable architecture for a given set of applications. The target architecture of the exploration process is a member of the KressArray family. The first KressArray architecture was the historic KressArray-I [Kres96]. However, the Xplorer system allows to explore much more advanced architectures. In the following, the term KressArray denotes a member of the KressArray family.

In this section, a novel design space for the KressArray family will be described, which resembles the base for the exploration process. In the KressArray Xplorer framework, the architectural properties are specified in the hardware file for the ALE-X compiler (see section 7.1) and in the intermediate file (see sectio nA.1).

The properties determining the design space can be classified into the following groups:

- rDPU properties. These are aspects applying to a single rDPU (reconfigurable datapath unit) cell. A rDPU is the basic processing element of the KressArray. An example for a rDPU property is the available operator set.
- Array properties. These are aspects of the whole array of rDPUs, like the size or the external memory connection.
- Communication resources. The type and number of the available resources in the communication network are the main topic of the exploration process.

In the following subsections, first a brief review of the basic KressArray principles will be given, identifying the basic aspects common to all members of the KressArray architecture family. Then, the groups of KressArray design aspects and the properties in each group will be discussed. After that, the classification of KressArray architectures given by these design aspects will be illustrated using two KressArray prototypes as examples.

6.1 KressArray Principles

A member of the KressArray architecture family is a rectangular array of coarse grained rDPU (reconfigurable Datapath Unit) cells connected by a communication network for the transport of data words between the cells. The communication network consists of nearest neighbor connections, backbuses connecting rows or columns of rDPUs and one global serial bus connecting all

rDPUs. An example KressArray with 16 rDPUs, one horizontal and two vertical bidirectional nearest neighbor connections, one column backbus spanning the whole array, and the global serial bus is shown in figur e6-1a.

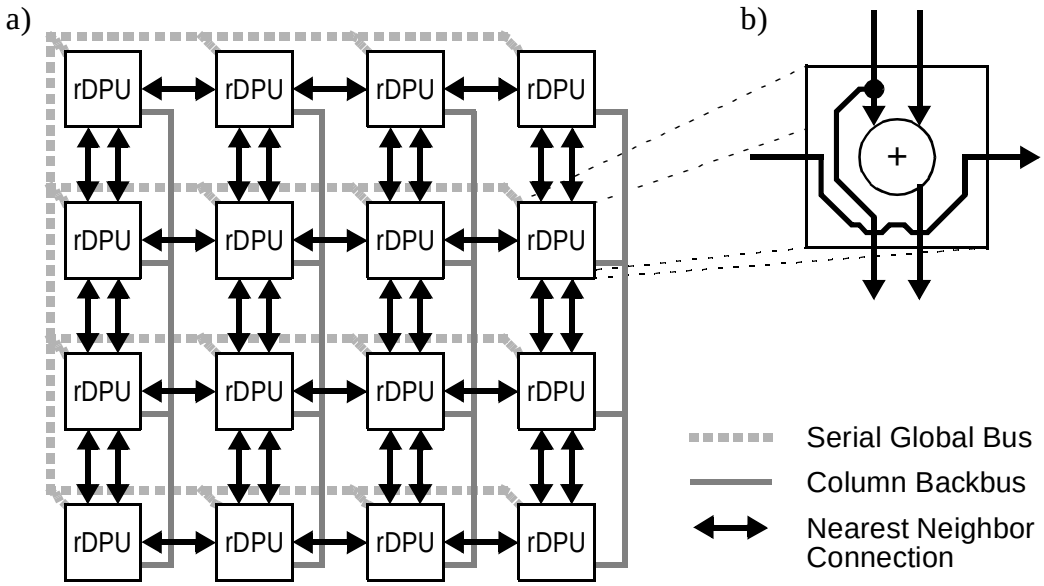

Figure 6-1: Basic KressArray architecture
a) array structure
b) rDPU operator and routing

The rDPUs can be used to implement computational operators, to route data words to other rDPUs, or for concurrent operation and routing. The latter situation is illustrated by an example in figure 6-1b. The addition operator gets operands from the two northern connections. The result is output to one of the southern links. Additionally, one of the operands is forwarded to the south, and another data word not involved in the operation is routed from west to east. For synchronization, the rDPUs provide registers at each operator input and also at the operator output. These registers are used to implement the transport-triggered execution model (see below in section 6.1.2).

VLSI Implementation . The envisioned realization of KressArrays on silicon is based on the wiring by abutment design style. The rDPUs are assumed to be present as tiles, with all connections, including the communication resources, abutting at the edges of the tiles. Thus, an array can be easily layouted by placing the desired number of rDPU tiles next to each other. At the array margins, special tiles will be needed to realize features spanning cells, like e.g. the global serial

bus, or segmented backbuses. However, parts of the additional wiring may be implemented outside of the array itself, so that the architecture itself resembles a regular layout.

In the following sections, the construction of datapaths and the execution model are briefly sketched. More details about these issues can be found in [Kres96]. Then, a new aspect of the KressArray design space regarding the implementation of datapaths with cycles is described. Afterwards, the usage of datapath cycles for the implementation of loops is described, with a novel synchronization technique based on special operators.

6.1.1 Datapath Construction

A datapath for processing data streams consists of several operators with certain mutual dependencies. A graphic representation of such a structure is a dataflow graph. It is derived from a procedural description of the datapath, which may contain assignments, conditional expressions, and loops. The implementation of these constructs follows the concepts of the historical KressArray-I and are thus only briefly sketched. For more details, refer to [Kres96]. In this and the following sections, several operators will be used resembling special functions. For better understanding, an overview of these operators used in datapaths is given in table 6-1.

While assignments can be transformed directly into a dataflow graph, conditional expressions containing an if-clause need to be transformed into functional if-expressions. The functional if-operator is also known as the select operator or the "?:"-operator in the programming language C. An expression with this operator has the syntax:

Z = *Cond* **?** *X* **:** *Y* ;

With the following semantic: If *Cond* is true, then *Z* results to *X*, otherwise, *Z* results to *Y*. As far as datapaths are concerned, this operator can be used to transform if-clauses. A small example datapath showing assignments and an if-clause is shown in figure 6-2. The procedural description and transformation is shown in figure 6-2a, while the according dataflow graph is shown in figure 6-2b. Note, that due to the parallel and transport-triggered execution, no conflict occurs, as the values are exchanged on two spatially separated paths.

For the implementation of loops, one possible method uses a control unit to feed back the operators (see [Kres96]). This technique is further supported. An alternative way to implement such loops are cycles in the datapath. This issue is discussed below in section 6.1.3.

a)

```
IF ( x > y )
{     x = y ;
      y = x ;
}
```

is transformed into:

```
x = ( x > y ) ? y : x ;
y = ( x > y ) ? x : y ;
```

b)

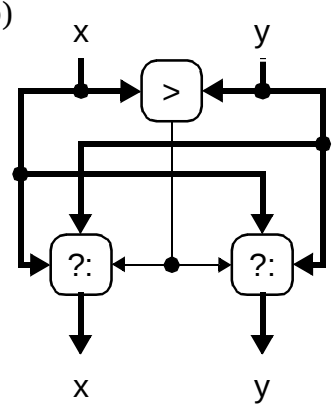

Figure 6-2: Example datapath (sort by conditional swap) with assignments and if-clause:
a) Procedural description and transformation
b) Dataflow graph

In order to implement the datapath on the KressArray, the operators have to be assigned to rDPUs and the connections to available communication resources by a placement and routing step (compare section 3.4 and section 3.5). A valid placement and routing is also referred to as a mapping. An example mapping for the datapath in figure 6-2 is shown in figure 6-3.

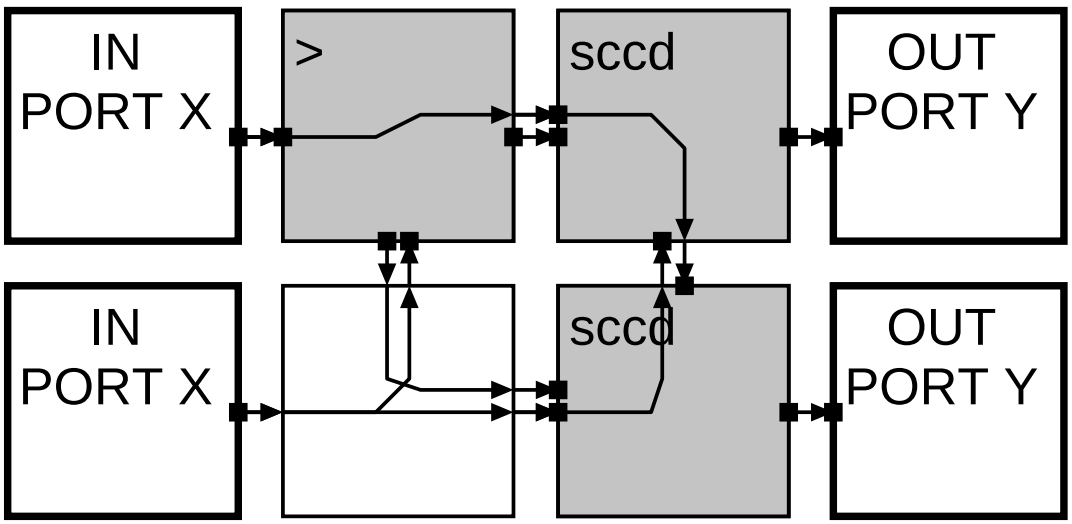

Figure 6-3: Mapping of the conditional swap datapath on example KressArray architecture

Operator	Inputs	Outputs	Description
+, −, *,...	A, B	X	Standard arithmetic operators +,-,*,... X=A+B, X=A-B, X=A*B,...
&, |,...	A, B	X	Standard logic operators & (AND), | (OR),... X=A&B, X=A|B,...
<, ==, >,...	A, B	Cond	Standard comparisons <, ==, >,... **IF** (A<B) **{** Cond=1; **} ELSE {** Cond=0; **}** (Accordingly for ==, >,...)
<c, =c, >c,...	A	Cond	Standard comparisons with constant c **IF** (A<c) **{** Cond=1; **} ELSE {** Cond=0; **}** (Accordingly for =c, >c,...)
<<, >>	A, B	X	Shift left (<<) or right (>>) by V bits
<<c, >>c	A	X	Shift left (<<) or right (>>) by c bits
LSB	A	Cond	Cond=(Least significant bit of A)
EN	A, Cond	X	Enable operator **IF** Cond **{** X=A; **} ELSE {** X=0; **}**
HD	A, B	X	Hamming distance X = number of differing bits of A, B
?:	A, B, Cond	X	Select operator **IF** Cond **{** X=A; **} ELSE {** X=B; **}**
{	A, B, Cond	X	Loop-start operator **IF** Cond **{** X=A; **} ELSE {** X=B; **}** If Cond is true, B is not consumed. (see section 6.1.4.1 for details)
}	A, Cond	X	Loop-end operator **IF** Cond **{** NO_OUTPUT **} ELSE {** X=A; **}** (see section 6.1.4.2 for details)
"c" denotes a constant value appearing in the operator name "A", "B", "X" represent numeric data values "Cond" represents a boolean condition value			

Table 6-1: Table of operators used for datapath construction in this section

6.1.2 Execution Model

The operators are meant to be arithmetic and logic functions working on data words rather than bit-level logic functions. A datapath consisting of a structure of connected operators follows a transport-triggered execution model. This means, that the operation starts as soon as all input operands are available at the input registers of the rDPU and the output register is not occupied. The output register is free, if it has transferred its contents to all following rDPUs in the datapath. More details about the implementation of this mechanism can be found in [Kres96].

