A. Appendix

In this chapter, additional information to the topics discussed before is provided. In the next section, the intermediate file format used by the MA-DPSS (see section 7) is described in detail. Then, an overview of statistic data generated by the analyzer tool described in section 8.3 is given. To illustrate the concepts of design suggestion generation, a simple example rule set is presented in the section afterwards. Finally, an exploration example is discussed, which uses the example rule set.

A.1 The Intermediate File Format

This section describes the intermediate format employed by the KressArray Xplorer framework to store an architecture information, application datapaths, and mappings. The exact amount of information depends on the type of the intermediate format. There are three types, which correspond to the stages in the MA-DPSS subsystem described in section 7.

- α-intermediate format contains information about the datapath of an application, containing the operators used and their dependency amongst each other. The representation of the datapath corresponds to a netlist as known from VLSI design systems.
- β-intermediate format contains all the information of the α format and additionally architectural data, like the number and type of communication resources or the size of the array.
- γ-intermediate format contains the data from the according β format and a mapping of the datapath on the given architecture. The mapping resembles the main part of the configuration code for the KressArray architecture.

The three types of intermediate format share the same syntax, with information being added to a file in a certain type to get a file of the next type. Thus, α format lacks the architecture information of the corresponding β file, and the γ file resembles the β file, plus the mapping information.

In the following, the general structure of the intermediate format, consisting of five sections will be described. Then, each section will be explained in more detail. Though the description has more a tutorial character, certain syntactical conventions have been used:

- Bold words denote keywords and tokens.

- Expressions in angular brackets "<", ">" denote syntactic elements, which are further structured, unless specified otherwise.
- Italic expressions in single quotes denote character strings.
- Unquoted italic expressions denote numeric values, either integer or float.

A.1.1 General Structure

A file in intermediate file format is made out of ASCII text arranged in lines. Each line starts with a keyword, followed by possible operands to this keyword, and terminated by a semicolon (";"). Such a line is called a statement in the following, the statements concerned with the general file structure will be discussed.

A.1.1.1 The SECTN and ESECT Statements

The intermediate format comprises five sections, which must be present in a file, although some sections are allowed to contain no data. The following sections exist, and have to appear in the file in the order given:

- Hardware parameter section
- Communication resource section
- Array header section
- Subnet list section
- rDPU mapping section

Each section is enclosed between section delimiter statements. At the start of the section, a SECTN statement is placed, while at the end, an ESECT statement has to appear. Both the SECTN and the ESECT statement do not have parameters. Thus, the coarse structure of an intermediate file looks like this:

SECTN ;
<Hardware parameter section contents>
ESECT ;
SECTN ;
<Communication resource section contents>
ESECT ;
SECTN ;
<Array header section contents>
ESECT ;
SECTN ;
<Subnet list section contents>
ESECT ;
SECTN ;
<rDPU mapping section contents>

ESECT ;

A.1.1.2 Extern Information with XB??? and XE??? Statements

The intermediate format provides a way to include extern information into the file. This information is supposed to be of relevance for extern usage and is to be skipped by the design tools. Extern data may be added at the beginning of a section, using a XB??? statement, and at the section end using a XE??? statement. The synopsis is as follows:

XB??? <Any text> **;**
XE??? <Any text> **;**

A XB??? or XE??? statement begins with a five-letter keyword, which has to start with "XB" or "XE" respectively. The remaining three letters may be chosen according to the purpose of the information. After the keyword, a line of arbitrary data follows, which is concluded by a semicolon.

A.1.2 Hardware Parameter Section

This section contains some of the array properties, parameters relevant for the mapping process, and a part of the delay values for the scheduler as numeric values. The types of parameters are:

- Architecture property parameters
- Annealing parameters
- Global parameters
- Routing control parameters
- Scheduling parameters

There is only one type of statement inside this section, thePARAM statement.

A.1.2.1 The PARAM Statement

The PARAM statement consists of the keyword followed by the parameter name and the value for this parameter. The value is either an integer or a floating point number, depending on the parameter.

PARAM <Parameter name and parameter value> **;**

The supported parameters will be described briefly in the following.

Architecture Property Parameters. There are five parameters dealing with architecture properties. Four of them dealing with the array size, and one denoting the bit width of the operators.

ChipSizeX *integer*

ChipSizeY *integer*
ChipCountX *integer*
ChipCountY *integer*

These parameters define the array size as being indicated in sectio n6.4.1.

BitWidth *integer*

This parameter defines the width of the datapaths in bits. Its meaning is described in more detail in section 6.2.2.

Annealing Parameters. These parameters influence the annealing process. They resemble the parameters with the same name used in the cooling schedule described in section 7.3.1.3.

MaxTemp *float*

This is the maximum temperature for annealing. This parameter is kept for historical reasons and has no more practical relevance.

CurTemp *float*
MinTemp *float*
TempFactor *float*
Iterations *integer*
NoChangeDie *integer*

These parameters correspond to those described in secti on7.3.1.3.

Global Cost Parameters. These parameters contain the cost parameters for the global bus and for routing elements, as well as the current cost of the mapping. There are five parameters in this class:

CostBusBase *integer*
CostBusStep *integer*
CostBusWrite *integer*

These three parameters define the cost function for global bus connections as described in section 7.3.3.1.

CostRoutElem *integer*

This is the cost for a routing element as described in section 7.3.3.3.

CurrentCost *integer*

This value is set by the mapper after the placement and routing step. It indicates the value of the cost function for the achieved mapping.

Routing Control Parameters. There are two parameters controlling the nearest neighbor routing algorithm described in section 7.3.2.2.

MaxRoutChan *integer*

This parameter denotes the maximal number of routing channels as discussed in section 6.3.2.3. Although it is in its nature an architecture property, it affects the routing algorithm, limiting the use of nearest neighbor connections.

RoutingDepth *integer*

This parameter limits the search range for the rDPUs to be connected by nearest neighbor routing. The effect of this parameter is discussed in secti on7.3.2.2.

Scheduling Parameters. A couple of parameters are used by the scheduler, being extracted from the hardware file (see section 7.1.2) and forwarded, contained in the intermediate file. These are mostly global delay times for communication resources. However, some parameters refer to a controller unit as found in the KressArray-I architecture [Kres96]. These are only relevant for architectures featuring such a unit. Most parameters are described in section 7.1.2.4.

NC_Delay_OnChip *integer*
NC_Delay_OffChip *integer*
NC_Delay_Torus *integer*
NC_Delay_Base *integer*
Rout_Delay_Base *integer*
Rout_Delay_Step *integer*

Those parameters are used to calculate the delay of paths made up of nearest neighbor connections. They are discussed in detail in section 7.1.2.4.

RCB_Multicast *integer*
RCB_Delay_Base *integer*
RCB_Delay_Step *integer*

These parameters deal with row and column backbuses. The first one denotes, if the backbuses have multicast capability, while the other two are used to calculate the delay for connections established over row and column backbuses. These issues are discussed in the according part of section 7.1.2.4.

Delay_Dpu_Cont *integer*
Delay_Cont_Dpu *integer*
Delay_Cont_Mem *integer*
Delay_Mem_Cont *integer*

These parameters are relevant for a KressArray featuring a control unit for central handling of global bus transfers, as in the KressArray-I. The first two delay times apply to transfers between the controller and rDPUs over the global bus. They are described in sectio n7.1.2.4. The last two parameters apply to an architecture with

an extern memory attached to the control unit. They denote the time for a transfer from a controller register to the memory (*Delay_Cont_Mem*) and vice versa (*Delay_Mem_Cont*) respectively. Thus, for example a transfer from an output rDPU over an I/O port implemented by the global bus to the memory, the delay would be calculated to *Delay_Dpu_Cont* + *Delay_Cont_Mem*.

FIFO_Delay *integer*

In the KressArray-I structure, a FIFO queue is installed between the memory and the controller to buffer memory data (see [Kres96]). If the value to this parameter is non-zero, it denotes the delay of a transfer from the FIFO to the controller. A zero value denotes, that there is no such FIFO installed.

RegisterFileLimit *integer*

This parameter applies also to a control unit as the one in the KressArray-I architecture. It denotes the size of the controller register file [Kres96], which is used amongst other purposes to store intermediate results.

CLOCK *integer*

The *CLOCK* parameter specifies the length of a clock cycle in time steps. Thus, the delay times for transfers and other operations need not be multiples of the clock cycle time.

A.1.3 Communication Resource Section

This section describes the communication resources featured by the architecture. As the global bus is always assumed to exist, the resources defined in this section comprise row and column backbuses and nearest neighbor connections. There is only one type of statement in the communication resource section.

A.1.3.1 The COMRC Statement

The COMRC statement provides a unified way to declare the available communication resources. The statement consists of the keyword followed by an appropriate resource description:

COMRC <Resource Description> **;**

The form of <Resource Description> depends on the type of resource to be described. In the following, the descriptions for nearest neighbor connections and row column backbuses is demonstrated in more detail.

Nearest Neighbor Connections. A description for a nearest neighbor connection looks like this:

COMRC <Connection> <Torus> *OnChipCost OffChipCost* **;**

The integer values *OnChipCost* and *OffChipCost* are the cost parameters for the according connection as described in section 7.3.3.3.

There are six possibilities for <Connection>, denoting the different types of connections addressed in section 6.3.2.1:

SimplexN
SimplexE
SimplexS
SimplexW
HDuplexH
HDuplexV

The first four keywords denote unidirectional (simplex) connections heading North, East, South and West, respectively. That means, that e.g. a SimplexS connection is the output for a rDPU to the south and the input from the north for the rDPU south from the first one. The last two keywords denote bidirectional (half duplex) connections of horizontal and vertical type respectively.

The <Torus> declaration describes the torus structure for the connection. There are four possible keywords:

NONE
SAME
NEXT
PREV

These keywords resemble the torus types defined in section 6.3.2.2.

