6. A Novel Data Sequencing Concept

This chapter will introduce a new method for data sequencing. It has been developed to support highly computation intensive applications to be implemented on word-level structurally programmable platforms. To realize the integration of such *soft ALUs* into a computing machine, a deterministic data sequencing mechanism is needed, because the von Neumann paradigm does not efficiently support *soft* hardware [HBH96e]. As soon as a data path is changed by structural programming, a von Neumann architecture would require a new tightly coupled instruction sequencer. A well suited backbone paradigm for implementing such a deterministic reconfigurable hardware architecture is based on data sequencing.

The data sequencing methodology, which will be introduced in this chapter, has evolved from the basic concepts used in the earlier Map-oriented Machines (MoM-1 see section 5.8 at page 83, MoM-2 see section 5.9 at page 83, and MoM-3 see section 5.10 at page 84). The underlying address generation method has been adapted and developed further for the requirements of computing machines based on structurally programmable platforms only. Besides the need for small reconfiguration sets, the major novelty of the data sequencing concept presented here is the waiving of a controller based on instructions. While earlier implementations of the Map-oriented Machines (MoM-2 and MoM-3) utilized a microprocessor to control data sequencing applications, here an autonomous concept will be presented. This chapter will introduce a data sequencing methodology based on the slider method (section 6.2 at page 96) and a stack-based control (section 6.4 at page 114) mechanism. No additional control processor will be needed to perform computations. With the stack-method the first time a highly flexible address generator is introduced, which does not fall back on software based sequencing for complex access patterns.

IP Concept

The way this data sequencing concept is implemented is not restricted to a specific target platform or technology. This is also supported by the underlying Xputer paradigm (see section 6.1.1 at page 90), which requires a loose coupling between reconfigurable datapath and data sequencer. The presented data sequencing concept is planned that data sequencers are designed as IP (intellectual property) block and integrated into reconfigurable computing machines. Therefore the concept is very flexible and may be adjusted to a specific case. This thesis presents two implementations of the data sequencing method:

- a KressArray implementation to be mapped onto the same device as used for the reconfigurable ALU (section 6.5 at page 121), and

- a stand-alone implementation to be integrated as an independent component (appendix D).

Chapter Overview

This chapter first briefly illustrates two level address generation and the slider method having been introduced elsewhere. Later a novel sequencing architecture based on these will be introduced, which relies on a new stack based control mechanism. After that the mapability as a soft macro onto KressArray architectures will be investigated.

6.1 The Basic Data Sequencing Principles

This section will give the foundations for the data sequencing concept to be presented in the subsequent sections. The data sequencing concept is based on the well defined Xputer paradigm, which will be shortly introduced in the following subsection. After that the general two-level address generation method (section 6.1.2 at page 92) of the Xputer paradigm and its basic hardware implementation will be described (section 6.1.3 at page 94).

6.1.1 The Basic Xputer Architecture

The data sequencing concept presented in this thesis has been developed for reconfigurable computing machines based on the Xputer architecture. The Xputer paradigm has been published in e.g. [HHW89], [HHW90], [HHS92], or [AHR94].

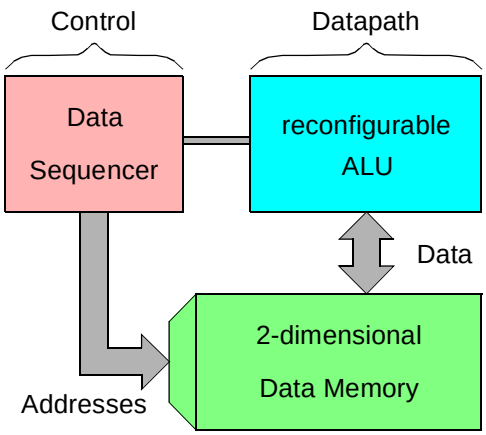


Figure 6-1: The basic Xputer architecture.

The Xputer is deterministically data driven and supports optimization methods, which are based on data dependencies, but it is not a data-flow machine [HHR91]. Its basic architecture consists of a data sequencer, a data memory and a reconfigurable ALU (see figure 6-1 at page 90). Both parts of the Xputer are programmable. While the datapath is typically built of reconfigurable devices, the data sequencer structure may be hardwired and programmed by only a few parameters for generic address generation. The components of the Xputer architecture may be integrated in one or separated devices.

During operations the data sequencer generates an address stream for the data memory to generate data streams to and from the reconfigurable datapath. All data manipulations are performed by the reconfigurable datapath.

Address Generation for Reconfigurable Systems

This thesis presents a data sequencing mechanism to be used in reconfigurable computing. Reconfigurable systems are usually programmed before an application is executed. Therefore the instruction fetch is also called configuration. During runtime no instruction fetches, i.e. configurations, are needed for the current application. As a result also no memory cycles for this task are required. This is an important benefit against microprocessors, because the memory interface is not burdened by instructions. To preserve this benefit of reconfigurable systems also the data sequencing part should be free of instruction cycles. For this reason generic address generation is introduced:

Definition 6-1: Generic Address Generation

Generic Address Generation is the automatic generation of an address sequence based on pre-programmed parameters and end conditions.

This kind of address generation can be expressed by a dedicated formula embedded into nested loops. The hardware implementation may consist of an arithmetic address manipulation datapath controlled by a finite state machine (FSM).

KressArray Use in Xputer-Based Systems

Already the MoM-3 (section 5.10 at page 84) has utilized the KressArray-1 as a reconfigurable ALU. The work presented in this thesis considers to utilize the KressArray-3 in a Xputer-based architecture. Since all computations inside the KressArray are data-driven, the application mapper MA-DPSS [HHH00] determines also the data streams into and out of the array similar to systolic arrays. Then it is the task of the data sequencer to carry out this data streams.

To implement window-based memory access as performed by the MoM-3 requires an application specific data cache. Such a cache will be integrated into the reconfigurable ALU. Thus the MA-DPSS will allocate registers inside the KressArray-3 to form the smart interface needed for a current application [Kre96]. Data needed several times will be stored in the smart interface registers and routed to the relevant operators.

6.1.2 A Two-Level Address Generation Method

Typical applications for reconfigurable accelerators consist of a function embedded in a loop nest [Bec97] [Sch94]. Usually this function has multiple input values and produces several results. Both, input data and results, are arranged in arrays and referenced by their index value, produced by the nested loops. This section presents an execution model for a 2-stage address generation method, which is directly derived from such applications (see also [HBH97a] or [HBH97e]).

Two-level address generation is based on a window, which is moved over the computation data. While the window points to all memory locations for one iteration, multiple iterations are performed by the movement of the window. The application of this method is very illustrative for image processing examples [HBH97b], but also well suited for all other applications. Other examples like Lee Routing [Hir91], design-, and electrical rule check (DRC, ERC, see [HHW89], [HHW90]) have been already published.

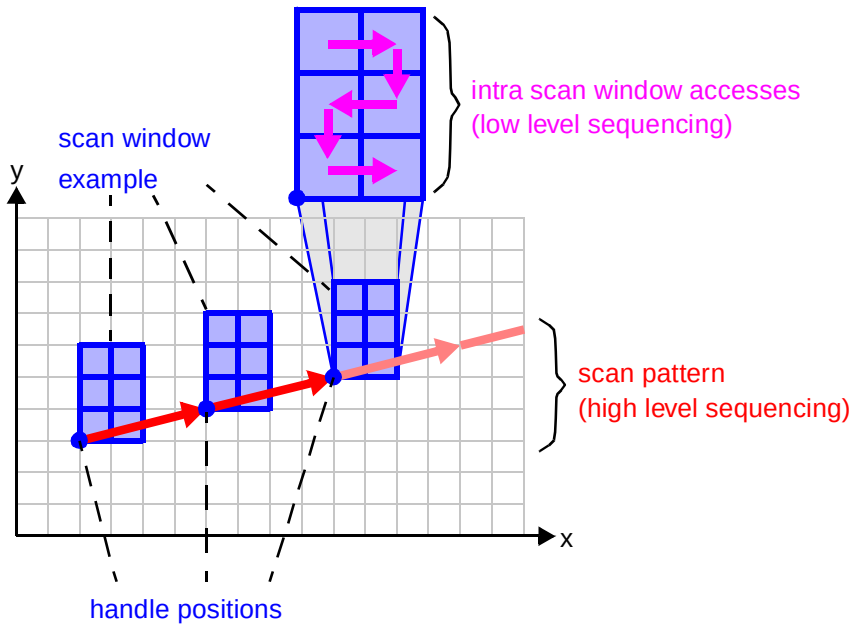


Figure 6-2: The 2-stage address generation.

To clarify how computations are performed an execution model is shown in figure6-2 at page 92. A large amount of input data is typically organized in arrays (e.g. matrix, picture) where the array elements are referenced as operands of a computation in a current iteration of a loop. These arrays can be mapped onto a 2-dimensionally organized memory without any reordering. The part of the data memory, which holds the data for the current iteration, is determined by a so called scan window, which can be of any size and shape. Each position of the scan window is labeled as read, write or read and write. The labels indicate the performed operation to the specified memory location. The intra scan window accesses are performed in a sequence. This address generation stage is called low level sequencing.

Definition 6-2: Scan Window

A scan window *SW* of the size *n* is a set of *n* relative positions *(x, y, m)* with an assigned access mode. Since all positions of the scan window are relative, there is at least one *i* and *j*, where $x_i=0$ and $y_j=0$:

$SW = \{(x_1, y_1, m_1), (x_2, y_2, m_2), \ldots, (x_n, y_n, m_n)\}$, $n \geq 1$, and

$\exists i \in \{1, \ldots, n\} \wedge x_i = 0$, and

$\exists j \in \{1, \ldots, n\} \wedge y_j = 0$.

Definition 6-3: Handle Position

The handle position of a scan window $h_{SW}=(x,y)$, $x, y \in \{0, 1, 2, 3, \ldots\}$ is an absolute address and specifies the lower left corner of the scan window *SW.*

The position of the scan window is determined by the lower left corner, called handle position. Each time a handle position corresponds to a valid set of index values generated by the loop nest of the application and the loop body is evaluated on the specified data set. Operations are performed by moving the scan window over the data map and applying the configured operator of the functional unit on the data in each step. Thus this movement of the scan window called scan pattern is the main control mechanism. The generation of the scan pattern is called high level sequencing.

Definition 6-4: Scan Pattern

A scan pattern *SP* is a finite set of handle positions:

$SP = \{h_1, h_2, \ldots, h_n\}$, $n \geq 1$.

The order of handle positions in *SP* corresponds to the temporal order, in which they are produced.

In fact the execution model realizes a 2-stage data sequencing. On the first stage with the position of the scan window all data for one iteration is indicated. On hardware level the position of the scan window is determined by the x- and y-addresses of the scan pattern. On the second stage the data is sequenced from the scan window into the functional unit, which implements the loop body, and back. This is done for each handle position depending on the read/write labels. On hardware level the data sequencer computes physical memory addresses for each scan window position.

The following sections will focus on high and low level sequencing to find an efficient implementation for reconfigurable computing machines.

6.1.3 The Principle Data Sequencer Architecture

In this subsection the basic hardware structure necessary for generic data sequencing is presented. The generation of data accesses as described before is performed in a two stage pipeline. Generic address generation has the advantage that no instruction and therefore no memory cycles are needed for address generation. Addresses are not computed by a command sequence but generated from parameters previously programmed.