6.1.3 Datapaths with Cycles

While a datapath is supposed to be a directed acyclic graph (DAG), there are cases, where cycles in the graph are desirable. These cases include iterative algorithms which feed back data words, e.g. the viterbi algorithm known from signal processing. However, the rDPUs have to have additional capabilities to implement such cycles. A very simple example for an operator with a feedback is the integration operator shown in figure 6-4a, which will be used to illustrate the implementation of datapath cycles.

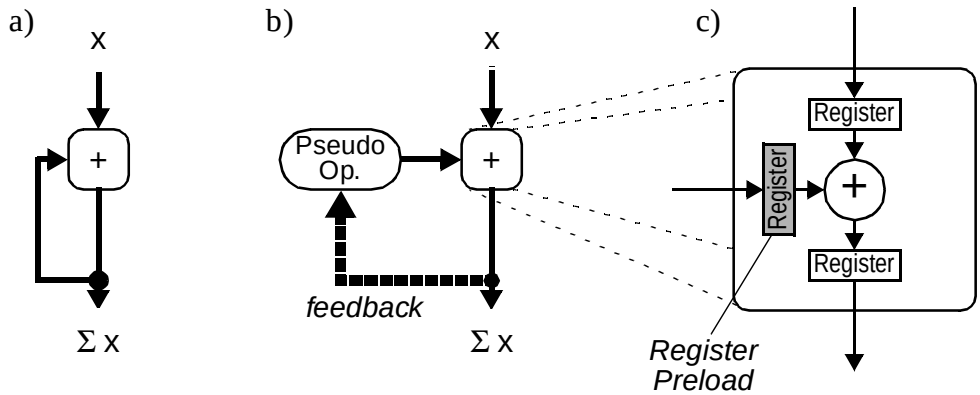

Figure 6-4: Example for datapath with a cycle:
a) Integration operator
b) Implementation with pseudo operation
c) Input register preload

The integration operator consists of a single adder, which feeds back its result to one input. Thus, the integrator provides as output a data stream resembling the current sum of all previous input values. The direct implementation as shown in figure 6-4a is inconsistent with the execution model. As the addition will not operate until both input operands are available, it will never start to work as the operand, which is fed back, is not available at the according input register on

system start. Thus, the cycle has to be described in an explicit way to enable special treatment. This is done by introducing a pseudo operator at the relevant input of the adder, as shown in figure 6-4b. This pseudo operator has the reference of the adder output as a special attribute, which is not used until the placement and routing phase. Thus, the datapath remains virtually acyclic, until the pseudo operator reference is finally resolved to implement the cycle in the mapping. When the configuration code is generated, the adder rDPU has to be configured to preload the input register connected to the pseudo operation with a value to enable proper execution (see figure 6-4b). The preloading is done at system start-up and reset. The value to be preloaded is dependent on the application and is specified together with the pseudo operation. In the example, a value of 0 has to be preloaded.

6.1.4 Implementation of Loops

The loops referred to in this section are controlled by a condition and allow a certain set of operations to be repeatedly executed until this condition is met. However, loops imply additional requirements for the architecture, as well as some restrictions. Either a control unit or two special kinds of operators are needed to implement loops. Furthermore, as the number of loop iterations is not known in advance, dynamic communication takes place, which forbids a static schedule. This affects e.g. the implementation of row or column backbuses (see below in section 6.3.3).

There are two kinds of loops supported: while-loops and do-while-loops. One possible approach implements loops by using a control unit, which handles every single loop iteration. A detailed description of this mechanism is given in [Kres96]. Besides this technique, which is also supported, loops can be implemented using cycles in the datapath (see section 6.1.3 above). However, this requires the existence of two special operators called loop-start ("{") and loop-end ("}") for synchronization. One pair of these operators is necessary for each variable existing in the loop body. Furthermore, loops constructed using the technique described in the following do not allow pipelining. The general structure of a loop datapath including these operators is shown in figure 6-5 for a while loop and in figure 6-6 for a do-while loop respectively. The cycles are supposed to be implemented according to the technique described in section 6.1.3.

As mentioned above, the loop-start and loop-end operators are used to synchronize the data rate inside the loop with the data rate outside. Both operators are controlled by the loop condition. The loop is iterated as long as this condition is "True". For every set of values entering the loop, one set of values will

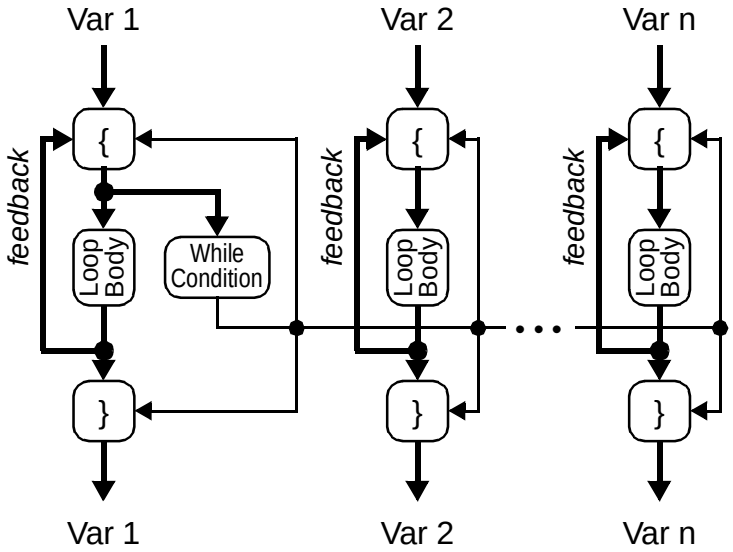

Figure 6-5: General structure of a while loop implemented using datapath cycles and loop operators

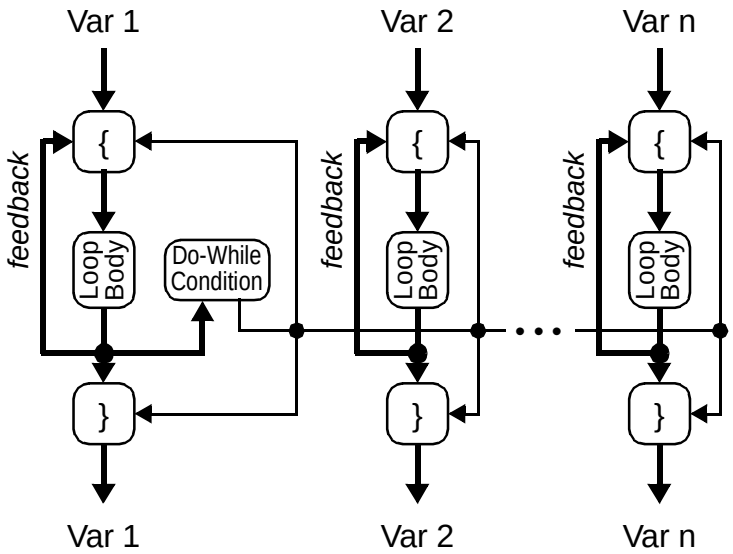

Figure 6-6: General structure of a do-while loop implemented using datapath cycles and loop operators

also emerge at the end. However, inside the loop, an arbitrary number of iterations will happen. In the following, the function of the loop-start and loop-end operators will be defined.

6.1.4.1 Loop-Start Operator

The function of the loop-start operator is similar to the select-operator ("?:") described in section 6.1.1. There are a feedback input, a loop input, a condition input, and an output. If a "False" condition signal arrives at the condition input, the operator will forward the data word at the loop input to the output, if a "True" signal arrives, the feedback input will be switched through. The difference to the select-operator lies in the trigger condition. The operator will forward the loop feedback values even if no data word at the loop input is present. Also, if the feedback input is forwarded and there happens to be a value at the loop input, this value is not discarded. On the other hand, the loop input is only forwarded if there is also a value at the feedback input. As the operator is put in a cycle, both the feedback input and the condition input have to be preloaded to enable the loop to be started. The condition input has to be initialized to "False", while the feedback input may contain any value. The structure of the loop-start operator is shown in figure 6-7a.

6.1.4.2 Loop-End Operator

While the loop-start operator is needed for every variable entering the loop, the loop-end operator is only needed if the variable is leaving the loop. The operator features a loop input, a condition input and an output. The task of the loop-end operator is the identification of the final result of the according variable and its forwarding to the output at the end of the loop. The end of the loop is reached, when a "False" signal arrives at the condition input. Then, the value at the loop input is forwarded to the output. During the loop, a "True" signal and an intermediate result will arrive in each iteration at the condition and the loop input respectively. In this cases, both the "True" signal and the data word are consumed without generating an output data, thus discarding the intermediate value. The structure of the loop-end operator is shown in figur e6-7b.

In the next section, the implementation of the two types of loops supported by the exploration framework will be described using illustrative examples.