Row and Column Backbuses. There are two syntax forms allowed for the declaration of a row or column bus:

COMRC <RCBus> *SegLength MaxWriter FirstSeg CostBase CostStep CostWrite* **;**
COMRC <RCBus> *SegLength MaxWriter CostBase CostStep CostWrite* **;**

The integer parameters *SegLength* and *FirstSeg* correspond to those used in the description of backbus segmentation in section secti on6.3.3.1. In an analog way, *MaxWriter* denotes the bus capacity as discussed in section 6.3.3.2. If *SegLength* equals zero, the segment length is assumed to be as long as the according array dimension. In other words, the bus will span the whole array unsegmented. In the second form of syntax, where *FirstSeg* is left out, it is assumed to equal zero, making the first segment as long as *SegLength*. The cost parameters *CostBase*, *CostStep* and *CostWrite* are used for the calculation of the cost function for backbus connection according to section 7.3.3.2.

The <RCBus> declaration determines the type of the backbus. There are two possibilities:

RowBus
ColBus

These keywords declare a row or column backbus according to the definition in section 6.3.3.

A.1.4 Array Header Section

This section keeps information about the following expression tree, which stems from the DPSS for the KressArray-I architecture and is kept for compatibility reasons. Further, information about heterogeneous array structures is stored here. There are two obligatory statements (ARRAY and FLAGS) concerning the header data, and one optional type of statement (ARCAP) for the definition of heterogeneous structures.

A.1.4.1 The ARRAY and FLAGS Statements

These two statements must appear in the section in the order given.

ARRAY *Number* **'***Name***' ;**
FLAGS <Flags> <Flags> <Flags> <Flags> **;**

Both the ARRAY and the FLAGS statement are normally generated by the ALE-X compiler. The integer *Number* is the consecutive number of the netlist for datapaths featuring several of those. The string *Name* denotes a name for the netlist created by the compiler as "a" followed by *Number*, e.g. "a0". The <Flags> declarations each contain four binary digits, thus there are four groups of four flags in the statement. Those flags are kept for compatibility reasons and control the status communication of the KressArray-I to the data sequencer, as described in [Kres96].

A.1.4.2 The ARCAP Statement

One or several ARCAP statements can be used together with OPREQ statements (see below in section A.1.5.10) to define heterogeneous array structures. The ARCAP statement defines the capabilities of rDPUs, while the OPREQ statement applies to operators and functions. The syntax of the ARCAP statement is as follows:

ARCAP *Mask* **{** *BaseY* **,** *StepY* **,** *LimitY* **} {** *BaseX* **,** *StepX* **,** *LimitX* **} ;**

The value *Mask* is implemented as an integer value, as the other parameters. All the parameters correspond to those used in the description of heterogeneous array concepts in section 6.4.2.

A.1.5 Subnet List Section

The subnet list section contains the expression tree, or netlist, of the datapath. According to the KressArray-I terminology, one subnet is a connected datapath to be mapped onto the array. Thus, one intermediate file may consist of several subnets. Each subnet may again contain several blocks, which typically contain several expressions. This leads to a larger number of statement types, as there are also special statements for organization needed.

A.1.5.1 The Subnet Header Statements

Each subnet starts with a header containing three statements, one SUBNT, one or more HANDL, and one PIPEL statement, which appear in this order. Their syntax is as follows:

SUBNT *Number* **'***Name***' ;**
HANDL '*ScanwindowName***' [** *Yoffset* **] [** *Xoffset* **] ;**
PIPEL *Pipelineflag* **;**

While *Number* is a consecutive number given by the ALE-X compiler, the *Name* string is taken over from the subnet name given after the "rALUsubnet" keyword in the ALE-X file, as described in section 7.1.1. The information provided by the HANDL statements is taken in an analog way from the scanwindow description of the same file. There is one HANDL statement for each scanwindow, bearing the scanwindow name in the string *ScanwindowName*. The movement of the scanwindow, which is described in the ALE-X file by the "HandleOffset" statement, is taken over into the integer values *Yoffset* and *Xoffset*, corresponding to the according values in the ALE-X statement. The value *Pipelineflag* is either zero or one, and is set to one by the ALE-X compiler, if pipelining is possible for this datapath.

A.1.5.2 The BLOCK Statement

This statement initializes a block containing expressions, which has been generated from a basic block of the original ALE-X program by the compiler. It has the following syntax:

BLOCK <Type> **;**

The <Type> declaration is one of four keywords describing the possible block types supported by the ALE-X compiler, plus one type for an unknown block. The possible keywords for <Type> are:

ASSIGN
WHILE
DOWHILE
UNKNOWN

While the first keyword initializes an assignment block (compare section 6.1.1), the next two keywords denote a loop body for either a while or a do-while loop (see section 6.1.4). The last keyword should not appear, but has been adopted for safety reasons.

A.1.5.3 The VAREF and VDEFN Statements

These two statements occur in a block and define an input port (VAREF) or an output port (VDEFN) of the datapath. The syntax of both statements is similar:

VAREF $*Expression* **'***Variable***' {** <Porttype> **} [** *Ypos* **] [** *Xpos* **]**
'*ScanwindowName***'** *Delay Group* **;**
VDEFN $*Expression* **$***From* **'***Variable***' {** <Porttype> **} [** *Ypos* **] [** *Xpos* **]**
'*ScanwindowName***'** *Delay Group* **;**

The I/O declarations VAREF and VDEFN are handled internally the same way as other expressions, like operators and functions. Thus, they bear an identification number *Expression*, which is unique throughout the file. By this number, the ports may be referenced by other expressions. Additionally, the output statement VDEFN bears the number *From* of the expression in the block, which produces the output value to be sent on the port. To keep compatibility with the KressArray-I DPSS, the I/O is assumed to be handled over scanwindows [HHRS91]. Thus, each VAREF and VDEFN expression is associated with a variable in the ALE-X file (see section 7.1.1) with the name indicated by *Variable*. This variable is located in the scanwindow *ScanwindowName* at the position indicated by *Ypos* and *Xpos*.

The parameters *Delay* and *Group* are used by the scheduler. They resemble the time for a transfer over the according port and the port group. The port group describes the extern attachment of edge ports to memory systems, as described in section 6.4.4.

The <Porttype> declaration describes the way the I/O connection is implemented. There are three basic forms of this declaration, according to the three port types described in section 6.4.3:

- Declaration of a port over the global bus
- Declaration of an edge port
- Declaration of a port inside the array

In the discussion of these declarations, the braces "{" and "}" are included for better readability.

Declaration of a Port Over the Global Bus .This type of declaration has the following form:

{ G }

Declaration of an Edge Port. An edge port declaration is more complex, as there are different possibilities for the range of the port on the according edge (see section 6.4.3.2). There are three different forms of such a declaration:

{ <PortSide> **}**

The <PortSide> declaration denotes the edge where the port is located, according to section 6.4.3.2. It consists of one of the letters N, E, S, W, denoting the northern, eastern, southern, or western edge of the array. The above form of the declaration with the <PortSide> alone denotes a port, which can be located by the mapper on the according edge without any restrictions to the exact position (compare section 6.4.3.2).

{ <PortSide> **[** *FirstPos* **] [** *LastPos* **] }**

This declaration allows the mapper to locate the port on the edge described by <PortSide> only in the range denoted by *FirstPos* and *LastPos*. These parameters have the same meaning as in section 6.4.3.2.

{ <PortSide> **[** *ThePos* **] }**

This declaration specifies a port on the edge <PortSide>, which has to be positioned exactly at the location denoted by *ThePos*, leaving no range for the mapper (see again section 6.4.3.2).

Declaration of a Port Inside the Array. This type of declaration has the following form:

{ [*Yposition* **] [** *Xposition* **] }**

The coordinates Yposition and Xposition specify the position of the rDPU, which serves as the source or sink for the data words, as described in section 6.4.3.3.

A.1.5.4 The CONST and CONOP Statements

These two statements are used to define constants in the datapath. These appear as expressions, which can be referenced by other expressions. The syntax is as follows:

CONST $*Expression Value* **;**
CONOP $*Expression Value* **;**

Both statements initialize the expression with the ID *Expression* as a constant with the float or integer value *Value*. While a CONOP statement is implemented by a rDPU, which will constantly deliver the same value, a CONST constant is implemented as an immediate operand at the referencing expression, thus being configured at the rDPU of this expression.

A.1.5.5 The OPERA and FCALL Statements

The OPERA and FCALL statements are typically the most frequently found ones, as they describe an operator (see section 7.1.2.2) or a user defined function (see section 7.1.2.3) of the original ALE-X code. The syntax for these statements are as follows:

OPERA $*Expression* '*Name*' '*Symbol*' **($***Oprnd* ... **$***Oprnd* **)** *Delay* **;**
FCALL $*Expression* '*Name*' **($***Oprnd* ... **$***Oprnd* **)** *Delay* **;**

Both expressions are identified by their ID *Expression*. The operator or function implemented by the expression is denoted by *Name*, which must be a valid name of an operator (for OPERA) or function (for FCALL) in the ALEX-compiler hardware file (see section 7.1.2). As described in the according section 7.1.2, operators and user defined functions differ in that operators bear an additional symbol, which is used in the ALE-X code, while functions are addressed by their function name. The symbol of an operator is described by the string *Symbol*.

Both operators and functions may reference one or more other expressions as their operands. These operands appear as the *Oprnd* expressions in the statement. Currently, there is a maximum of six operands allowed. Another common parameter is the execution time *Delay* of the according operation, which is forwarded to the scheduler.

A.1.5.6 The CNDEX Statement

This statement is only found in blocks of the type WHILE or DOWHILE. It appears exactly once in those blocks. The syntax is as follows:

CNDEX $*Expression* **;**

The parameter *Expression* identifies the (previously declared) expression of the block, which represents the loop condition. This means, the loop will be iterated as long as the loop condition evaluates to a non zero value or to true. This expression is used by loop implementations using the central control unit for loop control, as described in [Kres96].