The first pipeline stage generates gradually the handle positions, which form the scan pattern. This unit is called Handle Position Generator (HPG). The HPG consists of two identical 1-dimensional Generic Address Generators (GAG), which synchronously generate the x- and y-components of the handle position. The second pipeline stage processes the handle position and generates all accesses specified in the scan window. Thus this unit is called Scan Window Generator (SWG). Figure 6-3 shows all components.

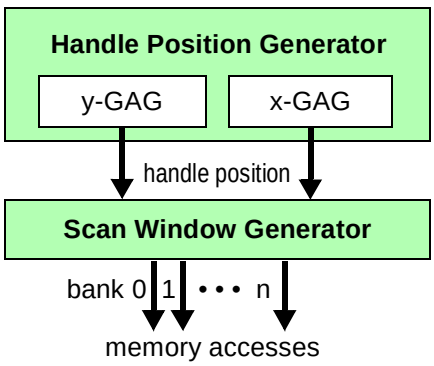


Figure 6-3: Basic data sequencer architecture consisting of a two stage address generator pipeline.

The Scan Window Generation

The second stage of the presented address generation method is the low level sequencing. In this stage the intra scan window accesses are performed.

The vision behind the presented data sequencing concept is, that the scan window holds all data needed to perform one computation step configured into the functional unit. In that context, the scan window indicates all necessary memory locations on the basis of its position. Performed memory accesses may be read, read and write, or write operations. Usually all read accesses are performed before write operations. The reason for this is, that usually an application requires its input data before it produces results.

The scan window may be of any size and shape. Basically the order of the accesses inside the scan window may be arbitrary. Furthermore the modification of the order of the accesses is the source of the access optimizations in chapter 7. To meet these requirements a look-up table-based implementation of the scan window accesses is suggested.

The basis for scan window accesses is the handle position. All accesses inside the scan window are relative to that position. Scan window locations are stored as an offset relative to the handle position in a look-up table. Each time a handle position is generated all offsets stored in the look-up table are gradually added to generate memory accesses in succession.

The look-up table holds besides the offset also the access type (read or write). Scan window locations, which are read and written, have two look-up table entries; one for read and one for write.

After all accesses, stored in the look-up table, have been performed, low level sequencing stalls until a new handle position is provided. Usually the generation of the handle position is faster than low level sequencing because the speed of low level sequencing is limited by the memory interface performance.

Figure 6-4 at page 96 illustrates the look-up table based scan window generator hardware. The handle position input is stored in a register and look-up table entries are added successively. The look-up table entries are sequenced one after another by a counter. After all scan window positions are generated a comparator raises the *new handle* signal to indicate, that a new handle position can be processed. A more sophisticated implementation of the scan window generator is the MoM-PDA scan window generator in appendix D.2.2 at page 269.

A similar method has already been used by the MoM-3 [Rei99], where a small RISC-like processor has been implemented for scan window generation. In that approach the offsets were integrated in the program instructions.

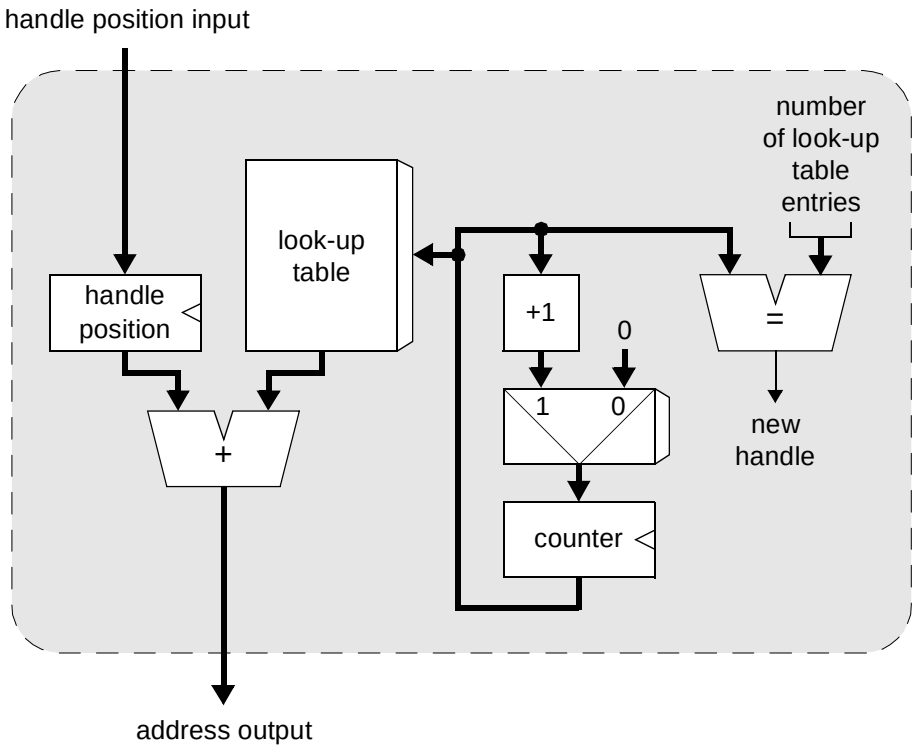


Figure 6-4: The scan window generator implementation.

6.2 Address Generation with the Slider Method

While the low-level data sequencing (figure6-2 at page92) is based on look-up tables integrated into the Scan Window Generator (figure 6-3 at page 94), the generic generation of the handle positions by the Handle Position Generator (figure 6-3 at page 94) will be the topic of this section. For this the slider method has been developed, which is the key element of the following classification of scan pattern (see definition 6-4 at page 93). While the slider model has already been denoted in [HHW90] and [Hir91], there has been scarcely any methodology for its application. This section presents a new classification of scan pattern, where the video scan generated on the basis of the slider model is the atomic element. The goals for this classification have been:

- flexibility for the efficient and fast generation of addresses,
- short configuration times for the reconfigurable system by using only a few parameters, and
- not restricted to special address sequences, i.e. capable to generate arbitrary access sequences.

While a scan pattern can be any arbitrary set of handle positions, many applications require a scan pattern of high regularity. To build a high performance data sequencer hardware, first the different types of scan patterns have to be determined. It is necessary to find suitable classes of scan patterns to have only a small parameter set for their description. This is very important in view to reconfigurable computing because a small parameter set requires less configuration time. In the following a discussion on possibilities to describe scan patterns is given. Other classifications of scan pattern have been presented in [Hir91] and [Rei99]. In contrast to the classification in this thesis, [Hir91] and [Rei99] aim to be general in respect to enclose all possible scan patterns. The classification here encloses all scan patterns as well, but aims to gather the access pattern in a way, that they can be described by only a few parameters.

The Single Step

There may be several ways to design scan patterns. For example the class of all scan steps on a 2-dimensional memory can be defined and then concatenations and variations may be built, further nesting and repetitions are possible. All scan patterns can be obtained by this method, but it isn‘t possible to get a short, efficient description of them.

Definition 6-5: Scan Step

A scan step $\bar{s} = \begin{bmatrix} x_s \\ y_s \end{bmatrix}$ is a vector

between two handle positions h_1 and h_2, such as $h_1 = h_2 + \bar{s}$.

The Linear Scan

To build a fast (re-)configurable hardware to generate scan patterns, a model to describe scan patterns generically is required. Therefore the linear scan, like shown in table 6-1 at page 108, could be defined as the atomic element. Obviously a linear scan may also be of length 1, consequently all single steps can be modeled. With this scheme a larger class of scan patterns is obtained, which can be described with 3 parameters:

Definition 6-6: Linear Scan

A linear scan is a scan pattern $SP = \{h_1, h_2, \ldots, h_n\}, n \geq 1$
can be described by the triple $(h_1, \bar{s}, n)$, where
h_1 is the first handle position of the scan pattern,
$h_{i+1} = h_i + \bar{s}, \forall i \in \{1, \ldots, n-1\}$, $\bar{s}$ is a scan step, and
n is the number of handle positions.

The linear scan as defined above can also be described by a MoPL (see appendi xB) program. Let *h1x* and *h1y* be the x and y components of the handle position h_1 and *xs* and *ys* the x and y components of the scan step $\bar{s}$, then the following MoPL code fragment is equivalent to definition 6-6:

```
/* Other declarations */
scanPattern       LinearScan is n steps [xs,ys];
begin
        ...
        move ScanWindow to Data [h1x,h1y];
        LinearScan[Data];
        ...
end;
```

6.2.1 The Video Scan

Since 2-dimensional memory is available always more than a linear scan is necessary to make use of it. Therefore the video scan (an example is pictured in figure 6-5a) is introduced as the atomic element of the classification. This is a regular scan pattern, which scans the 2-dimensional memory in both directions. The video scan has already been presented in [HHW87] and was also used in the classifications of [Hir91] and [Rei99].

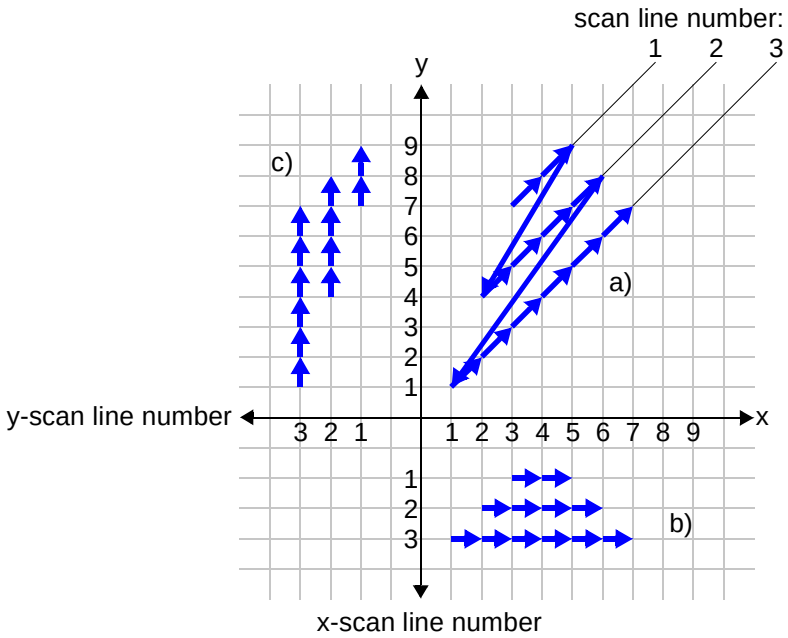


Figure 6-5: A video scan example:
(a) scan steps
(b) (c) with x- and y-components.

The video scan may be seen as a set of linear scans connected by single scan steps, where all linear scans use the same scan step. This scan step is also called inner scan line (see definition 6-7 at page 101) scan step. In figure 6-6 at page 100 the slider model is introduced, which defines video scans more efficient. It introduces parameters required for the generic generation of video scans.

Using the video scan as the atomic element for data sequencing is no limitation because video scans can be a linear scan or a single step. The number of connected scan lines may be only one and a linear scan is generated. By setting also the number of handle positions of this scan line to one, a single handle position is generated.

The Slider Model

The slider model proposed in [HHW90] describes generic address generation of a video scan for one dimension in the address- / time-space. The name slider model stems from the fact, that initial and end value for address generation are not fixed but modified during address generation. This results in a dynamic address sequence length.

To obtain a 2-dimensional video scan the slider model has to be used for both dimensions (see figure 6-5 at page 98). First the slider model for one dimension is introduced and later the combination for two dimensions is explained.