6.1.4.3 While Loops

In a while loop, the test for the loop condition happens at the beginning of the loop. Thus, the according test operator must be placed at the beginning of the loop datapath right after the loop-start operator. A special treatment must be taken for variables being output from the loop, as it might be, that the loop body is not executed at all. To handle this case, the loop datapath must be bypassed to enable the original, unchanged variable to be output. Thus, an additional select operator is needed at the end of the loop. For illustration of the concepts described here, an example is given in figure 6-8.

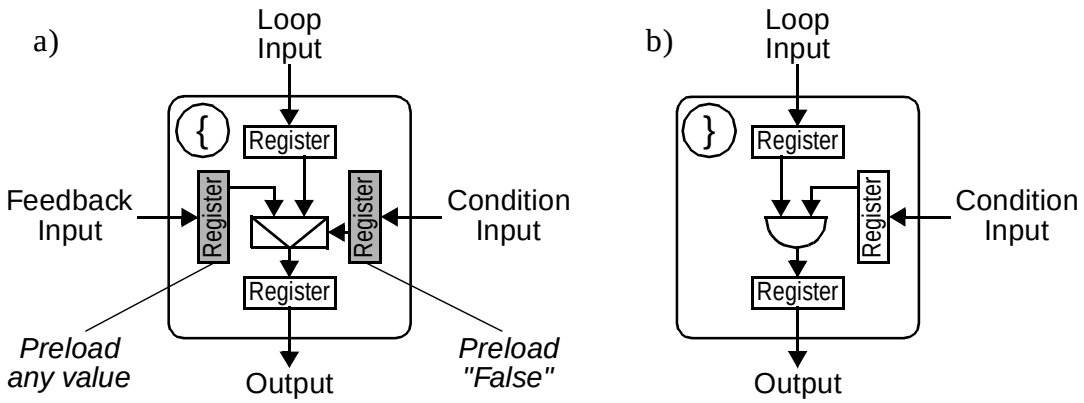

Figure 6-7: Special operators for loops:
a) Loop-start operator (see section 6.1.4.1)
b) Loop-end operator (see section 6.1.4.2)

a)

```
INPUT x, y ;
WHILE ( x > 0 )
{     IF ( x < y )
      {     swap( x, y ) ;
      }
      x = x - y ;
}
OUTPUT y ;
```

b)

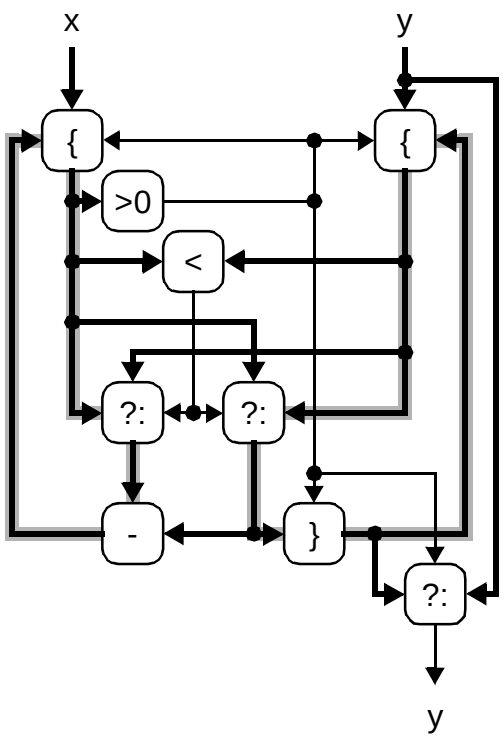

Figure 6-8: Example for a while loop:
a) Algorithm description in pseudo-code
b) Dataflow graph

The example shows an implementation of the Euclidean algorithm for the determination of the greatest common divisor (GCD) of two numbers. The algorithm in pseudo-code is described in figur e6-8a, and the according dataflow graph in figure 6-8b. The cycles formed by the variable datapaths are shaded for

illustrational purposes. Another cycle is formed by the loop condition (output of ">0"). As the value *x* is not output, no loop-end operator is needed. Whereas for *y* both a loop-end operator and an additional select operator at the end is needed to implement the loop.

6.1.4.4 Do-While Loops

In a do-while loop, the test for the loop condition is at the end of the datapath. Thus, the loop is executed at least once. This has an advantageous effect on the implementation according to the described mechanism, as the final select operator is not needed anymore. The example algorithm of figur e6-8 with a do-while loop instead of a while loop is shown in figure 6-9, with the algorithm in figure 6-9a and the dataflow graph in figur e6-9b.

a)

```
INPUT x, y ;
DO
{     IF ( x < y )
      {     swap( x, y ) ;
      }
      x = x - y ;
} WHILE ( x > 0 )
OUTPUT y ;
```

b)

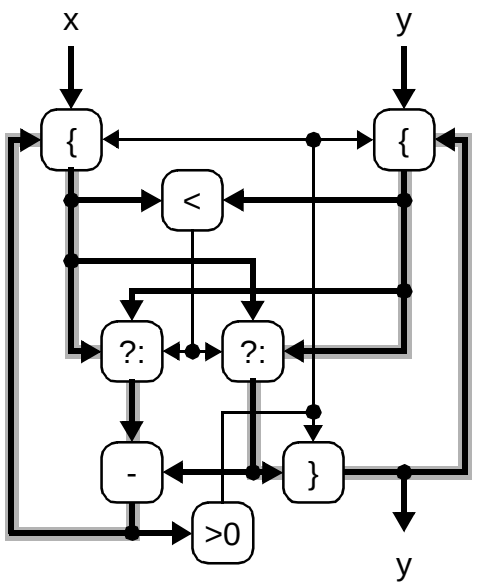

Figure 6-9: Example for a do-while loop:
a) Algorithm description in pseudo-code
b) Dataflow graph

6.2 rDPU Properties

The design aspects of a single rDPU cell are mainly related to the operator set of the rDPU. The properties are the set of available operators themselves and their bitwidth, which determines also the width of the communication structures. Also included in this section is a description of two special features, which may be appropriate for some applications or architectures: operators with double outputs and support for double-ALU rDPUs.

6.2.1 Operator Set

The operator set is described in a separate hardware file, which is read by the ALE-X compiler [HaKr95]. The designer is supposed to describe additional functions relevant for the applications to explore in this hardware file. It may be reasonable to explore different operator sets, for which the designer has to adapt the datapath of the applications. To illustrate the motivation for different operator sets, a simple example will be discussed in the following.

The example in figure 6-10 shows two alternative implementations of a multiplication operator. For readability, the implementation in operator set 2 is shown as a dataflow graph rather than as a mapping. While operator set 1 uses a single rDPU for the multiplication, operator set 2 employs a network of operators to implement a 4 by 4 bit multiplier, using only adds, shifts, and multiplexers, with the "EN" operator having the function described in tabl e6-1. In this case, the operator set affects the complexity of the rDPU cells.

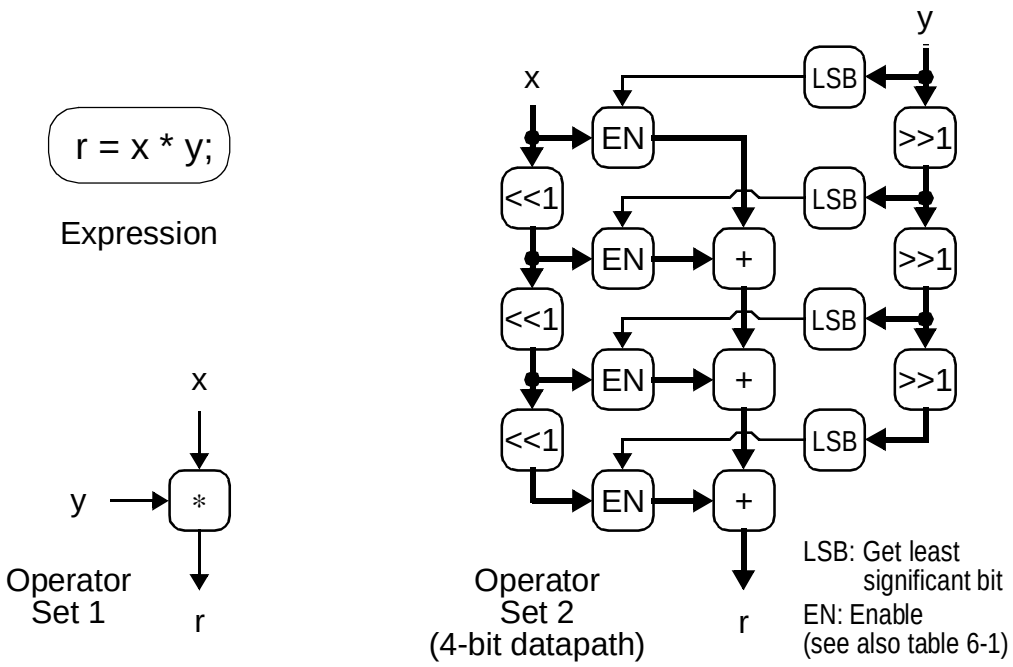

Figure 6-10: Example for different operator sets affecting rDPU cell complexity

6.2.2 Bitwidth

The bitwidth of the operators determines also the width of the routing resources and the chip area of the whole design. For the design framework, the bitwidth is of interest only for external tools, as the synthesis subsystem abstracts from it. The bitwidth is specified in the according *BitWidth* parameter of the intermediate file.

6.2.3 Double ALU rDPUs

For special purposes, it may be desirable to have two ALUs in one rDPU, which may work together to implement a certain complex operator. To implement other operators, a single ALU might be sufficient. Thus, one rDPU may implement one more complex operator or several less complex operators. The exploration framework supports rDPUs with up to two ALUs.

The specification of double-ALU rDPUs is implemented by an operator attribute, which describes how a specific operator may be paired with other operators. This attribute is implemented as a bitmask. Two operators are allowed to share one rDPU if a logic AND of their bitmasks results in a non-zero value. Typically, the operators would be ordered in groups, with the group members being able to share a rDPU. This can be done by assigning a bit position of the mask to each group. The bit positions may be chosen by the designer in an arbitrary way except the rightmost one (least significant bit). This bit is treated separately and denotes, if the operator may be paired with another operator of the same type.