A.1.5.7 The BLOOP Statement

The BLOOP statement defines a cycle in the datapath, according to the technique described in section 6.1.3. This statement has two possible forms:

BLOOP $*Expression* **$***Backloopexpression* **;**
BLOOP $*Expression* **$***Backloopexpression InitValue* **;**

Basically, the BLOOP statement initializes a pseudo expression with the identifier *Expression*. As discussed in section 6.1.3, the value of the expression is really taken from the expression *Backloopexpression*, which is defined later on in the file. Thus, the datapath appears linear, but has a cycle implemented by the BLOOP

expression, which can be referenced as any operator or function. Due to the implications caused by the transport-triggered execution model described in section 6.1.3, it is necessary to initialize the operand inputs of those expressions, which reference the pseudo expression, with a suitable value. The initialization value can be given in the second form by the integer or float value *InitValue*. In the first form, which lacks this parameter, the initialization value is assumed to be zero.

A.1.5.8 The BUDDY Statement

The BUDDY statement is needed to describe an operator with double output, as discussed in section 6.2.4. It has the following syntax:

BUDDY $*Expression* **$***From* **'***Name***' '***Symbol***'** *Delay* **;**

Basically, the BUDDY statement initializes a new expression with the ID *Expression*, which has a name (*Name*) and a symbol (*Symbol*) like an ordinary operand. However, the BUDDY expression is bound to another, previously defined expression *From*, with which it shares the operands and, later in the mapping, the rDPU. Thus, another output is added to the expression *From*, which may be referenced by other expressions like *From* itself. The expression *From* resembles therefore an operator, which generates two different outputs in one execution, e.g. division with remainder. As the BUDDY operator may have a different execution time than the main expression, also a *Delay* parameter is provided to specify this time.

A.1.5.9 The GROUP Statement

This statement is used in the context of double ALU rDPUs, as being described in section 6.2.3. For the GROUP statement, two different forms are possible:

GROUP $*Expression BitMask* **;**
GROUP 'Name**'** *BitMask* **;**

The purpose of this statement is the setting of the operator bitmask, which determines, how more than one operator can be mapped onto one rDPU, as described in section 6.2.3. In the first form of the statement, the bitmask is set for one single expression *Expression* to the value *BitMask*. In the second form, the mask is set for all operators or functions with the name denoted by the string *Name*.

A.1.5.10 The OPREQ Statement

This statement is the pendant of the ARCAP statement (see above in section A.1.4.2). It is used to define heterogeneous architectures on the operator side, according to the technique described in section 6.4.2. Like the GROUP statement described above, the OPREQ statement allows two forms:

OPREQ $*Expression BitMask* **;**
OPREQ 'Name**'** *BitMask* **;**

According to the approach in section 6.4.2, the OPREQ statement sets the requirement bitmask of operators to the value *BitMask*. The first form allows to set the mask for the single expression *Expression*, while the second form sets *BitMask* for all operators or function with the name *Name*.

A.1.6 rDPU Mapping Section

The last section of an intermediate file contains the mapping information, comprising the assignment of operators to rDPUs, and the assignment of connections to communication resources. The mapping section relies on the subnet list section, as the operators there are referenced by the statements describing the mapping. There are two types of statements in this section, the OPDPU statement, describing an operator mapped onto a rDPU, and the RTDPU statement, which describes routing connections of a rDPU. There may exist several RTDPU statements for one rDPU position. More than one OPDPU statements for the same rDPU exist only for architectures with double output or double ALU rDPUs. If a rDPU has only RTDPU statements, it is a routing element, which bears no operator.

A.1.6.1 The OPDPU statement

The OPDPU statement describes the assignment of a specified expression to a certain rDPU position. The syntax is as follows:

OPDPU [*Y* **] [** *X* **]** <Opfixing> **$***Expression* **%**<Con> ... **%**<Con> **%**<CG> **;**

The statement describes the expression *Expression* to be mapped onto the rDPU at the position with the coordinates *Y* and *X*. The <Opfixing> declaration contains information about the fixing of the expression to the rDPU, as described in section 7.3.4. For the maximal six operands of the expression, a constant number of six connection declarations <Con> are present. These connection declarations describe the communication resources, which are used to input the according operand. Unused operands are marked as free (see below). A final connection declaration <CG> describes the source for the global bus output. The different declarations will be described in more detail in the following.

Fixing Declaration. The format of the fixing declaration is described in the following. It should be noted, that the angular brackets "<" and ">" are characters as part of the syntax, and do not denote another declaration.

<*Yfix***,***Xfix***,***f0f1f2f3f4f5fg***>**

The special format of the declaration does not allow blanks, so they are also left out from the definition. There are nine parameters specifying the fixing, *Yfix*, *Xfix*, *f0, ..., f5*, and *fg*. Each single parameter may resemble either the letter "V" for variable, or "F" for fixed. Using the terminology of section 7.3.4, the parameters *Yfix* and *Xfix* denote, if the operator position is "Y-Fixed" or "X-Fixed" respectively, with the state of "Fixed" being represented by both parameters being "F". The parameters *f0, ..., f5* denote, if the corresponding operand connection is fixed. A fixed connection may not be ripped up by the mapper. In a similar way, *fg* describes the fixing for the global bus output.

Operand Connection Declarations . The <Con> declarations use abbreviations to indicate the communication resource by which the data is transferred to the according input. All possible abbreviations consist of two characters. The following possibilities exist:

FR

This denotes an unused (free) ALU input.

IM

The operand to this input is an immediate, constant value, which is configured into the rDPU rather than transferred.

GL

This input receives data over the global bus.

N*x*
W*x*
S*x*
E*x*
(where *x* is a digit)

The according input receives its data over the nearest neighbor connection number *x* from the specified direction. It should be noted, that the direction describes, where the data comes from, e.g. %N0 indicates, that the data for this input is received from the rDPU to the north of the current one, using connection number zero.

R*x*
C*x*
(where *x* is a digit)

This indicates, that the according input receives its data from the row (R*x*) or column (C*x*) backbus number *x*.

Global Bus Connection Declaration. The format of the <CG> declaration is analog to the operand connection declarations. In contrast to them, the <CG> declaration describes an output, and there are only two possible connections:

FR

This means, that the global bus of the according rDPU is not used to transmit a result.

A*x*
(where *x* is a digit)

This indicates, that the result from ALU output number *x* is transmitted over the global bus. The number *x* is normally zero, except for architectures featuring double output or double ALU rDPUs, where the second output is numbered one.

A.1.6.2 The RTDPU statement

This statement is used to describe an additional output for the operator result, or an internal routing connection. The syntax of the RTDPU statement is as follows:

RTDPU [*Y* **] [** *X* **] <<**Routfixing**>> %**<Source> **%**<Destination> **;**

Basically, the RTDPU statement defines a link from the connection declarations <Source> to the connection <Destination> at the rDPU at the position indicated by *Y* and *X*. The <Routfixing> declaration is a reduced form of the one used at the OPDPU statement and is put between angular brackets. It consists of one single letter, either "V" or "F", depending on if the connection is allowed to be changed by the mapper ("V") or not ("F").

The <Source> and <Destination> connection declarations are similar to those used in the OPDPU statement. However, the set of allowed connection types differs. For the <Source> declarations, the following possibilities exist:

A*x*
N*x*
W*x*
S*x*
E*x*
(where *x* is a digit)

For the <Destination> declaration, there are the following possible connections:

GL
N*x*
W*x*
S*x*
E*x*
(where *x* is a digit)

It should be noted, that although the <Source> and <Destination> declarations share common connections, the nearest neighbor connections in <Source> describe an input, while those in <Destination> denote an output connection. Thus, if e.g. <Source> were %N0 and <Destination> were %S0, then the link would route a data word from the rDPU to the north to the one south of the current one.

A.2 Mapping Examples

In this section, two example mappings are shown produced by the MA-DPSS mapper tool. The first datapath resembles a simple multiplication datapath for two by two matrices. The second example resembles a complex image processing application known as the symmetric nearest neighbor filter. The two examples are provided with the execution times of the mapper to give an estimation of the time for synthesis.

A.2.1 Matrix Multiplication

The example datapath shown in this section calculates the result of a matrix multiplication as the product $M = A \times B$ with A, B and M being two by two matrices. The datapath implements the following formula:

$$A = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} B = \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}$$

$$M = \begin{bmatrix} m_{00} & m_{01} \\ m_{10} & m_{11} \end{bmatrix} = A \times B = \begin{bmatrix} a_{00}b_{00} + a_{01}b_{10} & a_{00}b_{01} + a_{01}b_{11} \\ a_{10}b_{00} + a_{11}b_{10} & a_{10}b_{01} + a_{11}b_{11} \end{bmatrix}$$

An according ALE-X description of this datapath is shown in figure A-1. The ALE-X code features three scanwindows with four variables each, thus giving a total of eight input and four output ports. The according mapping onto a 16-rDPU KressArray with two nearest neighbor connections on each side is pictured in figure A-2. The mapping shown has been conducted assuming that the assignments of the variables of the ports is fixed in the order shown in the figure. The input data a_{yx} and b_{yx} enter the array from the left and the right side respectively, while the output data m_{yx} leave the calculation to the southern side.