The address generation of the slider model is illustrated in fi gure6-6 at page100. The slider model generates gradually linear scans in a 1-dimensional address space. Therefore five sliders are introduced: Base (B), Limit (L), Floor (F), Ceiling (C) and Address (A). Each time a linear scan is generated (i.e. the Address stepper performs one scan line) the Address is initialized at the Base position and addresses are generated in ΔA steps till the Limit is reached. ΔA may be positive or negative, i.e. $B_0 \leq L_0$ or $B_0 > L_0$. Before the first Address stepper loop, the Base slider is placed at B_0 and the Limit slider is placed at L_0. Each time one scan line is completed, Base is moved by ΔB and Limit is moved by ΔL. ΔB and ΔL may be positive or negative:

- ΔB positive: $B_0 \leq F$
- ΔB negative: $B_0 > F$
- ΔL positive: $L_0 \leq C$
- ΔL negative: $L_0 > C$

If Base meets or passes Floor or if Limit meets or passes Ceiling the address generation is finished. This method is illustrated in figure 6-6 at page 100. Figure 6-7 at page 101 lists exemplary a pseudo code notation of the slider model for $Floor \leq B_0 \leq L_0 \leq Ceiling$.

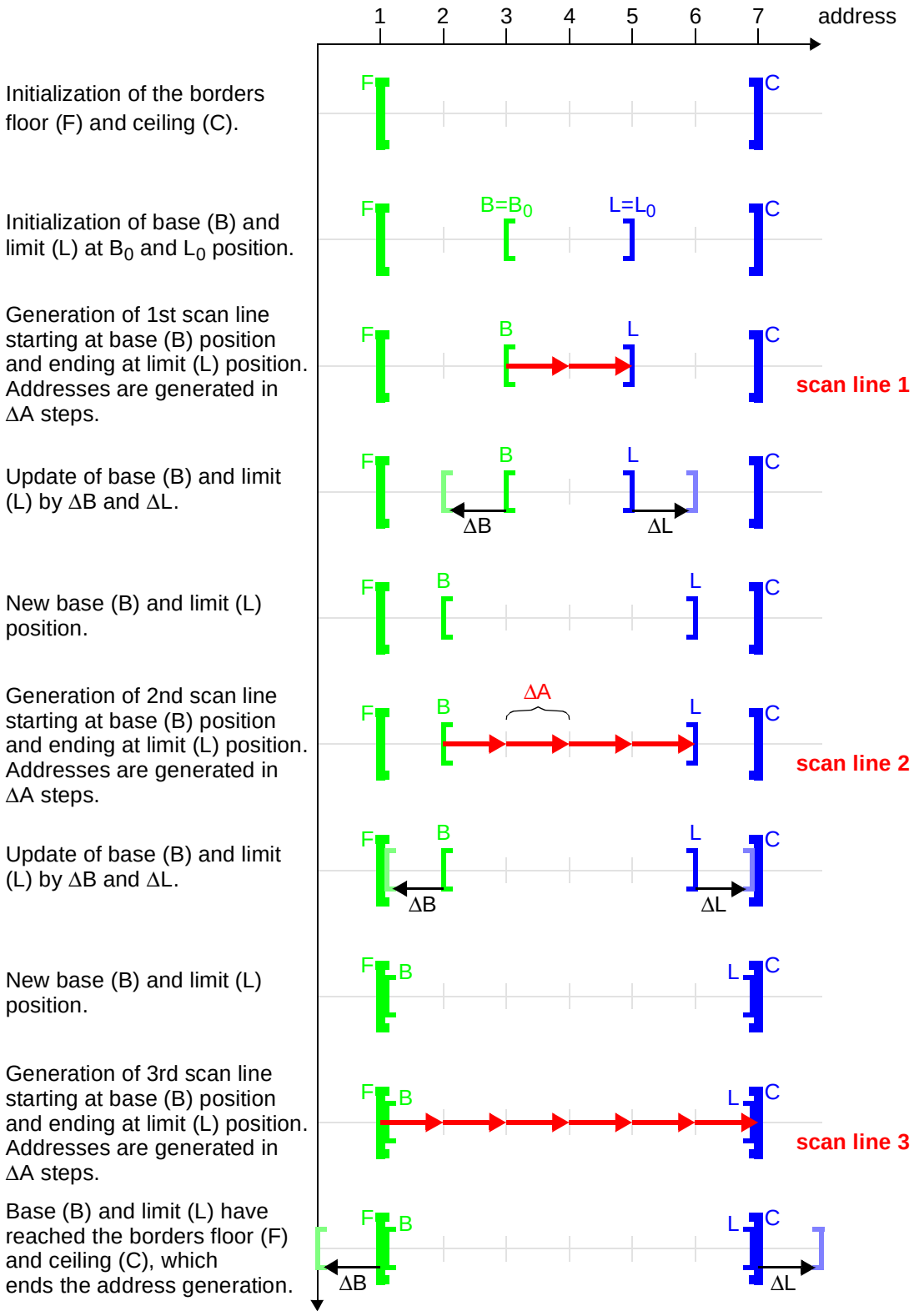


Figure 6-6: Illustration of the slider model [HBH97a] and generated scan sequence for x-component of the video scan in figure 6-5 at page 98.

```
for (  Base=B0, Limit=L0;
       Base>=Floor AND Limit<=Ceiling;
       Base=Base+ΔB, Limit=Limit+ΔL)
       {      for (  Address=Base;
                     Address<=Limit;
                     Address=Address+ΔA)
                     { output (Address); }
       }
```

Figure 6-7: Pseudo code description of the slider model for $Floor \leq B_0 \leq L_0 \leq Ceiling$.

Definition 6-7: Scan Line

Let $\overline{B}$ and $\overline{L}$ be valid Base and Limit values, which occur at the same state of the slider models of SP_{VS} during address generation for the video scan SP_{VS}.

A scan line $SP_{SL}(\overline{B}, \overline{L})$ of a video scan SP_{VS} is a linear scan which holds all handle positions, which are generated for $\overline{B}$ and $\overline{L}$ in SP_{VS}. $\overline{B}$ is the first handle position of $SP_{SL}(\overline{B}, \overline{L})$.

The Combination of Two Slider Models to Generate Video Scans

The generation of 2-dimensional scan patterns requires x- and y-addresses. For each dimension a separate Address slider based on the slider model performs the address calculations. Addresses are generated synchronously by both Address sliders, i.e. at each time step every Address slider performs one operation step.

The generated video scan ends when the slider model for one dimension indicates the end of address generation, i.e. Base or Limit exceeds Floor or Ceiling, or Address exceeds Base or Limit.

Definition 6-8: Video Scan

A video scan $SP_{VS} = \{h_1, h_2, \ldots, h_n\}$, $n \geq 1$ is a scan pattern, which can be described by the tuple $(end_{VS}, \overline{B_0}, \overline{L_0}, \overline{F}, \overline{L}, \overline{\Delta B}, \overline{\Delta L}, \overline{\Delta A})$, where end_{VS} is the end condition,

$$\overline{B_0} = \begin{bmatrix} B_{0x} \\ B_{0y} \end{bmatrix}, (B_{0x}, B_{0y} \in \{0, 1, 2, \ldots\}) \text{ is a point,}$$

$$\overline{L_0} = \begin{bmatrix} L_{0x} \\ L_{0y} \end{bmatrix}, (L_{0x}, L_{0y} \in \{0, 1, 2, \ldots\}) \text{ is a point,}$$

$\overline{F} = \begin{bmatrix} F_x \\ F_y \end{bmatrix}, (F_x, F_y \in \{0, 1, 2, \ldots\})$ is a point,

$\overline{L} = \begin{bmatrix} L_x \\ L_y \end{bmatrix}, (L_x, L_y \in \{0, 1, 2, \ldots\})$ is a point,

$\overline{\Delta B} = \begin{bmatrix} \Delta B_x \\ \Delta B_y \end{bmatrix}, (\Delta B_x, \Delta B_y \in \{\ldots, -2, -1, 0, 1, 2, \ldots\})$ is a vector,

$\overline{\Delta L} = \begin{bmatrix} \Delta L_x \\ \Delta L_y \end{bmatrix}, (\Delta L_x, \Delta L_y \in \{\ldots, -2, -1, 0, 1, 2, \ldots\})$ is a vector,

$\overline{\Delta A} = \begin{bmatrix} \Delta A_x \\ \Delta A_y \end{bmatrix}, (\Delta A_x, \Delta A_y \in \{\ldots, -2, -1, 0, 1, 2, \ldots\})$ is a vector, and

$(end_{VS}, \overline{B_0}, \overline{L_0}, \overline{F}, \overline{L}, \overline{\Delta B}, \overline{\Delta L}, \overline{\Delta A})$ are the parameters of an x- and a y-slider model, which generate exactly all handle positions of SP_{VS}.

With the slider model a complete video scan can be described by only 15 parameters: the end condition and for both dimensions Floor (F), initial Base (B_0), step width for Base (ΔB), Ceiling (C), initial Limit (L_0), step width for Limit (ΔL) and step width for the Address steps (ΔA).

The video scan may also be described by a MoPL (see appendix B) program. Basically the MoPL program uses the same parameters as definition 6-8 at page 101, but most parameters are not referenced directly. The parameters are used to calculate *NumberScanLines* and *NumberScanSteps*. The following MoPL code fragment describes a rectangular video scan:

```
/* Other declarations */
scanPattern     ScanLine   is NumberScanSteps   steps [dAx,dAy];
                OuterScan  is NumberScanLines   steps [dBx,dBy];
begin
    ...
    move ScanWindow to Data [B0x,B0y];
    OuterScan(ScanLine[Data];)[Data];
    ...
end;
```

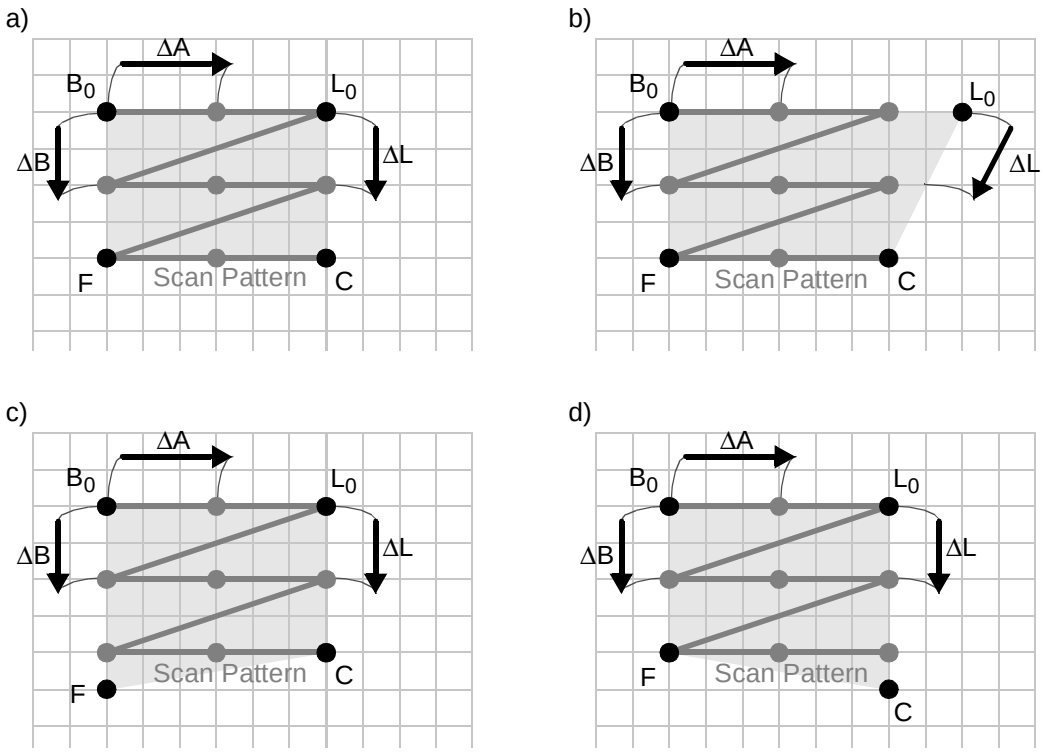


Figure 6-8: Video scan example with possible slider model parameters:
(a) all parameters are handle positions,
(b) Limit is no handle position,
(c) Floor is no handle position, and
(d) Ceiling is no handle position.