This approach will be illustrated by a simple example. Given an operator set consisting of addition ("+"), logic AND ("&"), logic OR ("|") and multiplication ("*"), the following situation is assumed: An rDPU contains two ALUs, each of which can implement addition and AND. Additionally, both ALUs can be combined to form one serial multiplication operator, which computes two result bits in one time step. However, it is assumed not to be possible that two independent additions be implemented at one time. Thus, for the pairing of operators, the following observations apply:

- A multiplication can be paired neither with another multiplication, nor with an addition or a logic operation.
- An addition can be paired with a logic operation, but neither with another addition, nor with a multiplication.
- A logic operation can be paired with an addition or any other logic operation, including one of the same type.

This relationship between the operators and a possible choice for bitmask values is shown in table 6-2. Two bits, including the special least significant bit, are necessary to implement the pairing relations of the operators. The value "00" for the multiplication will result in a zero value when ANDed with any other

bitmask, thus preventing any pairing. The second bit from right identifies a group containing the other three operators. A logic AND of the bitmasks of any two members of this group will result in a value of "10" or "11", thus allowing pairing. An exception is the addition, which has the least significant bit cleared, preventing it from being paired with another addition. However, an addition may join any logic operator, and the logic operators themselves may join any other logic operators, including one of the same type, as the least significant bit is set. The bitmasks for operators are described in the intermediate file.

Operator	Pair with "*"	Pair with "+"	Pair with "&"	Pair with "|"	Example Bitmask
Multiplication ("*")	No	No	No	No	0 0
Addition ("+")	No	No	Yes	Yes	1 0
Logic AND ("&")	No	Yes	Yes	Yes	1 1
Logic OR ("|")	No	Yes	Yes	Yes	1 1

Table 6-2: Example for operator bitmasks in Double-ALU rDPUs

6.2.4 Double Output Operators

In addition to normal operators bearing one single output, there are other functions possible producing two results. An example for this concept could be a division operator. As the division algorithm often produces both the division result and the rest (modulo operation), a double output rDPU may thus produce the division result at one output and the rest of the division at the other one. Such double output operators are also described in the intermediate file.

6.3 Communication Resources

The communication resources resemble the means for data transfer between rDPUs. They are a part of the datapath. The number, bitwidth, and type of the resources have a large impact on the area of a single rDPU cell. In the KressArray design space, three communication resources exist: A serial global bus, nearest neighbor connections, and backbuses connecting rows or columns of rDPUs.

6.3.1 The Global Bus

The concept of the serial global bus has been adopted from the historic KressArray-I. This bus connects every rDPU with every other one, and thus allows data transfers between arbitrary locations of the array. However, as the bus is capable of only one transfer at a time, the communication has to be done in a serial fashion. Thus, the global bus is likely to become a performance bottleneck. Because of this, using the global bus is normally discouraged, while the use of other communication resources is favored. However, the global bus is always assumed to be available, as it guarantees a possible datapath mapping provided enough rDPUs are available.

In the classic KressArray-I, the data transfers over the global bus are handled by a central control unit. This unit needs controlling data, containing a schedule of all bus transfers. This data comprises also additional information to handle dynamic transfers which occur e.g. in conditional loops. This concept is supported by the exploration framework by supplying an adequate control file for such a control unit. An alternative implementation of the global bus uses a bus arbitration hardware. Although possible, this approach bears the disadvantage of additional overhead due to the large number of bus writers.

6.3.2 Nearest Neighbor Connections

Nearest neighbor connections provide a link between two rDPUs, which are either horizontally or vertically adjacent. If the application shows a very high regularity, direct neighbor connections alone may be sufficient to provide the necessary communication. Examples for such applications are found in systolizable algorithms, which can be implemented as systolic arrays [Kung82]. In the general case, longer connections are implemented as paths constructed from several neighbor connections and routed through rDPUs. An example showing a mapping with longer paths can be seen in figur e6-3.

In the KressArray design space, horizontal and vertical connections may be defined in any combination. Also, the number of connections can be defined separately for the horizontal and the vertical ones. A single connection has the following individual properties:

- The type of the connection.
- The torus structure

These topics are being discussed in the following sections.

6.3.2.1 Types of Connections

There are two basic types of nearest neighbor connections. Unidirectional connections and bidirectional connections. Unidirectional connections, also referred to as simplex connections, can transfer data only in one specific direction. Thus, there are four types of unidirectional links, going to north, east, south and west. A bidirectional connection, also called a half-duplex connection, can be either horizontal or vertical. The direction of the data transport is determined by the configuration and is supposed to remain static during the operation of the KressArray. Examples for rDPUs with different nearest neighbor connections is shown in figure 6-11.

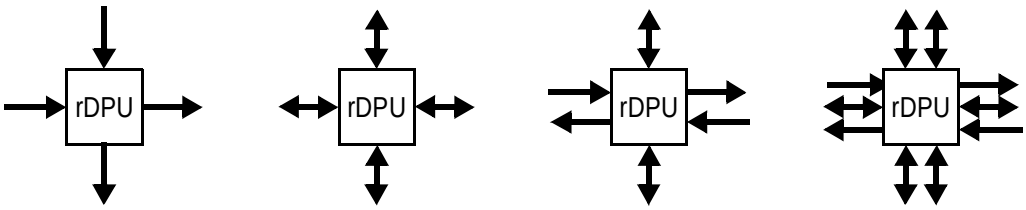

Figure 6-11: Examples for rDPU cells with different nearest neighbor connections

6.3.2.2 Torus Structures

For each individual nearest neighbor connect, a torus structure can be set, defining extra links between opposing ends of the array. Thus, the array is becoming topologically equivalent to a torus. The extra links can be implemented on the PCB or by a separate layer on the chip. A restriction applies, in that nearest neighbor connections which feature a torus structure can not be used to link to edge ports (see section 6.4.3.2 below). There are four possible torus structures supported for each single nearest neighbor connection. The structures are:

- "*None*": This means, there is no torus structure, and the connection ends at the according array border.
- "*Same*": This describes a torus structure, where a rDPU at the edge of a specific row or column is connected with the rDPU at the opposing edge of the same row or column.
- "*Next*": In this structure, a rDPU at the edge of a row or column is linked to the opposing rDPU of the next row or column. For a vertical connection, the rDPU at the upper end of a column is connected to the

rDPU at the lower end of the column to the right, for a horizontal connection, the rDPU at the right end of a row is connected to the one at the left end of the row below.

- "*Prev*": This describes a structure, where a rDPU at the edge of a row or column is linked to the opposing rDPU of the previous row or column. For a vertical connection, the rDPU at the upper end of a column is connected to the rDPU at the lower end of the column to the left, for a horizontal connection, the rDPU at the right end of a row is connected to the one at the left end of the row above.

The four supported torus structures are summarized for horizontal and vertical connections in figure 6-12.

6.3.2.3 Routing Channels

A further property related to nearest neighbor connections is the capability of rDPUs to pass through data words, which are not consumed or produced by the rDPU itself. Such links, which are also called routing channels in the following, are not only possible for routing elements, but also for operator rDPUs. An example for such a scenario is shown in figure 6-1b. The rDPU itself performs an operation with two operands, which are input at the north side, and outputs one operand and the result to the south. Additionally, a data word is transported from the western input to the eastern output. This additional transport is considered to be a routing channel. Depending on the physical implementation of the rDPUs, the number of possible routing channels may be limited. This number can be controlled by the global parameter *MaxRoutChan*. Even if the maximum number of routing channels is not limited, this parameter can be used to steer the design process towards the usage of other communication resources.

6.3.3 Row and Column Backbuses

Row and column backbuses are a novel communication resource for KressArrays. They represent long range connections which span several rDPUs in a row or column. These buses are an efficient communication means for data words to be broadcast to several rDPUs along a line. There may be several buses in one row or column, which may be segmented individually. Different bus implementations are supported, which imply different numbers of writers allowed to access the bus. A backbus is defined by the following properties:

- The bus type (row bus or column bus)
- The segmentation information
- The capacity "*MaxWriter*" (maximal number of writers allowed)

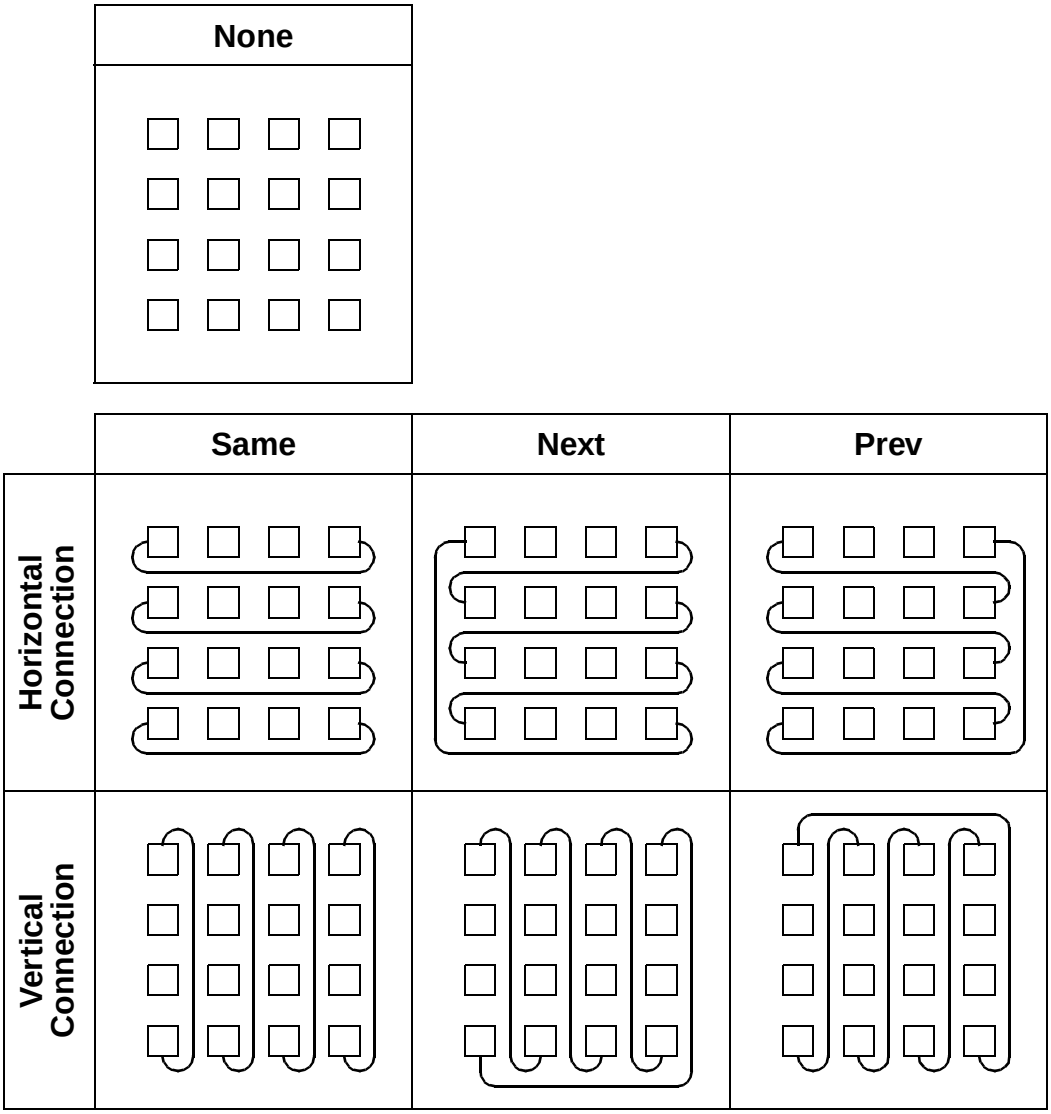

Figure 6-12: Illustration of torus structures

The bus type property consists of the two alternatives of horizontal row bus or vertical column bus. The segmentation information, cost parameters, and number of writers are discussed in the following subsections.