The mapping has been conducted on a dual-PentiumII system with 200 MHz running Linux. The computation took 1:28 minutes. On a system with a Celeron 366 running Linux, the same mapping took 38 seconds.

```
rALUsubnet SubNet_0 (MatA,MatB,MatOut)    // Subnet header

ScanWindow MatA                           // Scanwindow description
{     int a00 at [0][0];
      int a01 at [0][1];
      int a10 at [1][0];
      int a11 at [1][1];
      HandleOffset [0][2];
};
ScanWindow MatB
{     int b00 at [0][0];
      int b01 at [0][1];
      int b10 at [1][0];
      int b11 at [1][1];
      HandleOffset [0][2];
};
ScanWindow MatOut
{     int out00 at [0][0];
      int out01 at [0][1];
      int out10 at [1][0];
      int out11 at [1][1];
      HandleOffset [0][2];
};

{     out00=(a00*b00)+(a01*b10);          // Datapath description
      out01=(a00*b01)+(a01*b11);
      out10=(a10*b00)+(a11*b10);
      out11=(a10*b01)+(a11*b11);
}
```

Figure A-1: ALE-X description of a datapath for matrix multiplication (two by two matrices)

A.2.2 Symmetric Nearest Neighbor (SNN) Filter

The symmetric nearest neighbor (SNN) filter is an application from the area of image pre-processing.The rather complex datapath is illustrated in figure A-3. The algorithm processes image data consisting of e.g. 24 bit pixel values with a red, green and blue component. The new value of a pixel is calculated using its old value and the eight surrounding pixels. The eight neighbor pixels are separated in four groups of two pixels, according to the figure. From each group,

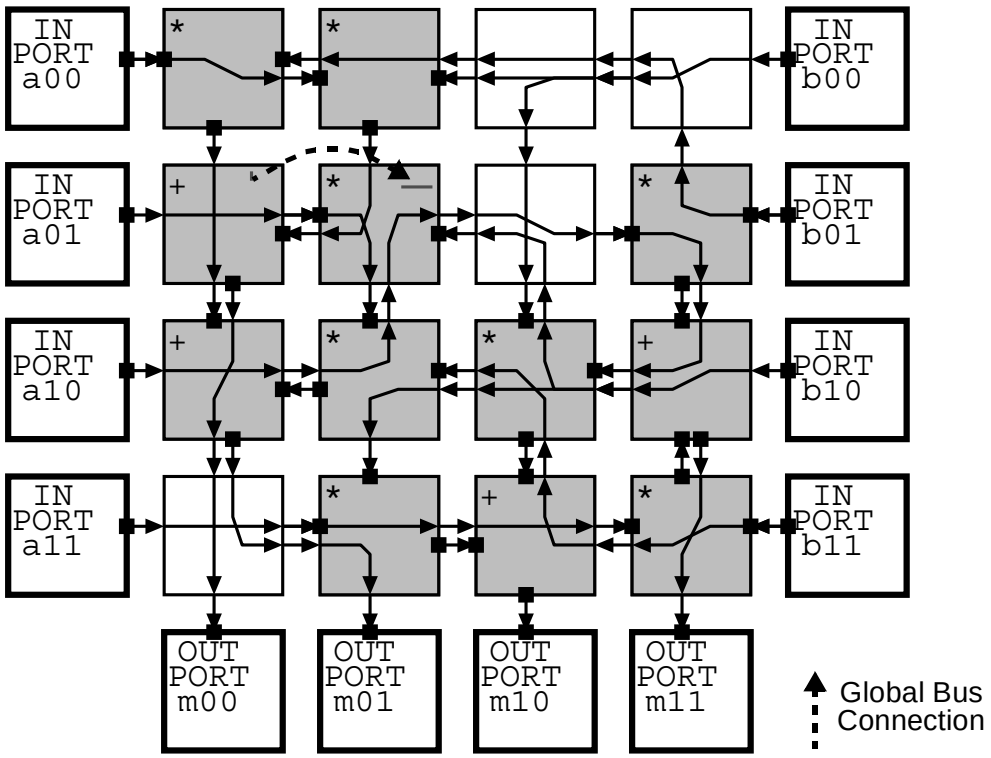

Figure A-2: Mapping of the matrix multiplication datapath with fixed assignments of variables to ports.

one pixel is selected, which shows the least difference in the color values to the center pixel. The color difference is calculated by the distance function *dist* shown in the figure. Finally, the new value for the central pixel is given by the average of the values of the four selected pixels. This average is calculated separately for the red, green, and blue component.

For the sake of simplicity, the ALE-X description of this complex datapath is omitted. The mapping of the SNN filter is shown in figure A-4. The datapath features nine inputs and one output (the one at the bottom). The mapping took 48 minutes on a dual-PentiumII 200 system and 19 minutes on a Celeron 366 system, both machines running Linux.

A.3 Statistic Data Produced by the Analyzer

The statistics collected by the analyzer subsystem described in sectio n8.3 are the basis for the reasoning process employed to generate design suggestions. New statistics can be added easily by providing an according plug-in, which can either use existing data to produce a value by a new computation, or use extern sources to add information to the database. New plug-ins integrate smoothly in the

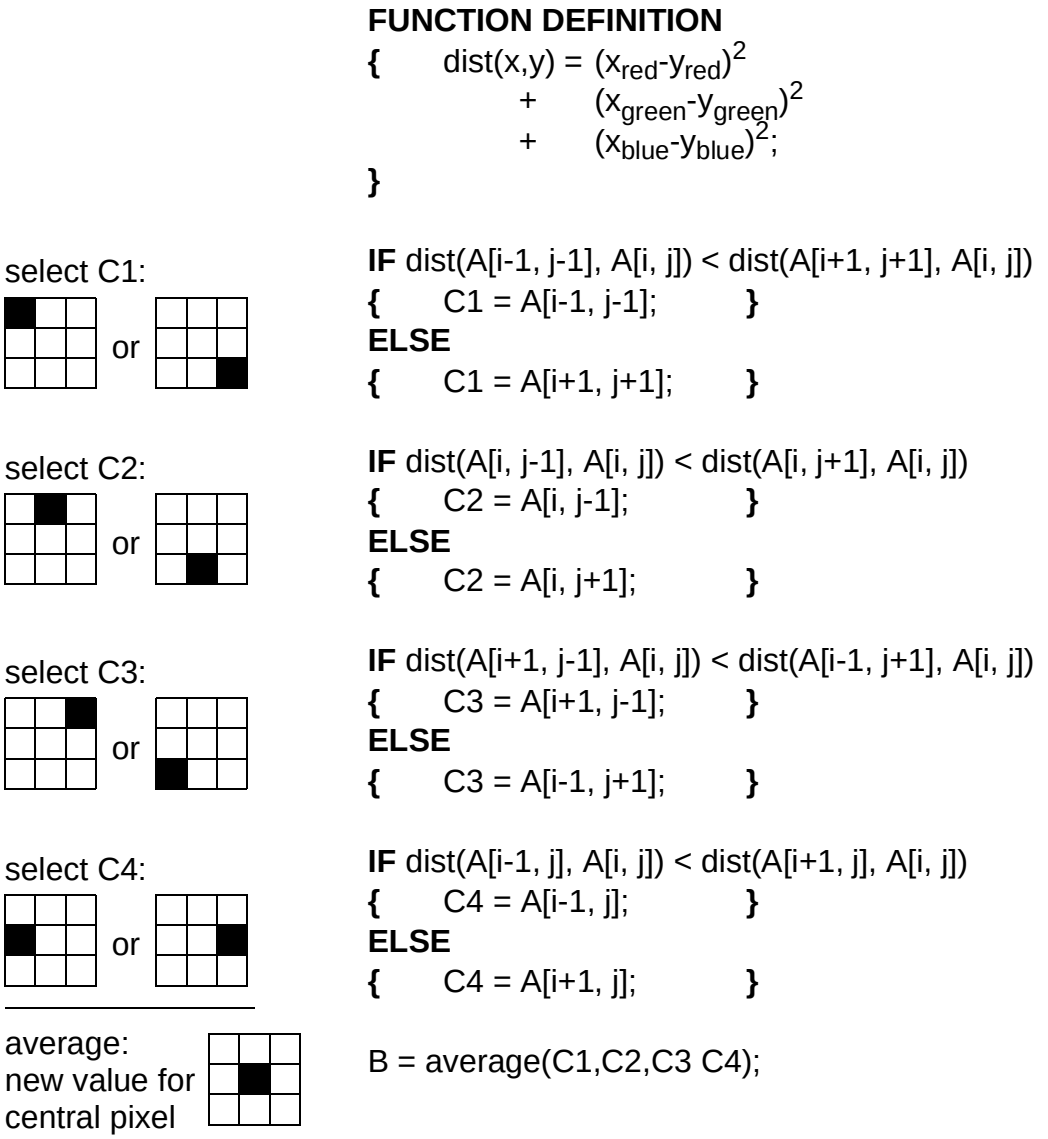

Figure A-3: Illustration of the symmetric nearest neighbor filter algorithm

system, as they can be invoked easily by a controller plug-in. The data is then available for a following call of the reasoning engine. Some examples for typical statistics are given below, together with a brief characterization of the according measure.

Array Dimensions, Available Communication Resources, Operator Placement, Connection Routing and Array Capabilities These data are retrieved directly from the intermediate file and are typically the base for other measures.