The parameters of the slider model can be visualized as shown in figure 6-8. Usually the parameters for the slider model are chosen that $\overline{B_0}$, $\overline{L_0}$, $\overline{F}$, $\overline{C}$ are handle positions (see figure 6-8a). In figure 6-8b Limit and Ceiling are no handle positions, the address stepper passes in the last step the Limit value and produces no handle position. Also Base and Limit do not necessarily meet Floor and Ceiling, they also may pass Floor or Ceiling respectively (see figure 6-8c,d).

By using the video scan as atomic element and building combinations, more scan patterns are classified. This results in four additional classes. The presented classes may be redundant but efficiently reduce the number of parameters needed to generate the necessary access pattern.

6.2.2 The Compound Scan

The compound scan $SP_{compound}(SP_1, SP_2)$ is the concatenation of two complex scans SP_1 and SP_2 (see section 6.2.5 at page 107). First all handle positions of SP_1 are generated and after that all handle positions of SP_2. If the last handle position of SP_2 is produced, the compound scan $SP_{compound}$ is finished. Since the video scan is the atomic element of this classification, the compound scan is a set of at least two video scans.

Definition 6-9: Compound Scan

Let $SP_1 = \{h_1, h_2, \ldots, h_n\}, n \geq 1$, $SP_2 = \{k_1, k_2, \ldots, k_m\}, m \geq 1$ be complex scans.

If $h_n \neq k_1$, then

$$SP_{compound}(SP_1, SP_2) = SP_1 \cup SP_2 = \{h_1, \ldots, h_n, k_1, \ldots, k_m\}, n \geq 1, m \geq 1$$

is the compound scan of SP_1 and SP_2.

If $h_n = k_1$, then

$$SP_{compound}(SP_1, SP_2) = (SP_1 \cup SP_2) \backslash k_1 = \{h_1, \ldots, h_n, k_2, \ldots, k_m\}, n \geq 1, m \geq 1$$

is the compound scan of SP_1 and SP_2.

If the last position of SP1 is the same as the first position of SP2, this handle position is only generated once. A result of definition 6-9 is, that if $h_n = k_1$ and *m=1*, then $SP_{compound}(SP_1, SP_2) = SP_1$.

Table 6-1(II) at page 108 gives some examples for compound scans. Obviously the class of compound scans contains all 2-dimensional scan patterns, because each single step is also a video scan. By forming concatenations all scan patterns can be built. But, because of configuration times having the number of parameters in mind, only *short* concatenations are of interest.

Using MoPL (see appendix B) for scan pattern specification the compound scan is described as follows:

```
/* Other declarations */
scanPattern     FirstScan   is <declaration> ;
                NextScan    is <declaration> ;
begin
    ...
    FirstScan[Data];
    NextScan[Data];
    ...
end;
```

6.2.3 The Nested Scan

Another important class of scan patterns are the scans where an outer video scan performs only one step or a scan line and then an inner complex scan (see section 6.2.5 at page 107) is performed completely each time (see table 6-1(III) at page 108). All parameters of the inner video scan are treated as relative to the actual position of the outer video scan.

Definition 6-10: Nested Scan at the End of a Scan Line

Let

$SP_1 = \{h_{S1P1}, \ldots, h_{S1Pn_1}, \ldots, h_{SmP1}, \ldots, h_{SmPn_m}\}, m, n_1, \ldots, n_{1m} \geq 1$ be a video scan with o scan lines, and

$SP_2 = \{k_1, k_2, \ldots, k_o\}, o \geq 1$ a complex scan.

If $k_1 \neq (0, 0)$, then

$$SP_{nested}(SP_1, SP_2) = \{h_{S1P1}, \ldots, h_{S1Pn_1}, h_{S1Pn_1} + k_1, \ldots, h_{S1Pn_1} + k_o, h_{S2P1}, \ldots h_{SmP1}, \ldots, h_{SmPn_m}, h_{SmPn_m} + k_1, \ldots, h_{SmPn_m} + k_o\}$$

is the nested scan at the end of a scan line of SP_1 and SP_2.

If $k_1 = (0, 0)$, then

$$SP_{nested}(SP_1, SP_2) = \{h_{S1P1}, \ldots, h_{S1Pn_1}, h_{S1Pn_1} + k_2, \ldots, h_{S1Pn_1} + k_o, h_{S2P1}, \ldots h_{SmP1}, \ldots, h_{SmPn_m}, h_{SmPn_m} + k_2, \ldots, h_{SmPn_m} + k_o\}$$

is the nested scan at the end of a scan line of SP_1 and SP_2.

Definition 6-11: Nested Scan at Each Step

Let $SP_1 = \{h_1, h_2, \ldots, h_n\}, n \geq 1$ be a video scan, and

$SP_2 = \{k_1, k_2, \ldots, k_m\}, m \geq 1$ a complex scan.

If $k_1 \neq (0, 0)$, then

$$SP_{nested}(SP_1, SP_2) = \{h_1, h_1 + k_1, \ldots, h_1 + k_m, h_2, \ldots, h_n, h_n + k_1, \ldots, h_n + k_m\}$$

is the nested scan at each step of SP_1 and SP_2.

If $k_1 = (0, 0)$, then

$$SP_{nested}(SP_1, SP_2) = \{h_1, h_1 + k_2, \ldots, h_1 + k_m, h_2, \ldots, h_n, h_n + k_2, \ldots, h_n + k_m\}$$

is the nested scan at each step of SP_1 and SP_2.

Definition 6-12: Nested Scan

A nested scan is either a nested scan at the end of a scan line or a nested scan at each step.

Using MoPL (see appendix B) for scan pattern specification the nested scan is described as follows:

```
/* Other declarations */
scanPattern     InnerScan  is <declaration> ;
                OuterScan  is <declaration> ;
begin
    ...
    OuterScan ( InnerScan[Data];)[Data];
    ...
end;
```

6.2.4 The Meshed Scan

The meshed scan is defined in definition 6-13. Table6-1(IV) at page 108 shows some examples for meshed scans.

Definition 6-13: Meshed Scan

A meshed scan is produced by two ore more video scans. The video scans are called in a fixed order. Each video scan generates a single handle position or a complete scan line. After that the control is passed to the next video scan. This procedure is iterated until the first video scan has finished.

Using MoPL (see appendix B) for scan pattern specification the following code fragment gives a meshed scan example:

```
/* Other declarations */
scanPattern     Scan1       is <declaration> ;
                Scan2       is <declaration> ;
                Scan3       is <declaration> ;
                Scan4       is <declaration> ;
                MeshedScan      is
                begin
                    while ( <end-condition> )
                    begin
                        Scan1;
                        Scan2;
                        Scan3;
                        Scan4;
                    end;
                end;
...
```

6.2.5 The Complex Scan

To complete the classification a last class of scan patterns called complex scans is introduced. Complex scans are all arbitrary combinations of above classes. An complex scan example will be presented in section 6.4 at page 114 (see figure 6-14 at page 117).

Definition 6-14: Complex Scan

All video scans are complex scans. Further all compound scans, nested scans and meshed scans built from complex scans are complex scans.

6.2.6 Summary of the Classification

Based on the discussion above the video scan has been defined as the basic scan pattern. A video scan can also be a single step or a linear scan. Further 3 important classes of combinations video scans have been figured out. These are the compound scan, the nested scan and the meshed scan. It can be proofed that the class of compound scans contains already all arbitrary scan patterns but the nested and meshed scans give the opportunity to describe important scans much more efficient. The fifth class contains all combinations of compound, nested and meshed scans.

As explained above also [Hir91] and [Rei99] give sufficient methods to describe all possible scan patterns. But both methods do not support to describe scan patterns with only a few parameters, which is an important topic in reconfigurable computing. This has been achieved here by using the video scan as the basic element for the classification, which can be efficiently described on the basis of the slider model.

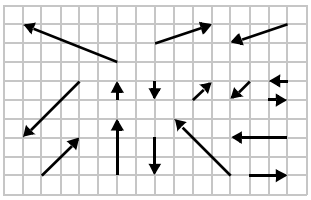
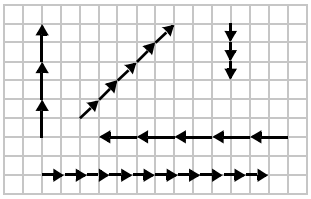
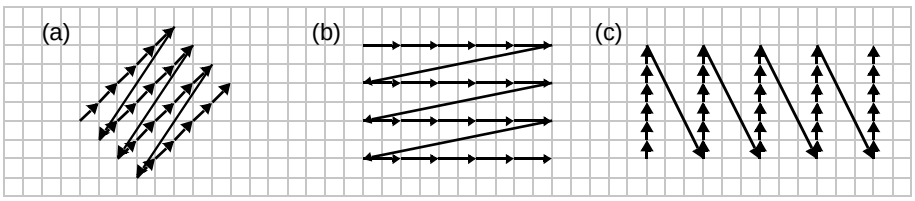
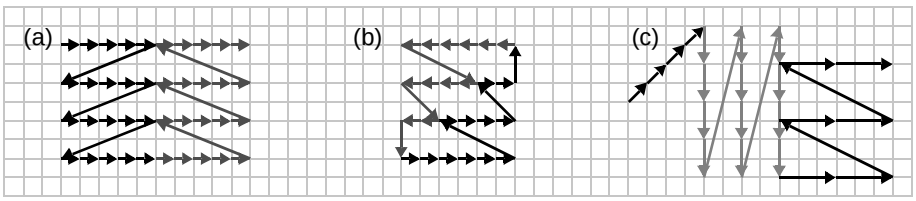
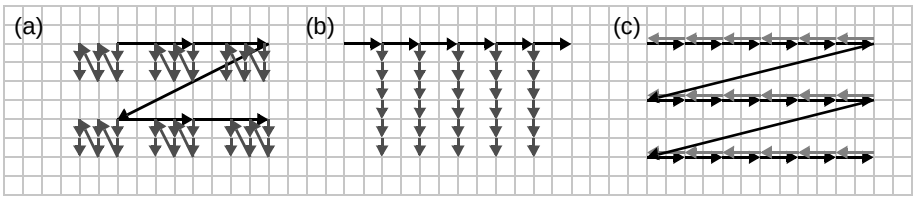
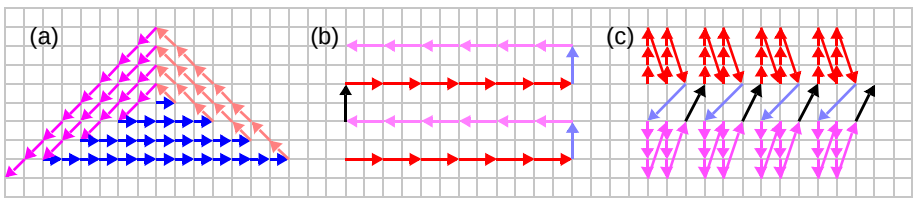
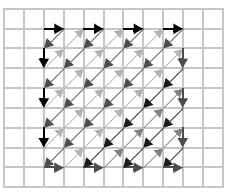
Classes	Examples
single steps	
linear scans	
(I) video scans	
(II) compound scans	
(III) nested scans	
(IV) meshed scans	
(V) complex scans	

Table 6-1: Scan pattern classes at a glance.