6.3.3.1 Segmentation

A row or column bus does not necessary have to span the whole array. Instead, it may be split up into several segments of a specific length. Furthermore, the length of the first segment may differ to allow a different coverage of rDPUs by the segments. The first segment is the one to the left for row buses and the top one for column buses. Following this approach, the segmentation information for an

individual backbus consists of two parameters, the segment length *SegLength* and the length of the first segment *FirstSeg*. To illustrate these parameters, examples for row buses with different segmentation are given in figure 6-13.

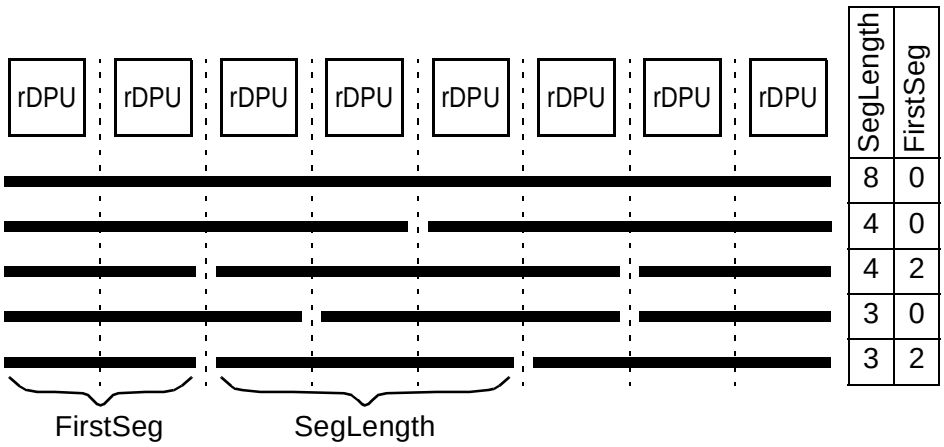

Figure 6-13: Example row backbus segmentations

6.3.3.2 Backbus Capacity

An important property of a backbus is its capacity, describing how many writers may use this bus in one configuration. A backbus with a capacity of one resembles basically a long range connection, as known from some FPGAs or other reconfigurable architectures (see section 2). The parameter describing the bus capacity is also referred to as *MaxWriters* in the following chapters.

If *MaxWriters* is greater than one, a similar situation applies as for the global bus. One implementation for multi-writer backbuses consists in providing an arbitration logic for each backbus. For this possibility, no further software support is necessary.

Distributed Transfer Control. In addition to arbitrated backbuses, an alternative implementation is supported, which uses a simple hardware for distributed bus transfer control. This implementation is described in the following. It requires a distributed schedule for each rDPU, which is provided by the exploration framework. The basic principle of the distributed transfer control utilizes the property of a deterministic schedule, which allows to statically assign to every rDPU which accesses the bus a time slot in which this access will happen. As the number of readers and writers is known at compile time, also the number of needed time slots is known. Thus, each rDPU can sense the bus and wait a certain number of bus transfers between consecutive bus accesses. This number of cycles, called *CycleLength* is the same for all rDPUs. However, to make sure the transfers happen in different time slots, the first transfer after system start or a

reset has to be delayed according to the position of the appropriate time slot. These *Latency* parameters are different for each writer. They are the same for a writer and its associated readers. To illustrate this mechanism, a simple example is shown in figur e6-14.

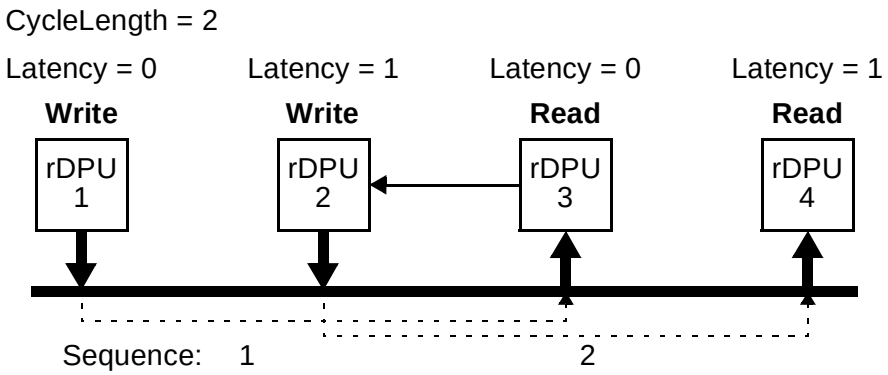

Figure 6-14: Example illustrating distributed transfer control by scheduling

In the example, rDPUs 1 and 2 transfer data on a row bus to rDPUs 3 and 4 respectively. As the figure suggests, rDPU 3 transfers also a word to rDPU 2 via a nearest neighbor connection. Thus, rDPU 2 can not start, until rDPU 3 has finished, which will in turn not happen, before rDPU 1 transferred its data word. Therefore, the transfer from rDPU 1 to 3 has to happen before the transfer from 2 to 4. This sequence is determined by a separate scheduling step done by the scheduler (see section 7.4). Following this sequence, the *Latency* for rDPUs 1 and 3, is set to 0, as this transfer has to start first. Accordingly, the *Latency* for rDPUs 2 and 4 is set to 1, as their transfer has to be delayed to the next time slot. As there are two writers in this scenario, the common *CycleLength* for all rDPUs is set to 2. The exploration framework provides adequate *Latency* and *CycleLength* parameters, if one or more backbuses are defined to allow multiple writers.

A major drawback of the approach described above is its need for a deterministic schedule. Therefore, dynamic connections, which appear in conditional loops, are not supported. On the other hand, the mechanism can be implemented with moderate hardware effort. A simplified block diagram of a possible implementation is shown in figur e6-15.

The basic control circuitry uses three registers and two downwards counters to implement the access mechanism for one bus port. This implementation allows the port to be used for concurrent reading and writing in one configuration. The counter is preloaded at a reset signal (or system start) with the according *Latency* value, and clocked with the transfer events sensed on the bus. Whenever it reaches zero, the counters are reloaded with the *CycleLength* and a bus access is possible for reading or writing respectively. The counters and the registers are *n*

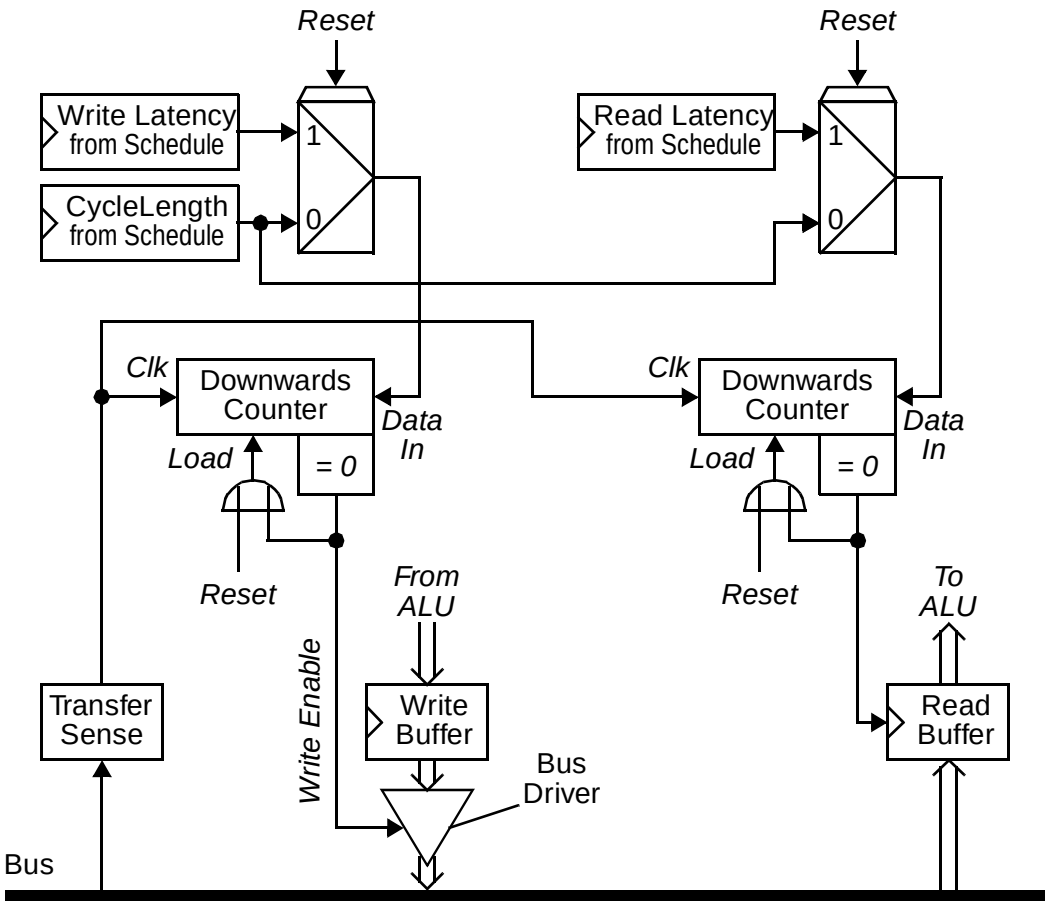

Figure 6-15: Example implementation of distributed backbus schedule execution (compare figure 6-14)

bits wide, with $n = \lceil \mathrm{ld}\ ArraySizeX \rceil$ for row buses and $n = \lceil \mathrm{ld}\ ArraySizeY \rceil$ for column buses. *ArraySizeX* and *ArraySizeY* are the horizontal and vertical array dimensions (see also section 6.4.1 below).