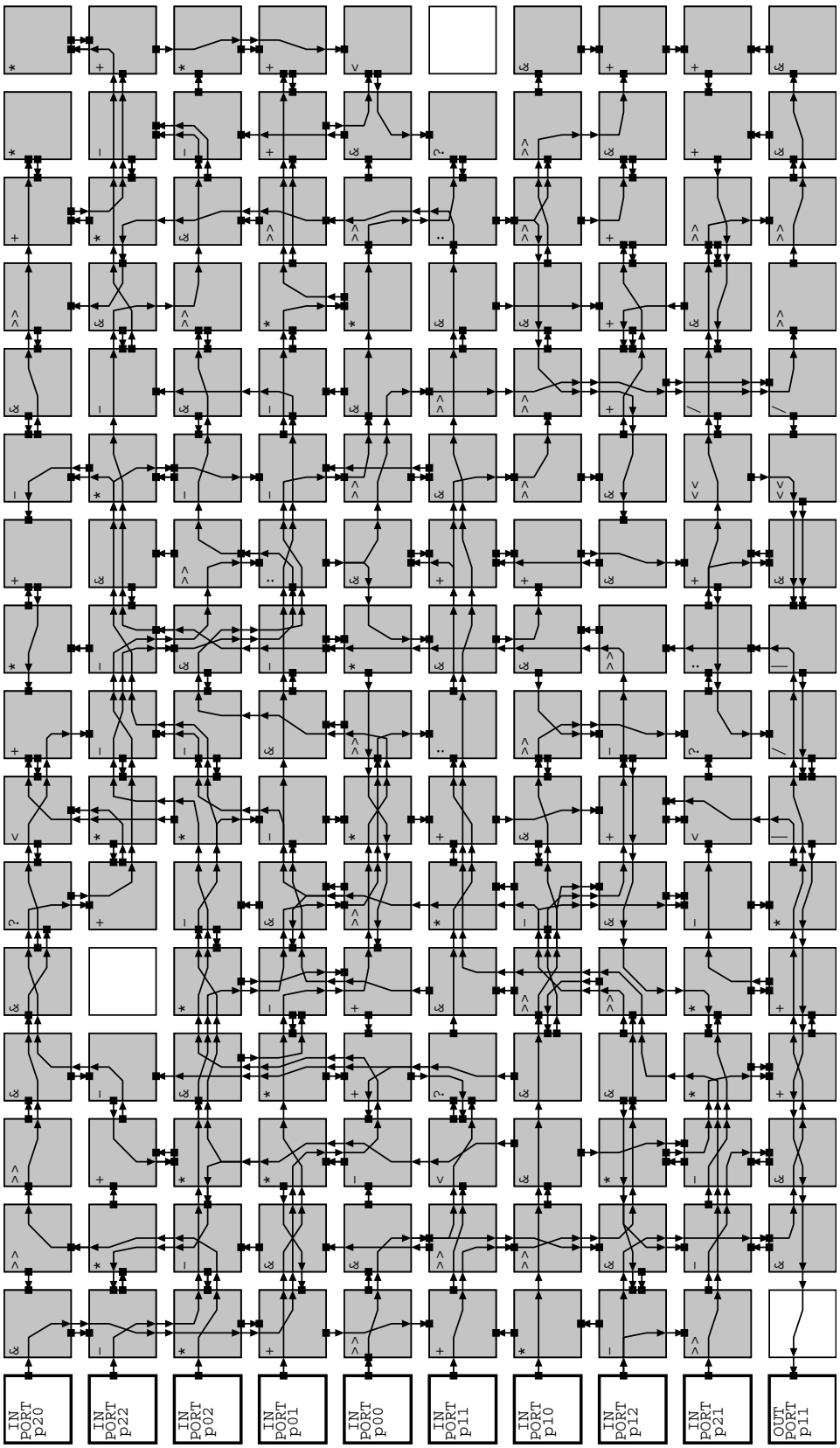

Figure A-4: Mapping of the symmetric nearest neighbor filter algorithm

Communication Resource Usage. This measure is the ratio of used communication resources to available communication resources, and is generated separately for horizontal and vertical connections and for nearest neighbor connections and backbuses. This value depends typically on the types of available communication resources, the prevailing direction of data transfer, and the complexity of the datapath. Besides the separate values for horizontal and vertical connections, also average values for the total usage of backbuses and nearest neighbor connections respectively are generated.

Main Dataflow Direction . This value is computed as the difference between the horizontal and vertical communication resource usage. Depending on it being positive or negative, it is a measure for the prevailing direction (horizontal or vertical) of data.

Number of Global Bus Connections. The serial global bus is used as a last resort, if no other routing resource is available to connect two rDPUs. Thus, the number of global bus connections is a direct measure for the suitability of the architecture to the datapath. This value is expressed as the ratio of the number of bus connections to the total number of connections.

Average Operator Fanout. This value allows conclusions about the reasonability of a broadcasting communication resource, i.e. a row or column backbus, instead of a nearest neighbor connection.

Performance Estimation. This value is produced by the MA-DPSS scheduler (see section 7.4). It depends on the number of global bus connections and the lengths of the routing paths.

Datapath Complexity. The computation of this measure is described in more detail in section 8.3.1. It allows general conclusions about the requirements of communication resources for the datapath.

A.4 Example Rule Set

In this section, a selection of possible exploration rules for six example design actions are presented, along with the definition of the according linguistic variables. Although this set of actions is kept quite small for illustrational purposes, it may well be used for simple explorations, as shown in section A.5.

However, depending on the exploration objective, other rules may probably be needed. As described in section 8.4.5, new rules for other actions can be easily composed using the FOOL editor [NLLH95].

The example ruleset is aimed for minimizing the global bus transfers, with the objective to remove them totally, if possible. This objective may be reasonable for creating an embedded KressArray architecture without the possibility for a sophisticated control unit.

In the following subsections, first the definitions of the linguistic variables are described, followed by the six rule sets for the selected design actions.

A.4.1 Linguistic Variable Definitions

The definitions for the linguistic variables and their adjectives, having been found by experiments, are given in the following. The selection of variables has been restricted to those measures and adjectives actually used in the rules. It should be noted, that there are more adjectives, which are reasonable, and that other rules may of course require different statistic data. The variables, which provide the base for the rules, and their corresponding adjectives, are:

- The percentage of global bus connections related to all connections ("Glob_Use"), with the adjectives "high" and "not_high" (Figur eA-5).
- The horizontal and vertical connection usage ("H_Use", "V_Use) in percent, each with the adjectives "high" and "low" (Figur eA-6).
- The total connection usage ("Total_Use") in percent, with adjectives "high" and "not_high" (Figur eA-7).
- The usage of the backbuses in percent ("BBus_Use") with the adjective "high" (Figure A-8).
- The number of horizontal and vertical nearest neighbor connections ("Num_HNN", "Num_VNN"), each with the adjective "low" (Figure A-9).
- The main dataflow direction ("Data_Dir") with the adjectives "horizontal" and "vertical" (Figur eA-10).
- The datapath complexity ("Complexity") with the adjective "not_high" (Figure A-11).
- The average operator fanout ("Fanout") with the adjective "high" (Figure A-12).

An additional linguistic variable is the output of a ruleset, consisting of the ranking for the according design action. This output value ("Ranking") comprises the two adjectives "low" and "high" (Figure A-13).

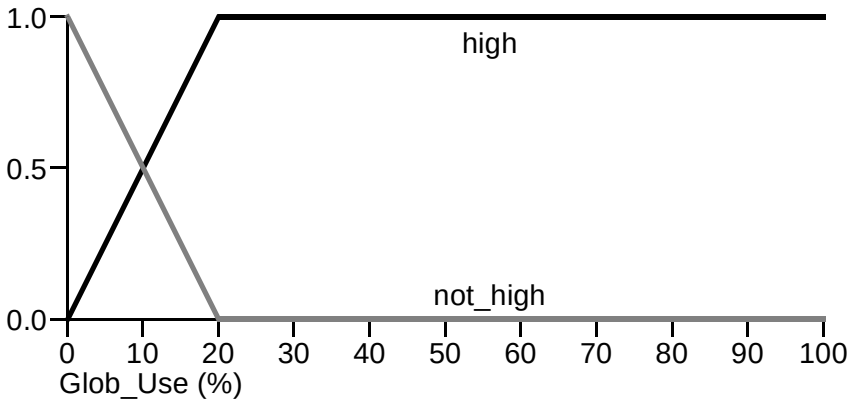

Figure A-5: Example Ruleset: Linguistic Variable "Glob_Use"

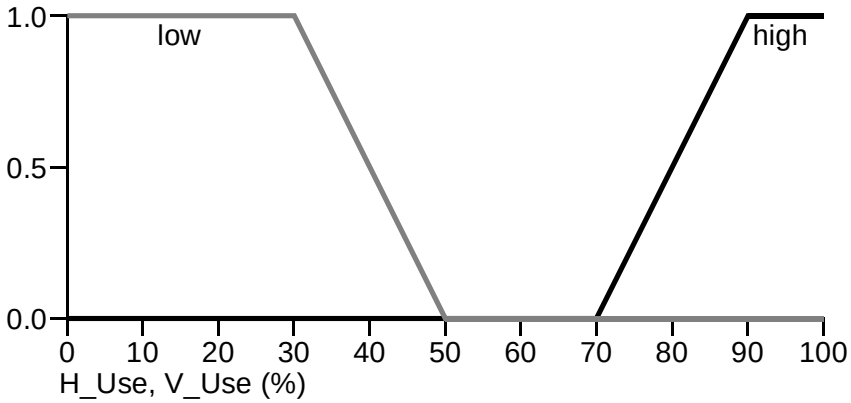

Figure A-6: Example Ruleset: Linguistic Variables "H_Use", "V_Use" (identical definitions of adjectives)

A.4.2 Rule Definitions for Design Actions

In this section, the definitions for an example set of design actions are presented. The set of actions is kept small for illustrational purposes. Also, the rules given apply to the example exploration objective of minimizing the global bus connections, as stated above. The six design actions considered are:

- Adding a horizontal bidirectional nearest neighbor connection
- Adding a vertical bidirectional nearest neighbor connection

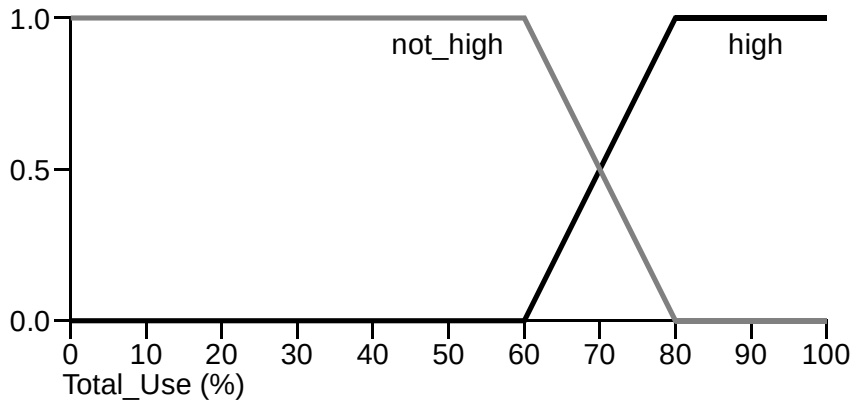

Figure A-7: Example Ruleset: Linguistic Variable "Total_Use"