6.3 The Hardware Implementation of the Slider Method

This section will explain the hardware implementation of the slider model. First the stepper is introduced, which implements a single slider of the slider model. To set up a complete slider model implementation three steppers (=one-dimensional Generic Address Generator, 1d-GAG) will be utilized. While the raw idea of this approach was mentioned first in [HHW90], no hardware implementation had been worked out. The 1d-GAG will be explained in section 6.3.2 at page 111. After that a complete two-dimensional video scan generator based on six steppers will be presented in section 6.3.3 at page 113.

6.3.1 The Stepper

Figure 6-9 at page 110 shows a stepper, which is capable to implement one slider of the slider model. It consists of an escape unit and a step generator. The step generator calculates one address (step) based on its current state. In combination with the escape unit an address sequence may be generated, whereas the escape unit verifies the validity of each address.

The address generation of the stepper is started by an initialization. For this the *Initial Position* is loaded into the slider position register. The *Initial Position* is usually also the first generated *Address Output*. In the context of video scan generation the *Initial Position* may be B, B_0, L_0.

During address generation, the *Step Width* is added in each step to the actual slider position. During video scan generation the *Step Width* may be ΔB, ΔL, *or* ΔA. After computation of a new *Slider Position* by the step generator, the *End Value* is compared to it by the escape unit. If the *Slider Position* exceeds the limit, which has been set to the *End Value*, the escape unit raises the *escape* bit. Otherwise the stepper continues address generation. The *End Value* may be C, F or L. Table 6-2 shows the parameters to be programmed for Base, Limit, or Address slider implementation.

	initial position	*step width*	*end value*	slider name
Base stepper	B_0	ΔB	*Floor*	Base
Limit stepper	L_0	ΔL	*Ceiling*	Limit
Address stepper	*Base*	ΔA	*Limit*	Address

Table 6-2: Parameter assignment for different stepper implementations.

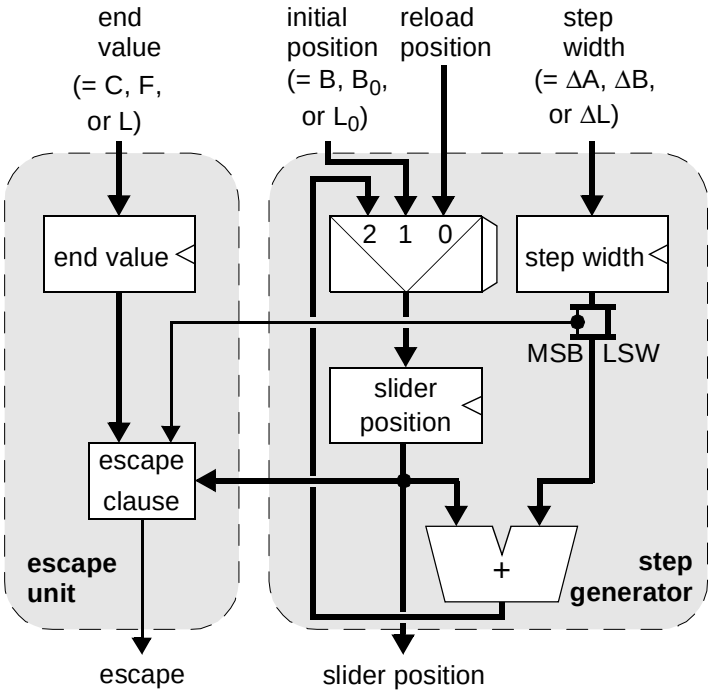


Figure 6-9: Stepper used in the Handle Position Generator to generate one slider (see figur e6-6 at page100) of the slider model for one coordinate (x or y).

The escape unit evaluates, if the *Slider Position* is in the range between *Initial Position* and the *End Value*. Depending on the parameters, there are two cases for the escape unit:

- *Initial Position* ≤ *End Value*, *Step Width* positive: In that case the *Slider Position* becomes invalid, if *Slider Position*>*End Value* (see figure 6-10a at page 111).
- *Initial Position* > *End Value*, *Step Width* negative: In that case the *Slider Position* becomes invalid, if *Slider Position*<*End Value* (see figure 6-10b at page 111).

Both cases are distinguished by the sign of *Step Width*. Therefore the stepper provides the MSB of *Step Width* to the escape unit. Based on this the appropriate result of two comparators is selected (see figur e6-10c at pag e111).

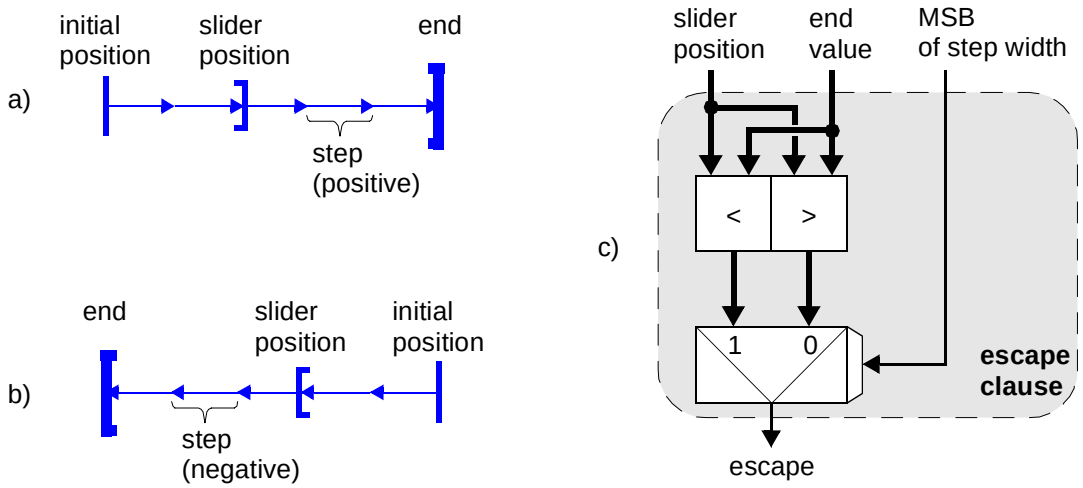


Figure 6-10: The escape unit. Different escape conditions:
(a) *End>Initial position*,
(b) *End<Initial position*, and
(c) hardware implementation of the escape unit.

The MoPL Description of the Stepper

The stepper design is a straightforward hardware implementation of MoPL (see appendix B) features. The following MoPL code fragment describes a 2-dimensional version of the stepper.

```
/* Other declarations */
ScanPattern     ScanStep     is 1 step [StepWidth];
                Stepper      is ScanStep until @[EndValue];
...
begin
        ...
        moveto Data [InitialPosition];
        Stepper;
end;
```

6.3.2 The One-dimensional Generic Address Generator

The 1-dimensional Generic Address Generator (1d-GAG) capable to implement the slider model is composed of three steppers. Each cell is programmed with the appropriate parameters to implement one of the sliders of the slider model (see table 6-2 at page 109). After initialization the 1d-GAG generates addresses autonomously. No programming is required during operation.

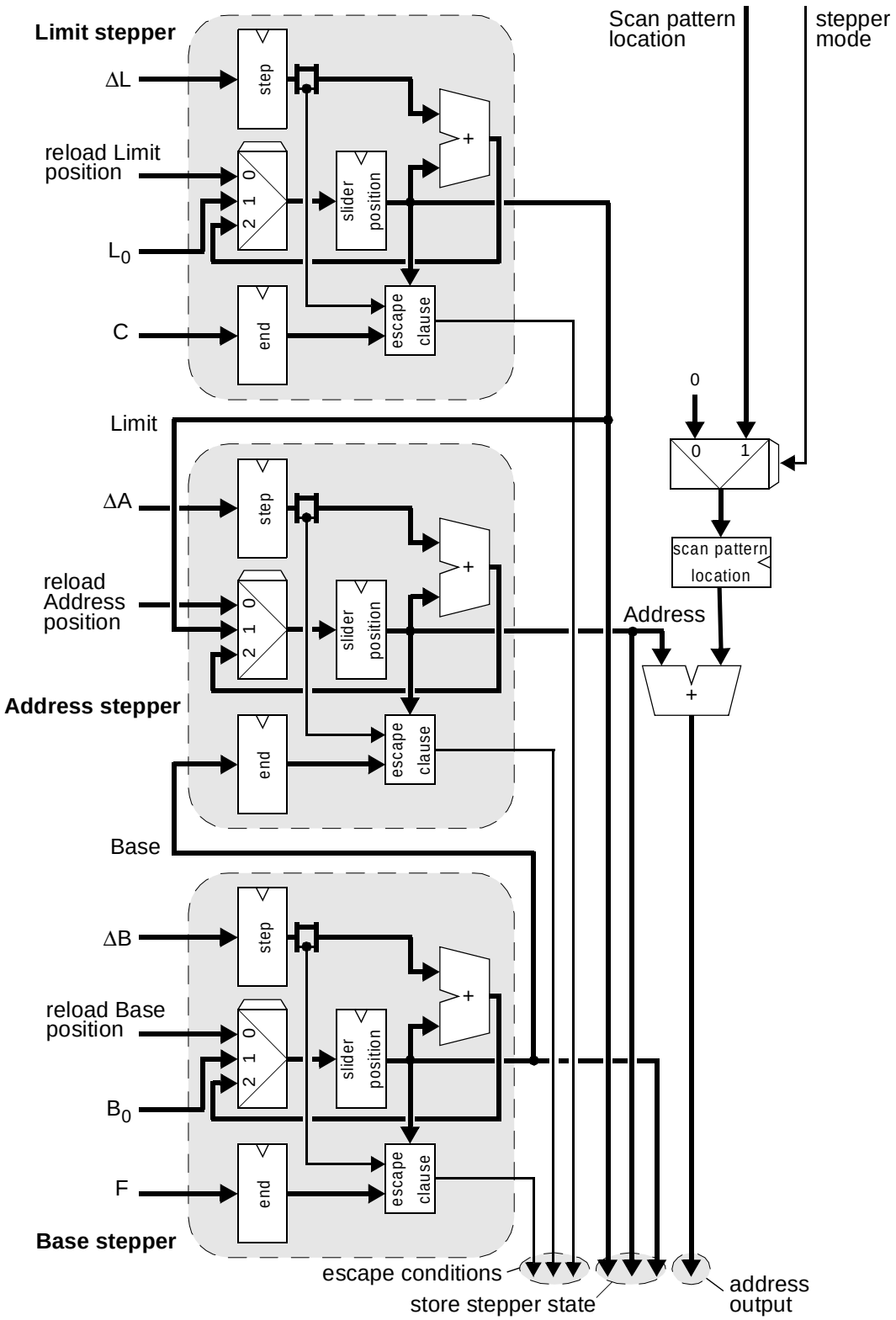


Figure 6-11: 1-dimensional generic address generator (1d-GAG).

Figure 6-11 at page 112 illustrates the interconnected steppers. The output (*Limit*) of the Limit stepper is assigned as the *End Value* to the Address stepper. As the *Initial Position* of the Address stepper is the output (*Base*) of the Base stepper is taken.

For complex scan generation, e.g. nested scans, some video scans need to be relative to a previously generated address. Therefore a *Scan Pattern Location* is added to the Address slider position. A *Stepper Mode* bit, which is set by an external control, selects between absolute or relative address generation.

Besides the *Address Output*, the 1-dimensional GAG provides each slider position as an output. This allows to save the actual stepper state and restore it via the *Reload Position* inputs. Additional output signals are the escape bits of the three steppers.