6.4 Array Properties

The KressArray itself is supposed to be a rectangular array of rDPUs. However, the array does not have to be homogenous, as it is possible to have rDPUs with different capabilities in different areas of the array. This novel feature alllows a more efficient realization of KressArrays, where complex operators do not have to be available in each rDPU. Properties to the global array structure affect those capabilities for heterogeneous structure, the size of the array, and properties concerned with data transfers to and from the array. All array properties are stored in the intermediate format.

6.4.1 Array Dimensions

The array is assumed to be a rectangular arrangement of rDPUs, thus the size is expressed in the length and the width of this array. For compatibility reasons, the physical array is supposed to be built on one chip or hierarchically from several chips, with each chip having a certain number of rDPUs. Thus, there are four parameters describing the array size:

- The number of rDPUs in a row on one chip ("*ChipSizeX*").
- The number of chips in a row forming the array ("*ChipCountX*").
- The number of rDPUs in a column on one chip ("*ChipSizeY*").
- The number of chips in a row forming the array ("*ChipCountY*").

Thus, if *ChipCountX*=1 and *ChipCountY*=1, then the array is implemented on a single chip. Using these definitions, the total width and height of the array also known as *ArraySizeX* and *ArraySizeY,* are defined by:

*ArraySizeX = ChipSizeX * ChipCountX ;*

*ArraySizeY = ChipSizeY * ChipCountY ;*

The total number of rDPUs is then given by:

*TotalArraySize = ArraySizeX * ArraySizeY ;*

The required size of the array is strongly dependent of the application, as enough rDPUs have to be available to hold the operators of the datapath. However, also the aspect ratio, i.e. the relation of width to height, is of importance in some cases, as this affects the efficient mapping of datapaths onto the array.

6.4.2 Heterogeneous Arrays

Although the array is assumed to be rectangular, the rDPUs themselves do not need to have the same capabilities. This means, that some operators may not be available in all rDPUs. In contrast to most existing approaches, the resulting array is heterogeneous. To implement such a heterogeneous structure, a relation between the operators of the application and the rDPUs has to be built. This relation can be formalized as a function *HasCapability* as follows:

$HasCapability : O \times D \rightarrow \{ True, False \} ;$

with O being the set of operators of an application, and D the set of available rDPUs.

HasCapability(o, d) = True, if and only if operator o can be mapped onto rDPU d, for every $o \in O$ and every $d \in D$.

This function is realized using an attribute for each rDPU and each operator, which is implemented as a bitmask, similar to the double-output operators (see above). An operator may be mapped onto a rDPU, if the logic AND of their bitmaps results in a non-zero value. Thus, groups of operators and rDPUs can be built by assigning a bit position to a group.

This concept shall be illustrated by a simple example. Let there be a hypothetical operator set consisting of addition ("+"), multiplication ("*") and hamming distance calculation, as used in some algorithms in communication ("HD"). Let there be also a hypothetical KressArray with two types of rDPUs. Type A can implement addition or multiplication, type B can implement addition or hamming distance calculation. Then the *HasCapability* relation is defined as follows:

HasCapability("+", A) = True ;

HasCapability("", A) = True ;*

HasCapability("HD", A) = False ;

HasCapability("+", B) = True ;

HasCapability("", B) = True ;*

HasCapability("HD", B) = True ;

This relation may be expressed using a bitmask of two bits. Bit position 0 (the least significant bit) is used for rDPU type A, while position 1 is used for type B. The operators have the corresponding bit set in their bitmasks, if they can be mapped onto the according rDPU type. The resulting bitmasks are shown in table 6-3.

rDPU type			**Bitmask**
A			0 1
B			1 0
Operator	**Map on A**	**Map on B**	**Bitmask**
Addition ("+")	Yes	Yes	1 1
Multiplication ("*")	Yes	No	1 0
Hamming Distance ("HD")	No	Yes	0 1

Table 6-3: Example for bitmasks in heterogeneous KressArrays

If there are rDPUs with different capabilities, they will typically have a different chip area than the other rDPUs. Thus, a regular arrangement of such rDPUs, e.g. in a column or in a row, is advantageous, as the chip layout is not that much disturbed. A regular arrangement is supported by the possibility to define bitmasks for several rDPUs at once. The mechanism employed uses two nested spatial iterations to traverse the array, one for each of the horizontal and vertical direction. These iterations are controlled by a set of 6 parameters, plus the bitmask value to be set in the according rDPUs. The six parameters represent the start, end, and step width of the corresponding iteration. The general concept is shown in figure 6-16a, showing the iteration in a pseudo code notation, accompanied by a simple example in figur e6-16b.

a) Parameter Set:
(BaseY, StepY, LimitY) (BaseX, StepX, LimitX) MaskValue

```
FOR Y=BaseY TO LimitY STEP StepY
{     FOR X=BaseX TO LimitX STEP StepX
      {     Set rDPUMask[Y,X] to MaskValue;
      }
}
```

b) Example:
(0, 1, 7) (1, 2, 3) *x*

```
FOR Y=0 TO 7 STEP 1
{     FOR X=1 TO 3 STEP 2
      {     Set rDPUMask[Y,X] to x;
      }
}
```

	0	1	2	3	4	5	6	7
0		X		X				
1		X		X				
2		X		X				
3		X		X				
4		X		X				
5		X		X				
6		X		X				
7		X		X				

Loop instances for example

Example 8x8 KressArray

Figure 6-16: Nested spatial iterations for definition of rDPU bitmasks:
a) General approach in nested loop notation
b) Example parameters and resulting KressArray structure

The example parameter set sets the bitmasks in the second (number 1) and fourth (number 3) column of an 8 by 8 KressArray. The first row and column have the index 0. Thus, the vertical loop has to run over the whole height of the array with a step width of 1. The resulting (first) parameter set is (0, 7, 1). The horizontal loop should start at position 1 and end at position 3, leaving out every second rDPU. Thus, the parameter set yields (1, 3, 2).

A more illustrative example for the use of array capabilities is given in figure 6-17. The two parameter sets described in figure 6-17a and figure 6-17b define two ranges of rDPUs, which together describe a chessboard layout. Setting the bitmaps in the rDPUs to zero results in those rDPUs being incapable of implementing any operators. Thus, a mapping on the resulting architecture has to use the remaining rDPUs to place the operators, like shown in the example mapping in figure 6-17c.

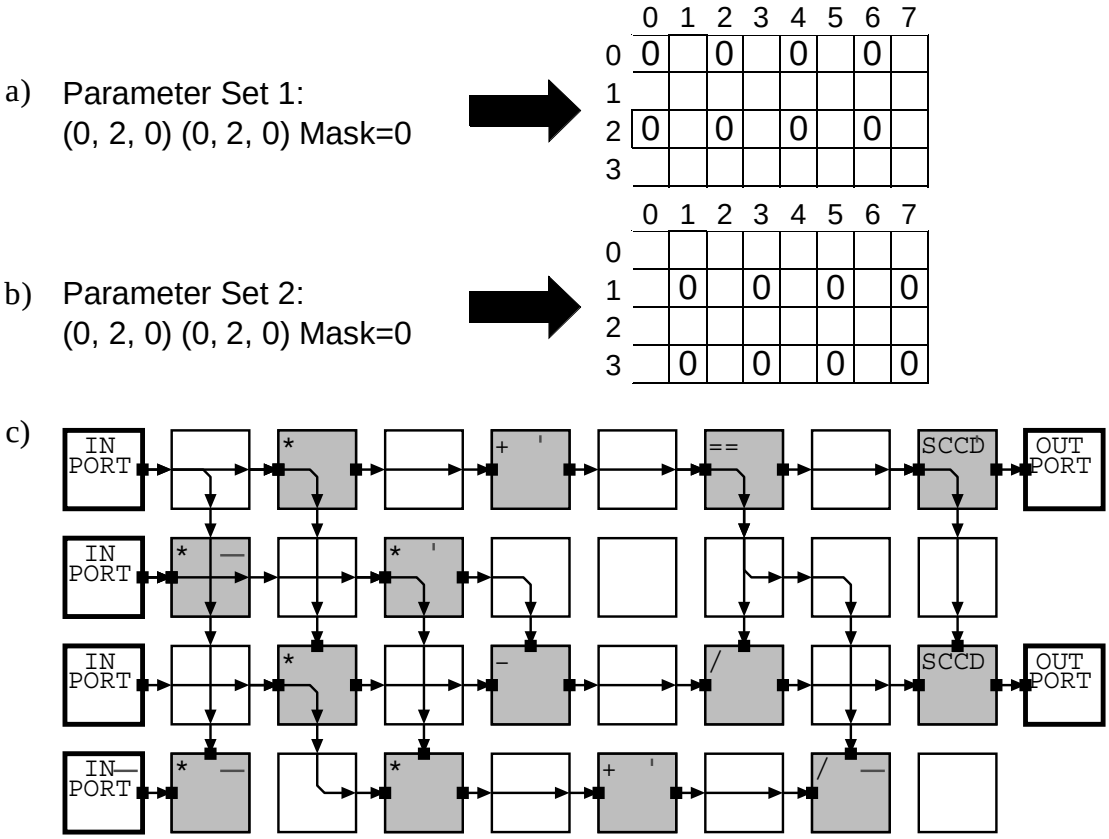

Figure 6-17: Example for heterogeneous array using array capabilities:
a), b) Parameter sets defining two ranges of rDPUs as not capable of implementing any operators
c) Example mapping on resulting architecture

6.4.3 Data I/O

As the KressArray is used to implement a computational datapath, features must be provided for data streams of operands, which lead into the array, and computation results, which leave the array. At datapath level, these data streams lead to the rDPUs at the beginning of the datapath for input streams, or at the end of the datapath for output streams.