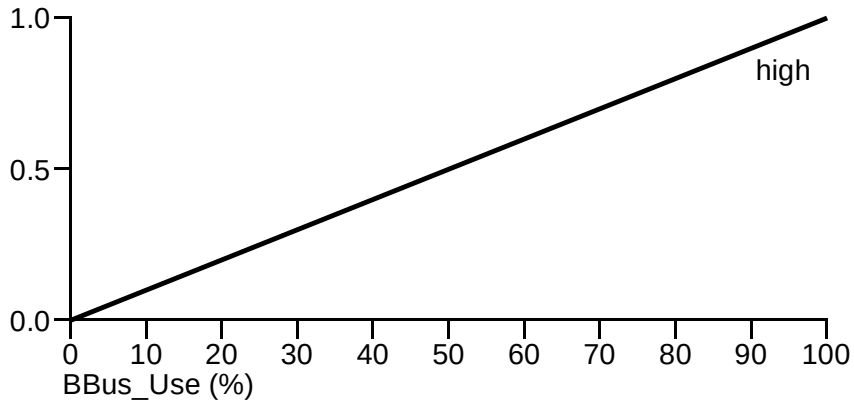

Figure A-8: Example Ruleset: Linguistic Variable "BBus_Use"

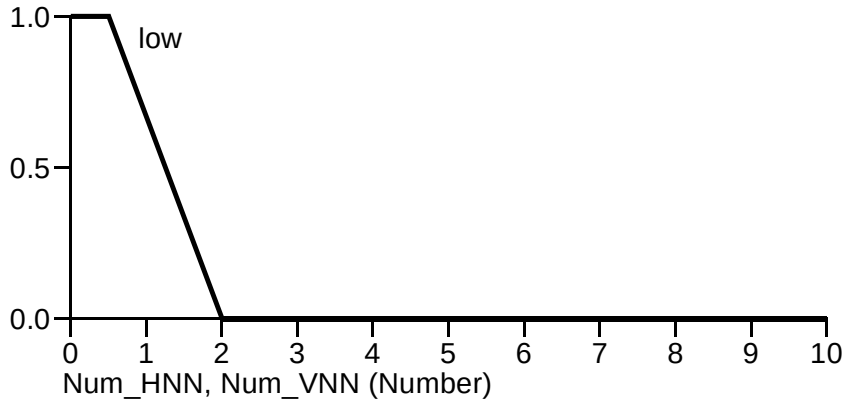

Figure A-9: Example: Linguistic Variables "Num_HNN", "Num_VNN" (identical definitions of adjectives)

- Removing a horizontal nearest neighbor connection

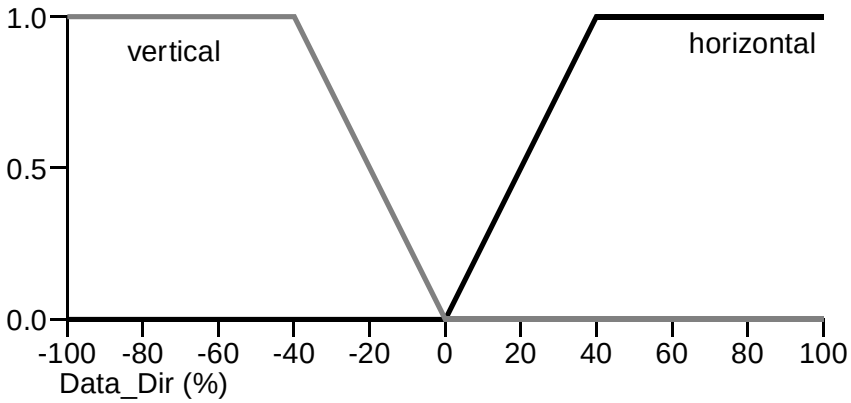

Figure A-10: Example Ruleset: Linguistic Variable "Data_Dir"

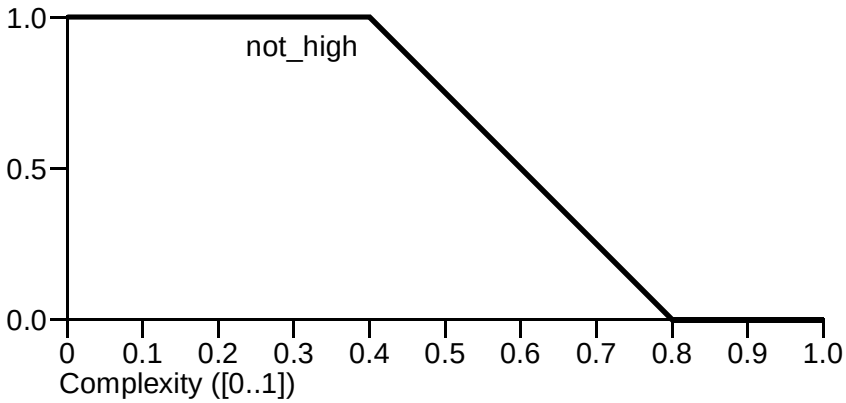

Figure A-11: Example Ruleset: Linguistic Variable "Complexity"

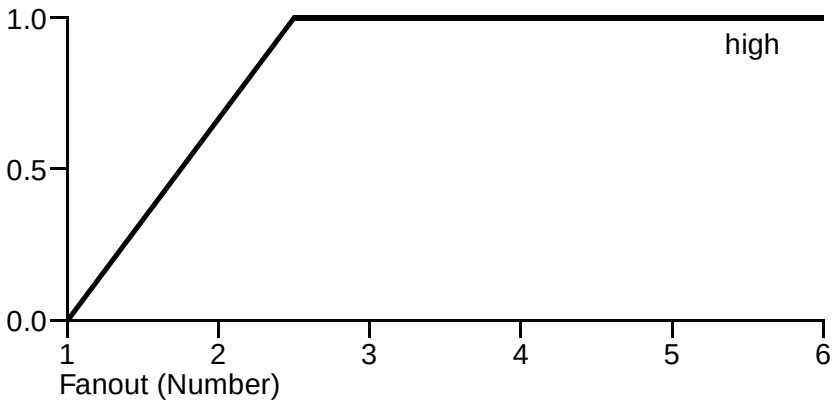

Figure A-12: Example Ruleset: Linguistic Variable "Fanout"

- Removing a vertical nearest neighbor connection
- Changing a horizontal neighbor connection to a row backbus

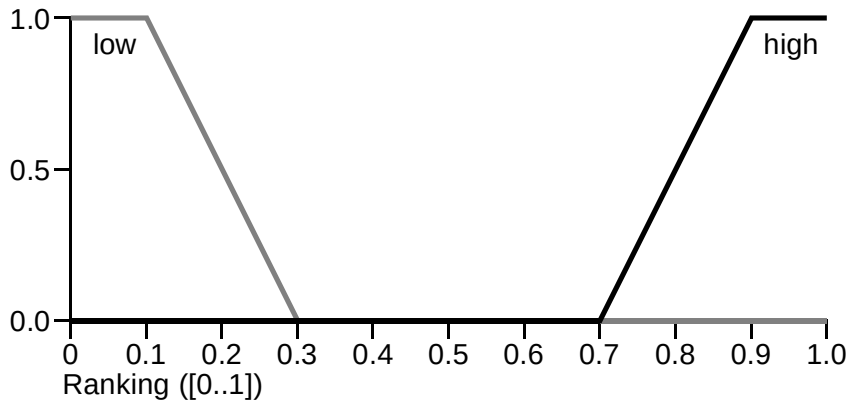

Figure A-13: Example Ruleset: Linguistic Output Variable "Ranking"

- Changing a vertical neighbor connection to a column backbus

The rule descriptions are given in the natural form used in the FOOL editor. For all rules, the minimum operator has been used for aggregation "&" (see section 8.4.3.2), and the maximum operator has been employed for accumulation (see section 8.4.3.5). In generating the crisp suggestion values shown in section A.5, the center of gravity method has been used for defuzzyfication (see section 8.4.4). The rules are based partially on experiments conducted with the Xplorer framework and partially on reason.

The output variable "Ranking" and the rules have been chosen in a way, that a value of 0.5 means an indifferent opinion to a specific action. A value of 1.0 would indicate a strong encouragement for the action, while a value of 0 would denote a strong discouragement.

A.4.2.1 Adding a horizontal bidirectional nearest neighbor connection

1. **IF** Total_Use=high **&** Glob_Use=high **&** H_Use=high
   THEN Ranking=high **WITH** 1.0
2. **IF** Glob_Use=high **THEN** Ranking=high **WITH** 0.25
3. **IF** Glob_Use=not_high **THEN** Ranking=low **WITH** 1.0

A.4.2.2 Adding a vertical bidirectional nearest neighbor connection

1. **IF** Total_Use=high **&** Glob_Use=high **&** V_Use=high
   THEN Ranking=high **WITH** 1.0
2. **IF** Glob_Use=high **THEN** Ranking=high **WITH** 0.25
3. **IF** Glob_Use=not_high **THEN** Ranking=low **WITH** 1.0

A.4.2.3 Removing a horizontal nearest neighbor connection

1. **IF** H_Use=low **THEN** Ranking=high **WITH** 0.75

2. **IF** Total_Use=high **THEN** Ranking=low **WITH** 1.0
3. **IF** BBus_Use=high **THEN** Ranking=low **WITH** 1.0
4. **IF** Glob_Use=high **THEN** Ranking=low **WITH** 1.0

A.4.2.4 Removing a vertical nearest neighbor connection

1. **IF** V_Use=low **THEN** Ranking=high **WITH** 1.0
2. **IF** Total_Use=high **THEN** Ranking=low **WITH** 1.0
3. **IF** BBus_Use=high **THEN** Ranking=low **WITH** 1.0
4. **IF** Glob_Use=high **THEN** Ranking=low **WITH** 1.0

A.4.2.5 Changing a horizontal neighbor connection to a row backbus

1. **IF** Complexity=not_high **&** Total_Use=not_high **&** Fanout=high **&** Data_Dir=horizontal **THEN** Ranking=high **WITH** 1.0
2. **IF** Total_Use=not_high **&** H_Use=low **THEN** Ranking=high **WITH** 1.0
3. **IF** H_Use=low **&** Data_Dir=vertical **&** BBus_Use=high **THEN** Ranking=high **WITH** 1.0
4. **IF** Num_VNN=low **THEN** Ranking=low **WITH** 1.0
5. **IF** Total_Use=high **THEN** Ranking=low **WITH** 0.15

A.4.2.6 Changing a vertical neighbor connection to a column backbus

1. **IF** Complexity=not_high **&** Total_Use=not_high **&** Fanout=high **&** Data_Dir=vertical **THEN** Ranking=high **WITH** 1.0
2. **IF** Total_Use=not_high **&** V_Use=low **THEN** Ranking=high **WITH** 1.0
3. **IF** V_Use=low **&** Data_Dir=horizontal **&** BBus_Use=high **THEN** Ranking=high **WITH** 1.0
4. **IF** Num_HNN=low **THEN** Ranking=low **WITH** 1.0
5. **IF** Total_Use=high **THEN** Ranking=low **WITH** 0.15

A.5 Exploration Example

In this section, an example exploration process is demonstrated using a small application domain comprising two applications. The two applications chosen do not make sense as they perform no useful computation. Instead, they have been chosen for illustrational purposes to show the steps in the exploration process.