6.3.3 The Two-dimensional Video Scan Generator

The 2-dimensional video scan generator (see figure 6-12 at page 114) consists of the following main components:

- the x-GAG to generate the x-part of the handle position,
- the y-GAG to generate the y-part of the handle position,
- a step counter, which counts the number of generated handle positions, and
- a stepper control unit.

The step counter operates in parallel to the x- and y-GAGs. Each time a handle position is generated, the step counter decrements its value by one. Before address computation the register *Count* is loaded with the *Initial Counter* value. While the step counter is not needed for video scan generation, it may be used to terminate the address generation before a video scan is finished. This feature is utilized to implement specialized applications.

The 2-dimensional video scan generator is controlled by the stepper control unit. It evaluates the escape lines of the steppers and the step counter. Based on its programming and the *End Detect* signal, the stepper control drives the control lines of the other units.

The end of address generation of the stepper is reached if either the step counter is zero or if one of the steppers exceeds its *End Value*. Which condition terminates the address computations is programmed into the stepper control. During video scan generation the *End Value* may be Floor (*F*), Ceiling (*C*), or Limit (*L*) for one dimension.

The stepper provides the capability to save the actual stepper state as well as the step counter may save the actual value of *Count*. This allows to stop and save the current scan pattern and to restart it later by reloading its parameters. In that case the 2-dimensional video scan generator is not initialized, but all registers are programmed by their *Reload* value.

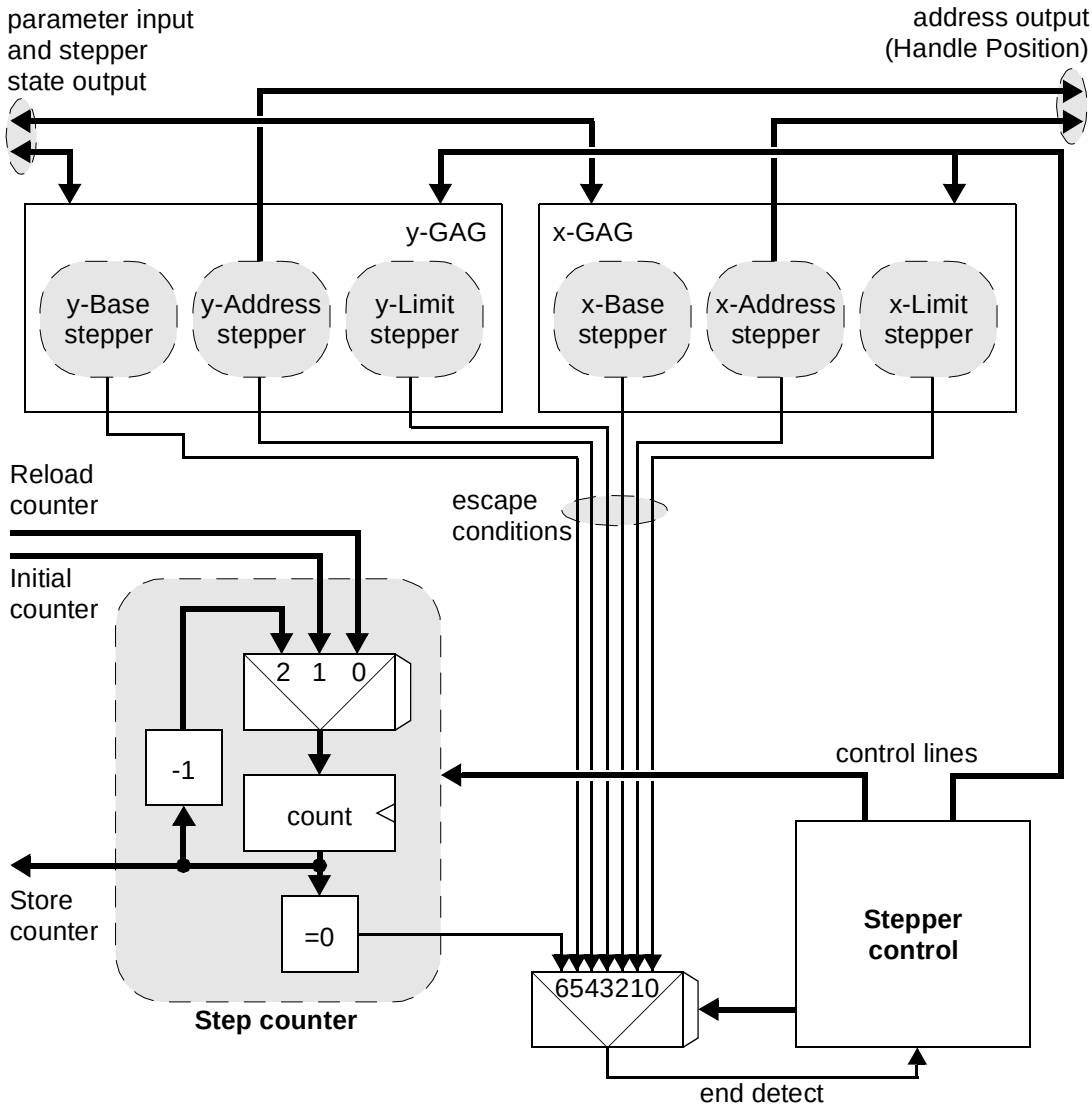


Figure 6-12: The 2-dimensional video scan generator with three steppers per dimension, which implement the sliders of the slider model (figure 6-6 at page 100).

6.4 A Stack Mechanism for the Generation of Complex Address Sequences

This section introduces a stack-based scan pattern generation method. The stack mechanism utilizes the stepper presented in section 6.3.1 at page 109 for address generation. Complex address sequences are generated by reprogramming the parameters of the stepper with parameters sets stored in a parameter stack. This method

saves address generation hardware since only one instance of the stepper will be needed for address generation. Furthermore the classes of compound (section6.2.2 at page 104), nested (section 6.2.3 at page 105), and meshed scans (section 6.2.4 at page 106) can be generated without an additional control instance.

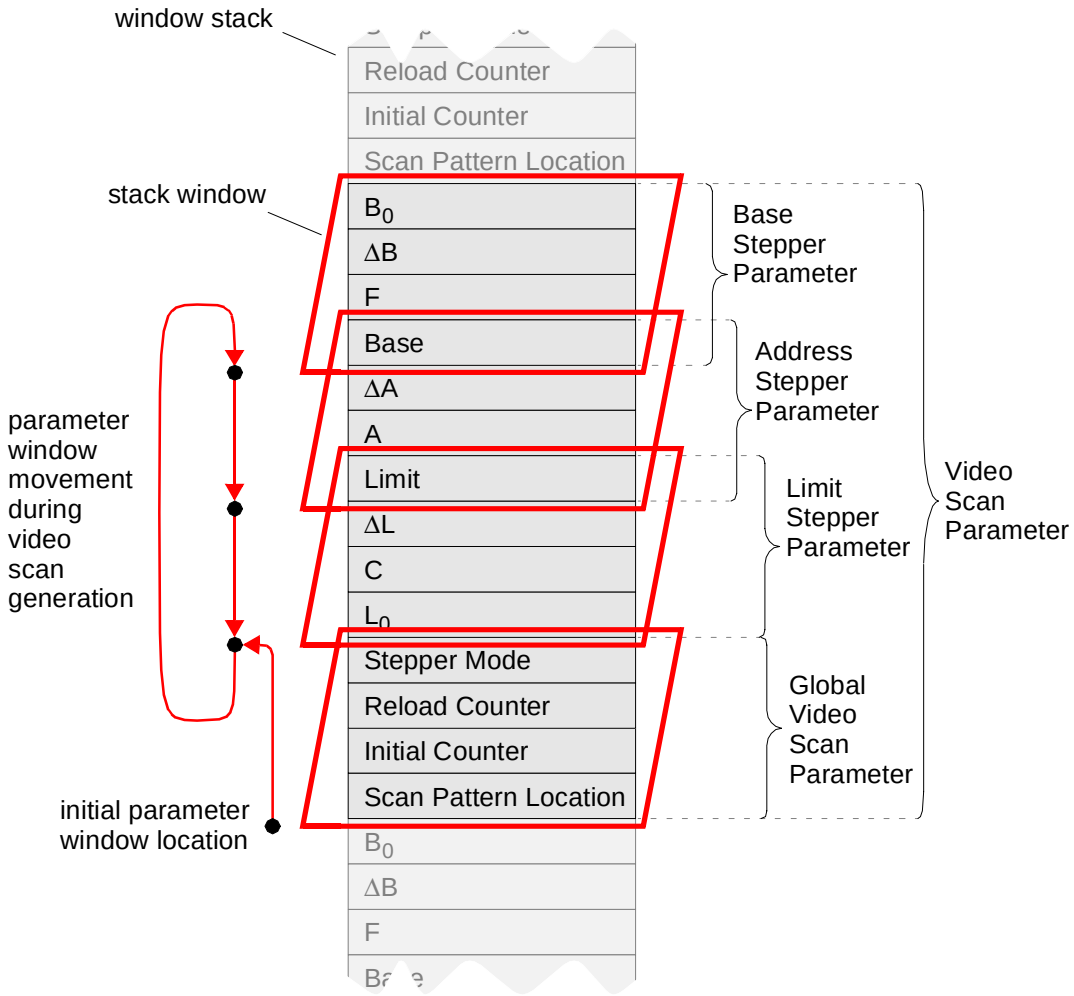


Figure 6-13: Window stack with video scan parameters and stack window movement illustrated.

The basis for the stack-based scan pattern generation is a window stack, which holds all parameters for scan pattern generation. Figure 6-13 illustrates a window stack example. The parameters required for a current stepper operation are accessed via a stack window. The parameter set for video scan generation requires 14 stack locations. It consists of four subsets of parameters:

- Base stepper parameters,

- Address stepper parameters,
- Limit stepper parameters, and
- global video scan parameters.

Scan Pattern Generation

For video scan generation based on the stack method the parameter window is initially placed at the global video scan parameter location. The *Scan Pattern Location* and the *Stepper Mode* are read. The *Stepper Mode* holds information for the stepper control unit. It specifies when the scan pattern is terminated, and what compound-, nested-, and meshed scans are called. For this it contains stack pointers to other video scans. Further it contains information, which escape condition controls the current video scan.

If the video scan is invoked the first time, the step counter is initialized by the *Initial Counter* value. Otherwise the previous state, which was saved to the *Reload Counter* stack position, will be restored.

After the global video scan parameters are read, the stack window moves to the Limit stepper parameters and generates a Limit (*L*) value. The parameter window moves to the Base stepper parameters and computes the a Base (*B*) value. The parameter window continues with the Address stepper parameters. Here the previously generated Base (*B*) and Limit (*L*) values are used. The stepper computes a complete scan line with the Address stepper parameters. If the scan line is completed, the parameter window moves again to the Limit stepper parameters. This proceeding is iterated until the video scan is finished or the stepper control unit interrupts the generation for a compound-, nested-, or meshed scan call.

A Complex Scan Example: The JPEG Zig-Zag Enumeration

In the following the application of the stack-based address generation with a real world example will be demonstrated. It will also be shown how the presented scan pattern classes can be used to find an efficient description of memory access sequences. It will be explained how a complex scan is composed from other classes to implement the zig-zag enumeration example for the run length coding step in JPEG image compression method [BK95].

The zig-zag enumeration reads the data of an image in a special order an writes it back sequentially. Figure 6-14 at page 117 shows the order of read operations. An image is processed in 8 by 8 pixel blocks. Each block is read from the upper left corner in zig-zag style to the lower right corner. After a block is read the next 8 by 8 block is processed.