The data streams are normally implemented by connecting the KressArray to a data memory, which is addressed by a suitable address generator, e.g. the one described in [HBHN97a], [HBHN97b]. A logic connection between a rDPU and such a data source or sink is called a virtual port in the following. The realization of a virtual port is done by mapping it to a physical port. The available physical ports are dependent on the architecture. In the KressArray design space, three possibilities for physical ports exist: The serial global bus, the edges of the array, or rDPUs inside the array.

6.4.3.1 Ports Using the Global Bus

The serial global bus (see section 6.3.1) resembles a physical port, which has been adopted from the historical KressArray-I prototype. The advantage of this concept is the capability of the global bus to carry an arbitrary number of virtual ports, reaching every rDPU independently of the location in the array. However, the general drawback of the global bus, being likely to become a performance bottleneck, applies also. A bus control unit acts as an additional interface between the KressArray and the data memory. As all transfers over the global bus are handled by this controller, the transfers to the data memory have also to be considered in the controller data to be supplied for proper operation. The basic scenario is illustrated in figure 6-18.

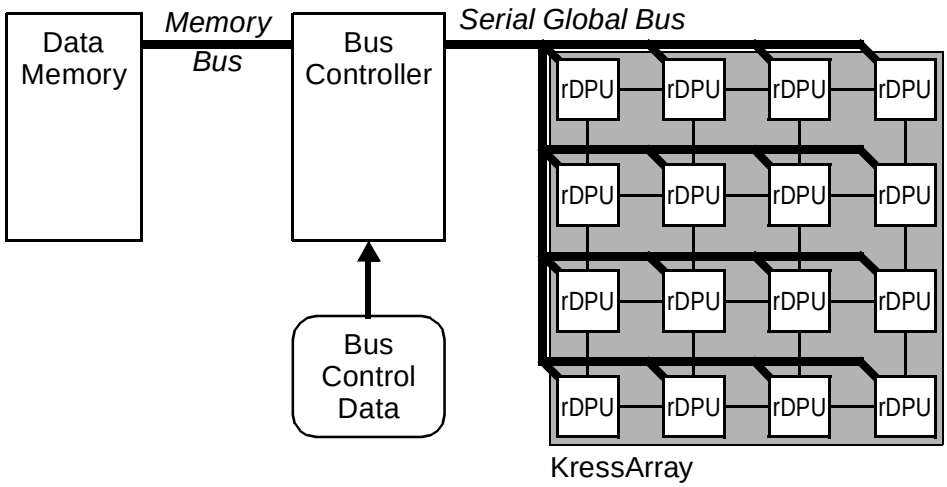

Figure 6-18: System setup for data I/O over global bus

6.4.3.2 Ports at Array Edges

Data words may also enter and leave the array over the edges. The ports may reside on any of the four array borders. To enable data transfers over the edges, communication resources available inside the array have to be also available at the borders. For the different connection resources, the following rules apply:

- Nearest neighbor connects are assumed to extend to the edges only if no torus structure has been defined for them (see section 6.3.2.2 above). If the connection is unidirectional, it can be accessed over the edge only in the specified direction.
- For row and column backbuses, the segment nearest to the corresponding border is assumed to extend to this border, so it is accessible over the edge.
- The global bus is generally assumed to be connected to the edges. Thus, an edge port may transfer data also over the global bus, if there is no other possibility.

Depending on the actual connection of the data stream sources to the physical ports, the position of a virtual port may be restricted. Generally, it is assumed, that the side of the array, where the port has to reside, is fixed. The exact position of the port may be restricted or subject to the synthesis step. Thus, an edge port is defined by the side of the array and by the position on the specified side.

The side of the array is specified by the *PortSide* parameter, which may have the value of north, east, south or west. For the position, there are three possibilities.

- No restrictions apply for the position. In this case, the exact position on the specified side is determined by the synthesis step following the optimization objective.
- The port has to be inside a certain range at the specified side. This range is specified by its first and last position described by parameters *FirstPos* and *LastPos*. The exact position of the port is again determined by the synthesis step, but will be inside the specified range.
- The port has to be at one specific position. This is a special case of the previous scenario, with *FirstPos* and *LastPos* being equal.

An example architecture, where edge ports are used to attach multiple memories to one KressArray is depicted in figure 6-19.

The example architecture uses two independent memories for input operands and two memories for result values. As can be seen, the following port requirements are implied by this architecture:

Any virtual port for the input of data from memory A has to have the *PortSide* set to west. Furthermore, the port must lie in the range of "I0 ... I1". Thus, *FirstPos* has to be set to 0 and *LastPos* has to be set to 1. The virtual ports linking to memory B also have to be placed on the west edge by setting*PortSide* to west. The range for memory B is "I2 ... I3". Therefore, *FirstPos* must be set to 2 and *LastPos* to 3. The output data memories are attached to the right side of the array. Thus, any virtual port for the transfer of results has to set the *PortSide* to east. However, there is no restriction for the position of the ports. As both output memories share a common

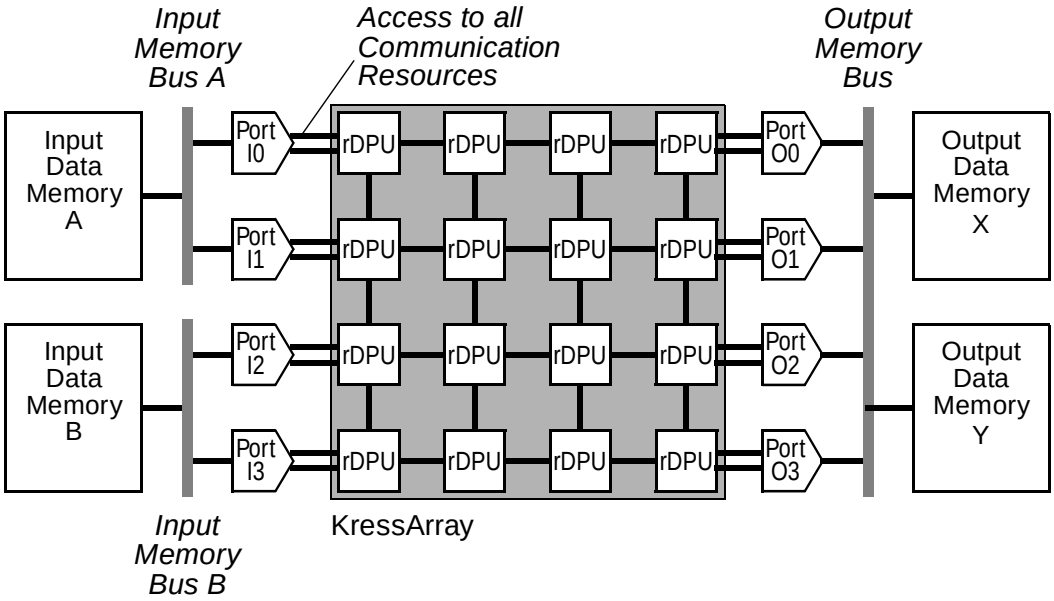

Figure 6-19: Example architecture configuration for edge ports

memory bus spanning the whole array, a virtual port writing to either memory can lie at any vertical position of the eastern edge. Therefore, the position specification can either be omitted or set to the maximum by defining *FirstPos* to 0 and *LastPos* to 3.

6.4.3.3 Ports Inside the Array

Although ports are mainly a means for communication of the KressArray to the outside, it is sometimes desirable to have ports also inside the array. These ports resemble rDPU positions, and are defined by two coordinates, the x and y position *XPos* and *YPos* respectively. There are mainly two cases which require internal ports, IP (intellectual property) cores and special cells in heterogeneous structures.

IP Cores. According to standard cell libraries known from VLSI design, certain functions may be mapped in advance, resulting in a mapping with a fixed placement on the array. These IP cores may be incorporated into other datapaths by copying the mapping of the function into the KressArray and synthesizing the new datapath to link with the function. For the synthesis of the new datapath, the structure of the IP core does not have to be disclosed to the designer of the new datapath. Instead, the positions of the interfacing rDPUs of the IP core are defined as ports to the new datapath, and the rDPUs occupied by the IP core are masked out for use by the synthesis process. In a post-processing step, both mappings are

merged to form the new complex datapath. This post-processing step can be done when the final configuration data is generated, therefore hiding any information about the internal structure of the IP core.

To illustrate this concept, an example for the application of an IP core is shown in figure 6-20. The desired application datapath is given by the expression:

```
Result  = 6 * GCD( X, Y ) + Z
        = ( GCD( X, Y ) << 1 ) + ( GCD( X, Y ) << 2 ) + Z ;
```

With GCD denoting a function to calculate the greatest common divisor. The multiplication is expressed by two shifts and one addition. The GCD function is supposed to exist as an IP core, which is to be incorporated into the rest of the datapath.

For illustrational purposes, the mapping of the GCD function is shown in figure 6-20a. The mapping implements the dataflow graph shown above in figure 6-8, which uses a while loop. Altogether, nine rDPUs are occupied by operators, an additional three rDPUs are used only for routing. There are two input ports and one output port. The IP core derived from this mapping is shown in figure 6-20b. The ports at the edges are no more used. Instead, the rDPUs, which were connected to the ports, are now the interface to the embedding datapath. These rDPUs are drawn with a fat border in the figure. These rDPUs will be called the interface rDPUs of the GCD core in the following explanations.

The rest of the datapath, which embeds the IP core, consists of two shifters and two adders. To the outside, this datapath has three inputs and one output. To incorporate the IP core, the rDPU positions occupied by the GCD datapath are masked out for the synthesis of the embedding datapath. This scenario is shown in figure 6-20c, with the IP core rDPUs crossed out diagonally. As can be seen, the connection between the GCD core and the embedding datapath is realized by the interface rDPUs of the GCD core. From the view of the embedding datapath, the inputs of the core are ports to write data to, and the output of the core resembles a port to get intermediate results from. Thus, the input interface ports of the GCD core appear as output ports inside the array. The output interface port appears as an input port inside the array. These ports appear at the positions they occupy in the GCD mapping. When the final configuration code is being generated, the embedding datapath is combined with the GCD core. The resulting datapath is shown in figure 6-20d for illustrational purposes only. The internal structure of the core would normally be hidden.