Furthermore, the Xplorer is based on extensive user interaction. Thus, a specific train of thoughts has been assumed, so the solution is only one possibility, depending on the designer's goals. In this example, the aim of the exploration is to find an architecture for both applications, which avoids the use of the global

bus at all. Data I/O is done over edge ports, with the input ports located to the west, and the outputs to the eastern edge. Such an architecture is suitable e.g. for embedded systems. Besides the constraint of global bus avoidance, the designer will aim to minimize the communication resources needed, implementing as few nearest neighbor connections and backbuses as possible. The objective of the example exploration can thus be summarized to find suitable combinations of a minimal number of communication resources, such that the datapath can be mapped without needing a global bus connection. For this objective, the example rule set described in sectio nA.4 can be used.

It is further assumed, that in the underlying production process a row or column backbus is easier to implement and causes less cost than a nearest neighbor connection. The backbuses are allowed one writer in the configuration, but are able to broadcast data to several readers.

The exploration is done following the approach described in section 8.1, thus starting with the specification of the applications in the domain to be explored and the selection of one application (preparation phase), followed by the exploration process itself, and a final verification step to make sure the architecture suits also the second application. The three steps will be described in the following.

A.5.1 Preparation Phase

In this phase, the designer specifies the applications and selects the one for the exploration process. The specification is usually done in the ALE-X language. The two example applications are called "Datapath 1" and "Datapath 2". They are chosen for illustrational purposes and do not make sense otherwise. The ALE-X code for the two applications is shown in figur eA-14.

A run of the architecture estimator and a preliminary complexity estimation (see section 8.3.1.3) render the results given in table A-1.

	Datapath 1	**Datapath 2**
Minimal Array Size	4 x 4	4 x 4
Minimal Horizontal / Vertical Connections	1 / 1	1 / 1
Number of Operators	14	12
Number of Input / Output Ports	4 / 4	4 / 2
Datapath Complexity	0.70	0.59

Table A-1: Estimation results for example applications

a)

```
//    Datapath 1
rALUsubnet SubNet_0 (SW0)
ScanWindow SW0
{
  int i0 at [0][0];
  int i1 at [0][1];
  int i2 at [0][2];
  int i3 at [0][3];
  HandleOffset [0][4];
};
{ // Datapath description
  int a,b,c,d;
  a=i1-(i0*i1);
  b=i3-(i2*i3);
  c=i1*(i0+a);
  d=i3*(i2+b);
  i0=i0+c+i0;
  i1=i0+c-i1;
  i2=i2+d+i2;
  i3=i1+d-i3;
}
```

b)

```
//    Datapath 2
rALUsubnet SubNet_0 (SW0)
ScanWindow SW0
{
  int i0 at [0][0];
  int i1 at [0][1];
  int i2 at [0][2];
  int i3 at [0][3];
  HandleOffset [0][4];
};
{ // Datapath description
  int a,b,c,d;
  a=(i0*i2)-(i1*i3);
  b=(i0*i3)+(i1*i2);
  c=(i0*a)-1;
  d=(i0*b)+a;
  i0=c;
  i1=d;
}
```

Figure A-14: Example applications for exploration:
a) Datapath 1
b) Datapath 2

Both applications have the same minimal requirements for array size and connections. However, datapath 1 features two more operators, two more ports, and a higher datapath complexity. Thus, it is reasonable to select datapath 1 for the exploration, as an architecture satisfying its requirements is likely to suit also datapath 2.

A.5.2 Exploration Phase

The exploration of datapath 1 starts with an initial mapping onto the minimal architecture, which is version 0 in the exploration process. The result of this mapping, and the architecture are shown in figure A-15. The analyzer extracts (amongst others) the set of statistics from the mapping, which is shown in the according column in table A-2. For simplicity, only the values relevant for the example rule set are shown. The suggestions for the six design actions are shown in the according column of table A-3. A value greater 50% means, that the according action is encouraged, while a value lower than 50% means, that the action is probably not so good an idea.

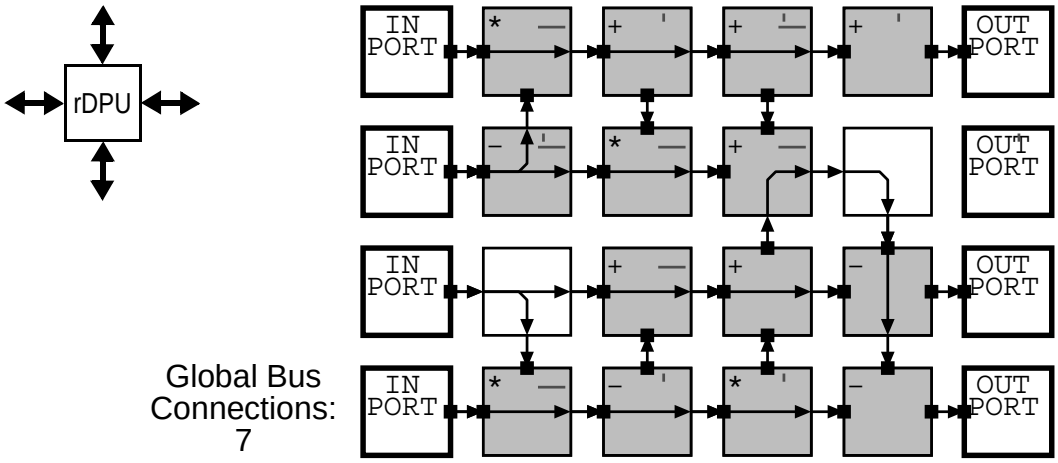

Figure A-15: Example exploration: Initial architecture and mapping

The initial architecture features seven global bus links, which is not bearable. The suggestions show, that it would be a good move to add a horizontal or a vertical neighbor connection. It is assumed, that the designer decides to do both, resulting in the architecture version 1 with the according mapping of the datapath shown in figure A-16.

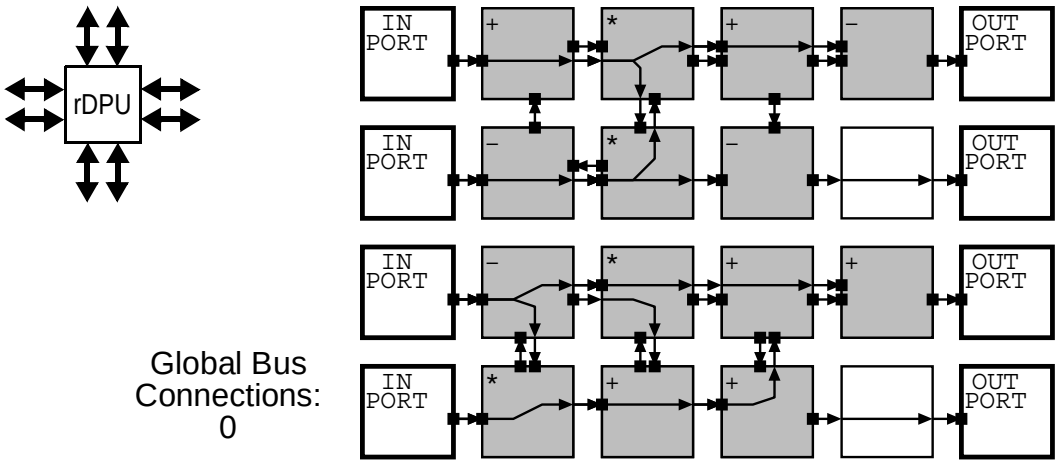

Figure A-16: Example exploration: Version 1 architecture and mapping

The new architecture satisfies the application requirements, as it does not need any global bus links. However, it features two nearest neighbor connections on each side, giving room for optimization. The suggestions in table A-3 hint to three possibilities: Removal of a vertical NN connection, or conversion of either a horizontal or vertical NN connection into a row or column bus respectively. At a first try, the designer decides to remove a vertical connection, resulting in the scenario shown in figure A-17.

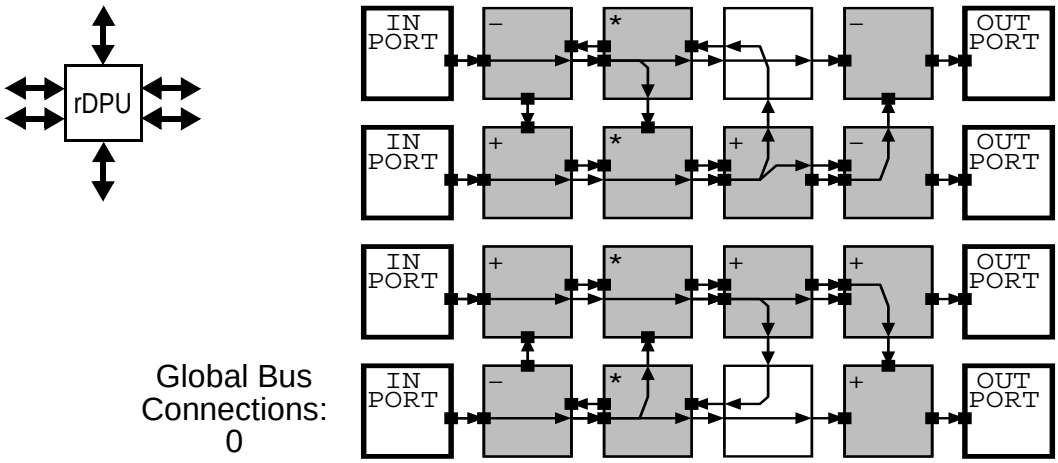

Figure A-17: Example exploration: Version 2 architecture and mapping

This version 2 architecture fulfils the requirements. Though, the suggestions for this architecture show no further idea for improvement. In fact, any conversion or removal of a communication resource at this point would lead to a mapping with several global bus connections (the according mappings are left out for space reasons). It is thus worth to mark this version and continue the exploration from version 1, by converting a horizontal connection into a row backbus. This results in version 3 shown in figure A-18.