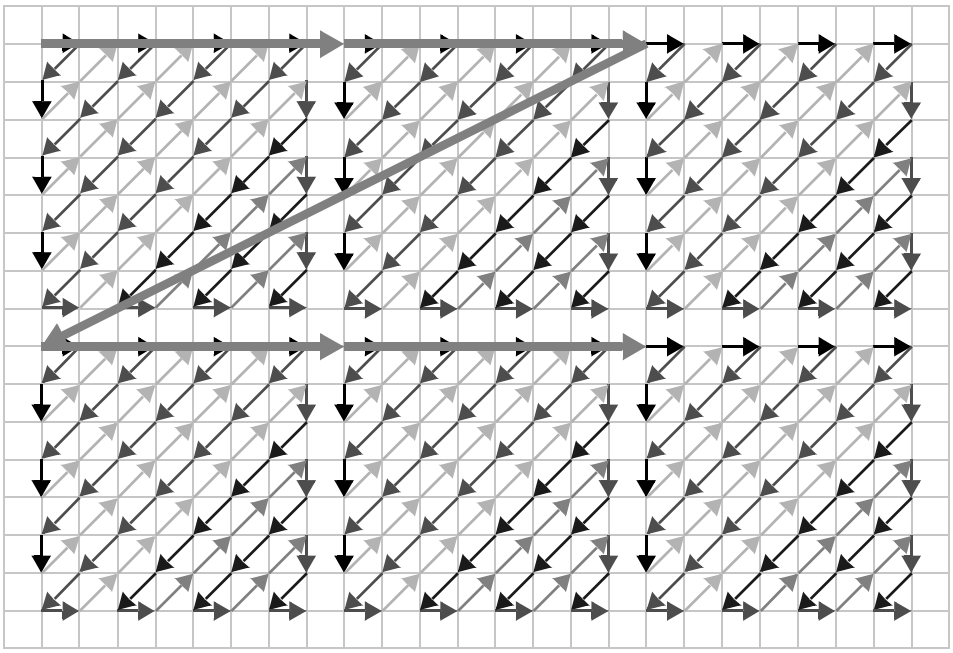


Figure 6-14: Run length coding scan for a 24x16 pixel image in JPEG image compression method.

To implement this application a video scan is needed, which covers the complete image to access the 8 by 8 pixel blocks. It performs 8 pixel steps and calls a complex scan to perform the zig-zag enumeration inside the 8 by 8 pixel block each step.

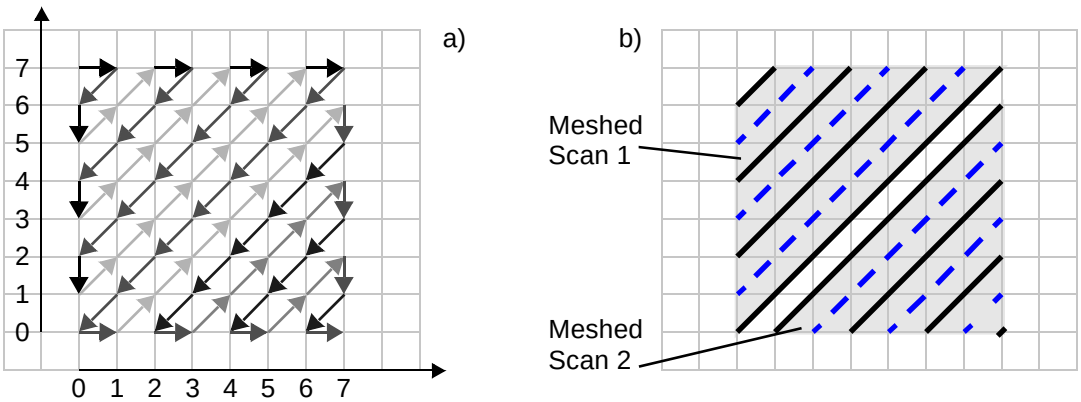


Figure 6-15: Inner scan of the JPEG zig-zag enumeration:
(a) scan steps, and
(b) generated by two meshed scans.

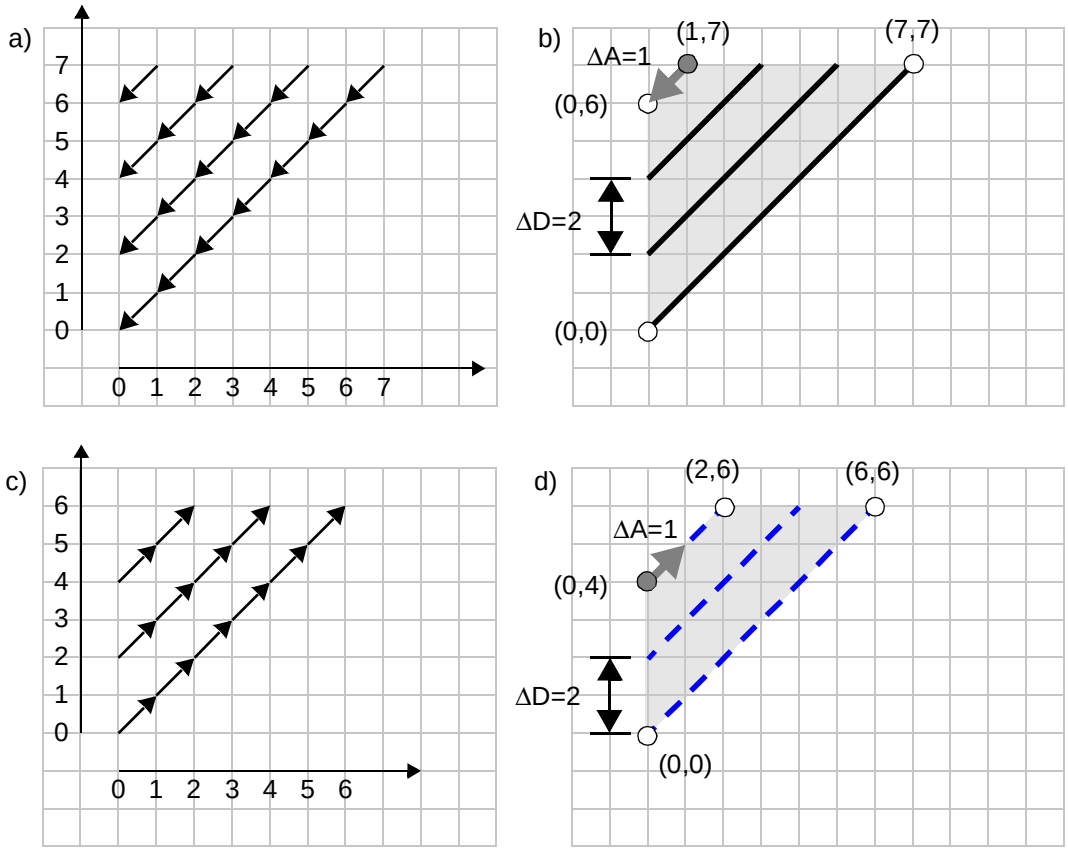


Figure 6-16: Meshed scan 1 of the JPEG zig-zag enumeration (see figur e6-15 at page 117):
(a) scan steps of video scan 1, and
(b) the covered area in the data memory, and
(c) scan steps of video scan 2, and
(d) the covered area in the data memory.

To implement the zig-zag enumeration of the 8 by 8 pixel block four video scans are combined to a complex scan. As demonstrated in figure 6-15b at page 117, one meshed scan (meshed scan 1) covers the upper left part and another meshed scan (meshed scan 2) covers the lower right part of the 8 by 8 pixel block. The two meshed scans form a compound scan.

Figure 6-16 and figure 6-17 at page 119 illustrate the video scans of meshed scan 1 and meshed scan 2. All involved video scans are designed to have one handle position in the origin of the coordinate system. During scan pattern generation, all these video scans are relative to the outer video scan. Therefore an offset is added (see figur e6-9 at page 110 for the hardware background) to perform the 8 by 8 pixel block at any position of the memory.

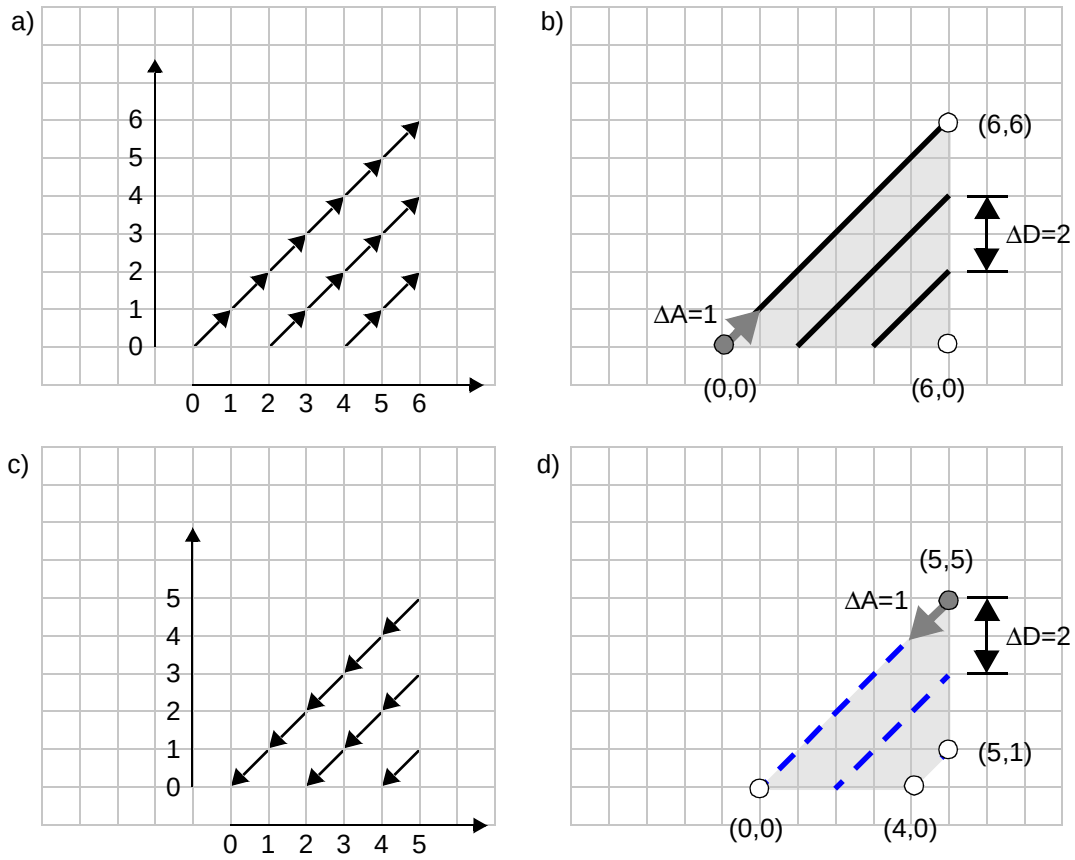


Figure 6-17: Meshed scan 2 of the JPEG zig-zag enumeration (see figure 6-15 at page 117):
(a) scan steps of video scan 1, and
(b) the covered area in the data memory, and
(c) scan steps of video scan 2, and
(d) the covered area in the data memory.

Meshed scan 1 does not start with the most upper left pixel of the 8 by 8 pixel block (see figure 6-16a at page 118), since this handle position has been generated by the outer video scan covering the complete image. In both meshed scans the video scans perform one diagonal row of the 8 by 8 pixel block. Since video scans may only increase or decrease the length of their scan lines, two meshed scans are required. More details on this implementation can be found in [Bra99].