Special Cells. As described in section 6.4.2, the rDPUs of the KressArray do not have to have the same capabilities, so heterogeneous array structures are possible. As a special case, some cell positions may have no computation facilities at all but may contain special cells, like for example embedded memories. An

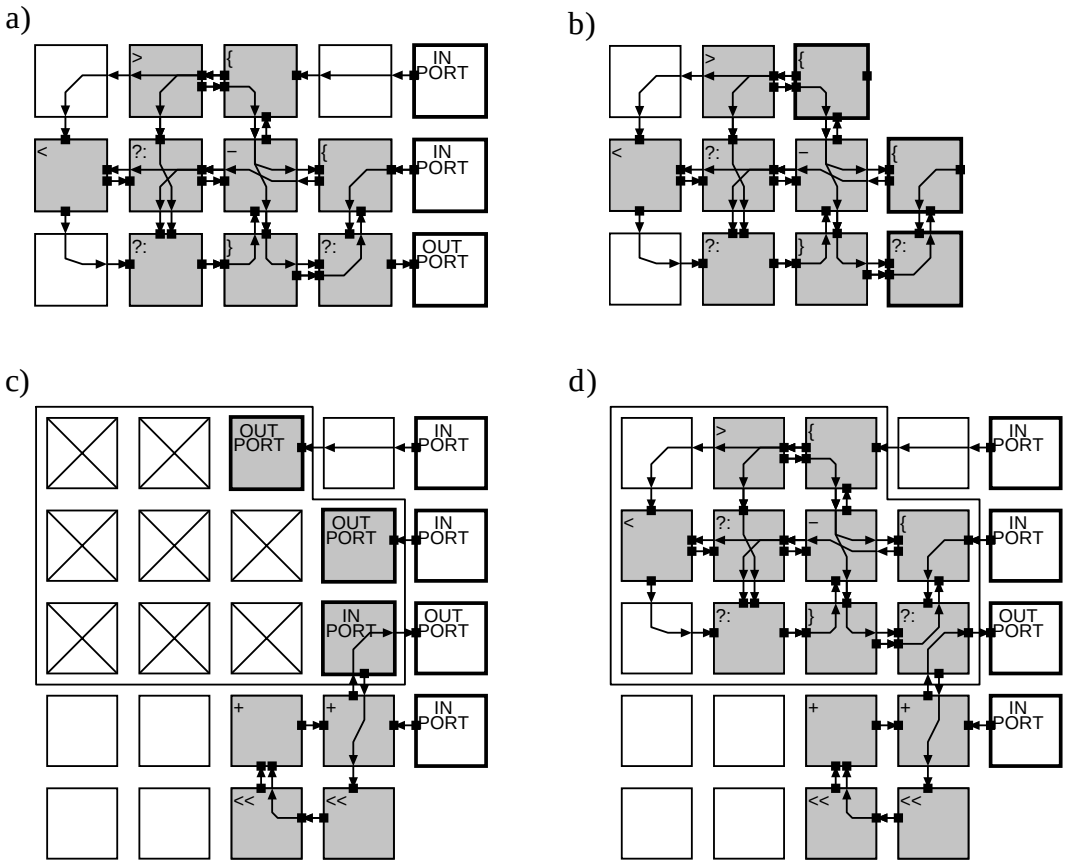

Figure 6-20: Mapping with embedded IP core:
a) GCD datapath
b) GCD core with interface rDPUs marked
c) Embedding datapath with core positions masked out
d) Overall mapping

example for a coarse grain architecture with such features is the CHESS array (see section 2.1.9), which does have embedded memory cells together with the functional units.

Such special cells like described above are located at fixed rDPU positions and cannot be incorporated into a dataflow graph, as they do not represent an operation. A viable way to incorporate those cells is to define the input and output positions of the cells as ports inside the array and mask out the cell positions for the synthesis process. The datapath must afterwards be properly connected to the special cell positions.

6.4.4 Port Grouping

The port grouping feature of the exploration framework makes sense mostly for edge ports. According to the specific KressArray architecture, several memory banks may be attached to the array. If the memories use separate buses, accesses to these memories can happen in parallel. However, all accesses to the same memory or over the same bus to different memories have to happen sequentially. For the sequential accesses, the exploration framework will provide a schedule according to the datapath. In order to specify, which virtual ports can make independent accesses and which use the same bus as others, the ports can be divided up in groups. All ports in one group use the same bus and have therefore to be scheduled together. Ports in different groups are supposed to be independent of each other. The group for a specific port is defined by the *PortGroup* parameter. This number can be set to an arbitrary value by the user. Ports with the same *PortGroup* value are in the same group.

To illustrate this concept, the architecture shown in figure 6-19 is used as example. There are three memory buses, thus there are three port groups required. The values for *PortGroup* are chosen arbitrarily as 1, 2, and 3. The ports connecting to input memory A can access this memory independent of all other memories. However, among each other, they have to be scheduled. Thus,*PortGroup* for these ports is set to 1. The ports accessing input memory B have also independent access, as this memory has an own bus. The *PortGroup* value for the ports accessing memory B is therefore set to 2. The output memories X and Y share the same bus, which prohibits concurrent accesses to these memories. Thus, all ports accessing either memory X or Y have to be scheduled in the same group. this is done by setting their *PortGroup* to 3.

6.5 Classification of KressArray Prototypes

To illustrate the KressArray design space parameters, two prototypes of the KressArray architecture will be described briefly and classified by the parameters discussed in the previous sections. These prototypes are the historical KressArray-I [Kres96], which is the grandfather of the KressArray family, and a more sophisticated architecture, which has been emulated [Zimm99].

6.5.1 Classification of KressArray Architecture I

The KressArray-I has been described already in section 2.1.2. As it includes some basic features of the KressArray architecture family, it has also been referenced several times in the discussion above. Thus, the architecture classification is merely summarized in this section.

6.5.1.1 rDPU Properties

The rDPU properties relevant to the KressArray-I are restricted to the operator set and the bitwidth. Further concepts like double-ALU rDPUs and double-output operators are not implemented.

Operator Set. A single rDPU is capable of implementing all integer operators of the programming language C. Additional operators may be defined by microcoding them into the rDPUs.

Bitwidth. The bitwidth of the KressArray-I is 32 bits for datapaths and operators.

6.5.1.2 Communication Resources

The KressArray-I features a global bus, and nearest neighbor connections. Row or column backbuses are not implemented.

Nearest Neighbor Connections . There are two unidirectional connections. One Simplex connection to the east and one to the south. The resulting rDPU architecture is pictured above in figure 6-11. Although not implemented by default, seven different torus structures are supported, as described in table 6-4.

KressArray-I Terminology	Torus Structure for Simplex East	Torus Structure for Simplex South
Structure 0	None	None
Structure 1	Same	Same
Structure 2	Next	Same
Structure 3	Prev	Same
Structure 4	Same	None
Structure 5	Next	None
Structure 6	Prev	None

Table 6-4: KressArray-I torus structures

The routing channels are not restricted in the KressArray-I. Therefore, a rDPU which is employed as a routing element can implement two routing channels using all its nearest neighbor connections, thus *MaxRoutChan* = 2 due to the restricted communication resources.

6.5.1.3 Array Properties

The KressArray-I features rDPUs which all have the same capabilities and does therefore not implement heterogeneous structures. The relevant properties are the dimensions and the Data I/O.

Array Dimensions. The array has a hierarchical structure, being built of an array of two by four chips with each chip containing an array of three by three rDPUs. The connections between the chips are significantly slower than those between rDPUs on the same chip. Thus, a hierarchical representation follwing the conventions in section 6.4.1 is adequate. The parameters have the following values:

ChipSizeX = 3 ;

ChipSizeY = 3 ;

ChipCountX = 4 ;

ChipCountY = 2 ;

The according array sizes calculate to:

ArraySizeX = 12 ;

ArraySizeY = 6 ;

TotalArraySize = 72 ;

Data I/O. All data communication from or to the KressArray-I is done over the global serial bus. No edge ports or ports inside the array are supported. As the global bus is scheduled by default, no additional port grouping is required.

6.5.2 Classification of KressArray Architecture II

The KressArray architecture II described here has been emulated in a research project [Zimm99] using Motorola MC68EC000 processors [Moto93] and Xilinx XC4010XL FPGAs [Xili98]. The design space parameters for this architecture are described in the following:

6.5.2.1 rDPU Properties

This architecture does not implement double-ALU rDPUs or double-output operators. Thus, the relevant properties are the operator set and the bitwidth.

Operator Set. Like the KressArray-I, the rDPUs of the emulator can implement the integer operators of C. Additional operators can be programmed using MC68000 assembler.

Bitwidth. The bitwidth of the emulator is 32 bits for datapaths and operators.

6.5.2.2 Communication Resources

The KressArray emulator has enhanced communication resources compared to the KressArray-I. It features all communication resources, i.e. the global bus, nearest neighbor connections, and backbuses.

Nearest Neighbor Connections . There are two bidirectional nearest neighbor connections on each side of the rDPU. Following the terminology in section 6.3.2.1, there are two horizontal half-duplex connections and two vertical half-duplex connections. Torus structures are not supported. There is no restriction of the number of routing channels. Thus, with eight connections at one rDPU, *MaxRoutChan* equals to 4.

Row and Column Backbuses. The KressArray emulator features one row and one column backbus, each having the same properties. Both buses span the whole array, thus having the segmentation parameters *FirstSeg = 0* and *SegLength = 4*. Both buses support only one writer in a configuration. Thus, *MaxWriters* = 1, and no special scheduling is required.

6.5.2.3 Array Properties

Being assembled of 16 modules, the KressArray emulator has a simpler structure than the KressArray-I. Being also a homogenous array, the array properties reduce to the dimension specification and the I/O ports.

Array Dimensions. In contrast to the KressArray-I, the emulator is a plain array of four by four rDPUs. Thus, the dimension parameters are given by:

ChipSizeX = 4;

ChipSizeY = 4;

ChipCountX = 1;

ChipCountY = 1;

The according array sizes calculate to:

ArraySizeX = 4 ;

ArraySizeY = 4 ;

TotalArraySize = 16 ;

Data I/O. The KressArray emulator performs all data input and output over the serial global bus.