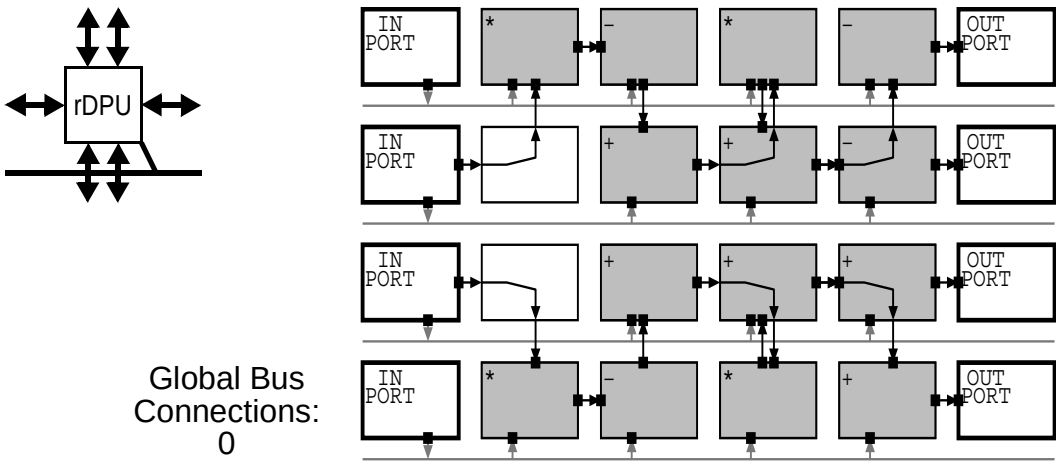

Figure A-18: Example exploration: Version 3 architecture and mapping

While this version still meets the designer's goals, the framework suggests another possible enhancement by converting also a vertical connection into a backbus. The resulting architecture version 4 and the mapping are shown in figure A-19. According to the suggestions shown in table A-3, this architecture

shows also no further possibilities for enhancement. The results of the exploration phase are thus the two architectures denoted as version 2 and version 4.

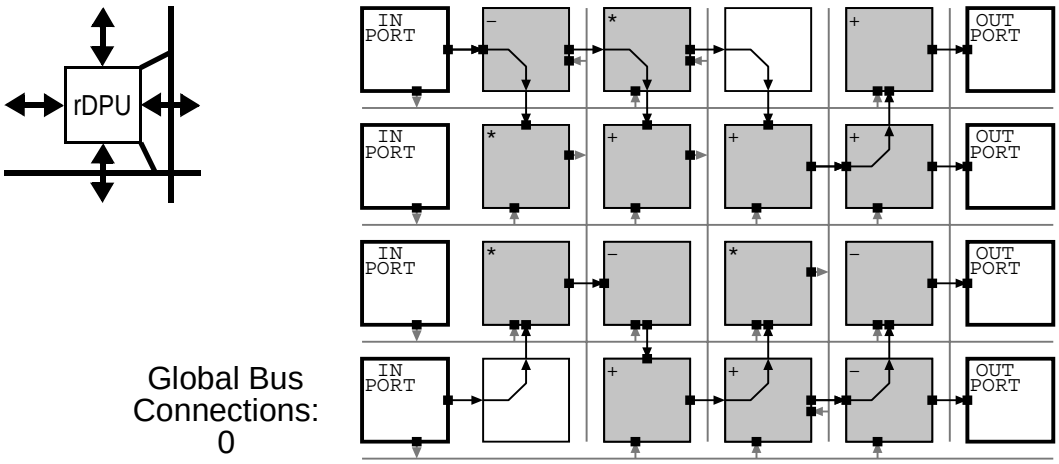

Figure A-19: Example exploration: Version 4 architecture and mapping

	Version 0	**Version 1**	**Version 2**	**Version 3**	**Version 4**
Horizontal NN Connects	1	2	2	1	1
Vertical NN Connects	1	2	1	2	1
Row Backbuses	0	0	0	1	1
Column Backbuses	0	0	0	0	1
Global Bus Links / Share of All Links	7 / 22%	0 / 0%	0 / 0%	0 / 0%	0 / 0%
Horizontal Connect Usage	95%	84%	93%	80%	80%
Vertical Connect Usage	75%	42%	66%	42%	71%
Backbus Usage	0%	0%	0%	100%	88%
Total Connect Usage	85%	63%	80%	75%	75%
Average Fanout	1.78	1.78	1.78	1.78	1.78
Datapath Complexity	0.55	0.51	46%	51%	51%
Main Dataflow Direction	+20%	+42%	+27%	+38%	+9%

Table A-2: Statistics gathered by analyzer in example exploration

	Version 0	Version 1	Version 2	Version 3	Version 4
Add a Horizontal NN Connection	**95%**	5%	5%	5%	5%
Add a Vertical NN Connection	**87%**	5%	5%	5%	5%
Remove a Horizontal NN Connection	5%	13%	5%	5%	6%
Remove a Vertical NN Connection	5%	**65%**	5%	44%	6%
Convert a Horizontal NN Connection Into a Row Backbus	8%	**68%**	13%	37%	35%
Convert a Vertical NN Connection Into a Column Backbus	8%	**65%**	8%	**65%**	8%

Table A-3: Design suggestions for example exploration. Bold values denote promising actions.

A.5.3 Verification Phase

The task of the verification phase consists in mapping the other applications of the domain onto the resulting architecture(s) from the exploration phase. In this example, there were two suitable architectures marked as result. There is only one other application, which is now mapped onto both architectures. The resulting mappings are shown in figure A-20. Both architectures are suitable also for the second application "Datapath 2", as no global bus links are needed. Thus, no further iteration is necessary and the exploration is finished, resulting in two possible architectures for the given application domain.

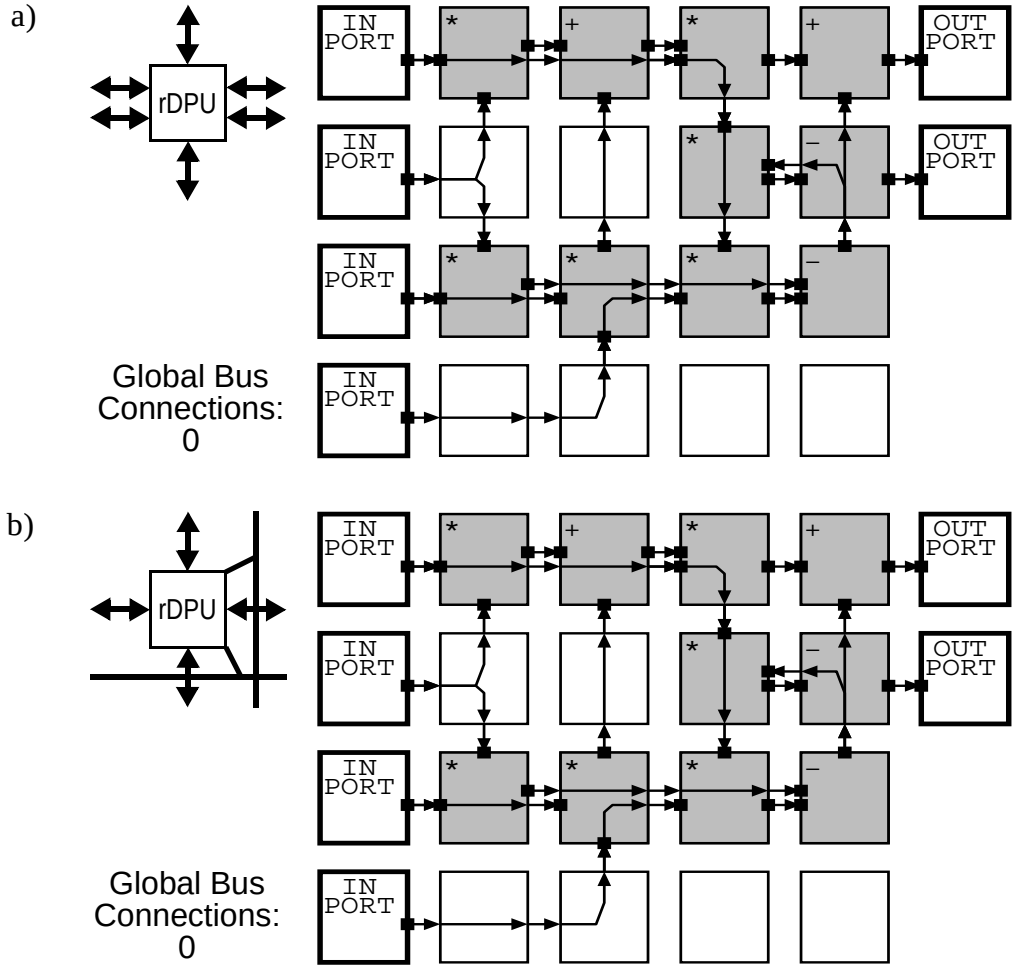

Figure A-20: Example exploration verification phase:
a) Mapping of Datapath 2 onto architecture version 2
b) Mapping of Datapath 2 onto architecture version 4