Figure 6-18 at page 120 illustrates the parameter stack programming for the JPEG zig-zag enumeration example. The data sets for the five necessary video scans explained above are stored in the stack. Address generation is started with the outer video scan on top of the stack. The movement of the stack window is shown on the left part of figure 6-18 at page 120. The conditions, when the current video scan is changed, are written in cursive text.

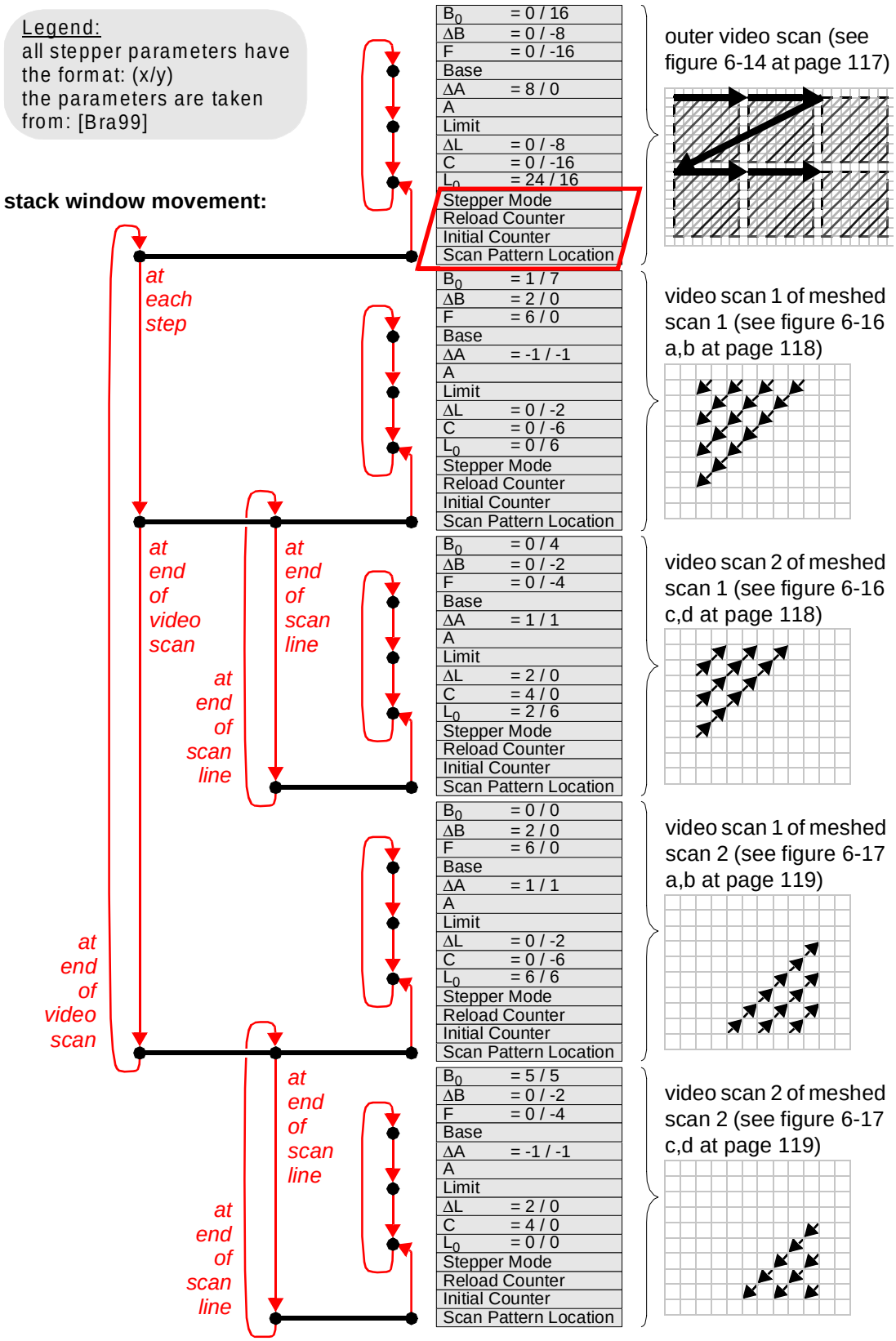


Figure 6-18: Stack based implementation of the JPEG zig-zag enumeration.

MoPL Program for the JPEG Zig-Zag Enumeration

The following MoPL (see appendi xB) program gives an alternative description of the JPEG Zig-Zag Enumeration example:

```
/* Other declarations */
Array          PixMap [1:8,1:8,15:0];
ScanPattern    EastScan        is 1   step  [ 1, 0];
               SouthScan       is 1   step  [ 0, 1];
               SouthWestScan   is 7   steps [-1, 1];
               NorthEastScan   is 7   steps [ 1,-1];

               UpLzigzagScan is
               begin
                    while (@[<8,])
                    begin      Eastscan;
                               SouthWestScan  until @[≤1,];
                               SouthScan;
                               NorthEastScan  until @[,≤1];
                    end;
               end UpLzigzagScan;

               JPEGzigzagScan is
               begin
                    UpLzigzagScan;
                    SouthWestScan;
                    rotu (reverse(UpLzigzagScan));
               end JPEGzigzagScan;
...
begin
     ...
     moveto PixMap [1,1];
     JPEGzigzagScan;
end;
```

6.5 The Data Sequencer Mapped to the KressArray

This section describes the KressArray implementation of the above data sequencer concept. The main characteristic of an integrated data sequencer is, that it is situated in the same device as used for data manipulations. In the area of reconfigurable computing the capability of reprogramming the hardware structure may be exploited. Additionally this section presents a method to generate application specific data sequencers, which are still programmable for specific data sets.

As target technology for the integrated data sequencer implementation the KressArray (section 2.7 at page 15) has been chosen. Basically coarse-grained reconfigurable architectures are preferred for data sequencer implementation, because all hardware components are datapaths. Therefore coarse-grained reconfigurable devices are better suited for this task. Related publications on area-efficiency of reconfigurable architectures are [BHH98], [DeH99], [DeH96b], [DeH96a], [Har97], [MHA97].

This section is structured as follows. First different memory communication models will be determined. After that the general data sequencer implementation will be described, and then the generation of application specific data sequencers will be explained. An application example of the presented method will be given in chapter 8.

The Graphical Representation of KressArray Operators

In this thesis two different views (see table 6-3 at page 123) of the KressArray operators are used:

- The schematic view is used, when the function of an implementation is illustrated.
- The layout view illustrates the mapping of a specific application generated by the MA-DPSS (see [HHH99a] or [HHH00]).

The schematic view illustrates DPUs as circles. If the DPU performs an operation, the operation symbol is pictured in the center of the circle and in some cases an operator number at the bottom. The operator number is written in cursive font. It is only printed if a layout of the schematic is available. Data input is illustrated with arrows ending at the circle border. The output of the operation result is shown with arrows originating from the circle border. If DPU internal data routing is illustrated (routing only or calculation and routing), the arrow, illustrating the signal flow, does not end at the DPU circle border but permeates the DPU.

In the layout view of the KressArray (used in figure 6-24a at page 132, figur e8-21 at page 214, and figure 8-23 at page 216) the DPU is pictured as a square. If the DPU performs an operation, the operation symbol is pictured in the upper left corner of the DPU. The operator number is written at the lower right corner. If there is a schematic and a layout view of the same application, DPUs use the same operator number in both views. Data in- and output of a DPUs operation is illustrated with arrows ending at the DPU border. Additionally a dot at the DPU border emphasizes this. If data routing is illustrated (routing only or calculation and routing), the arrow, illustrating the signal flow, is continued inside the DPU. Furthermore, there is not dot at the DPU border.

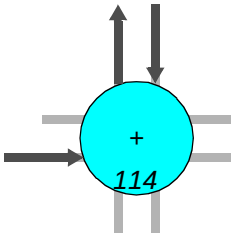
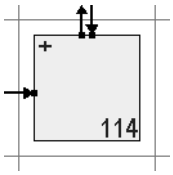
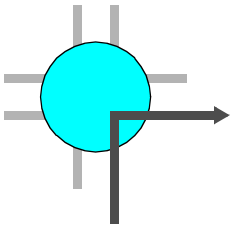
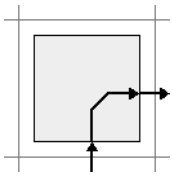
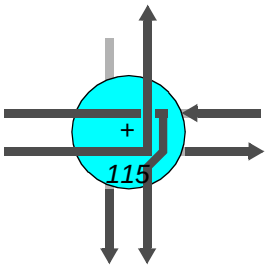
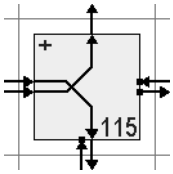
DPU Function	**Schematic view** used in the drawings	**Layout view** (used in figure 6-24 at page132, figure 6-24 at page132, figur e8-21 at page 214, and figure 8-23 at page 216) generated by the MA-DPSS (see [HHH99a] or [HHH00])
calculation only		
routing only		
calculation and routing		

Table 6-3: Graphical representation of DPUs used in this thesis.

6.5.1 Memory Communication Models for the KressArray

The utilized memory organization is an important topic for the integrated data sequencer implementation. While the memory communication paths in independent data sequencing are strongly limited by the number of available chip pins, integrated data sequencers are not restricted by this. In the following three different memory communication models will be discussed.

The Central Data Memory Model

The principle central data memory model is pictured in figure 3-2a at page 30 and figure 3-2b at page 30. In this model all data is stored in one data memory, which may be internal or external. To access the data memory a global address- and data bus is needed. All datapath units (DPUs) have access to these busses. The data sequencer may be implemented at any location in the KressArray. It generates addresses and applies

them to an address bus. The accessed data is sequenced from the central data memory to the DPU, which processes the data, or results are provided by a DPU to be stored in the central data memory.

The central data memory may be implemented with parallel memory banks as described in section 7.1 at page 147. The advantage of this memory communication concept is, that the data is stored at a central location. This simplifies the design of applications. The major disadvantage is the routing overhead of this approach. The central data memory eats up an enormous amount of routing resources to connect all DPUs to the data memory, but only a small fraction of the DPUs utilizes these resources for a given application.

The KressArray Surrounded by Data Memory With an Integrated Scan Window Generator

A KressArray architecture may also be surrounded by data memories with integrated hardwired scan window generators (SWG). A homogenous array of datapath units (DPUs) is located at the center of the architecture. All DPUs have nearest neighbor connections in four directions. Further they are connected to long range lines. At the border of the DPU array small memory units are located (see figure 3-2c at page 30). Each memory unit (see figure 6-19 at page 125) spans several rows or columns of DPUs. Inside the memory unit the following components are located:

- a SWG, which is directly connected to
- a small data memory, and
- a reconfigurable switch, which connects the data lines of the data memory, the *handle_position* input of the SWG, and the external I/O lines with the local interconnect of the bordering DPUs and the long range lines. Further neighboring reconfigurable switches are directly interconnected as additional routing resource.

In this communication model one or more handle position generators (HPG) are implemented at any location of the reconfigurable array. The generated handle position is passed via any routing resource to the target memory unit(s). The SWG inside the memory unit performs scan window accesses on the basis of the handle position to the internal data memory. Data is also routed via any available routing resource to the targeted DPU.

The benefit of this approach is that no extra routing resources to an external memory are needed. The same interconnect is used for computational datapath implementation and memory communication. The integration of the SWG into the memory unit additionally saves routing resources but also restricts the design space, because the assignment between the SWG and the memory bank is fixed.

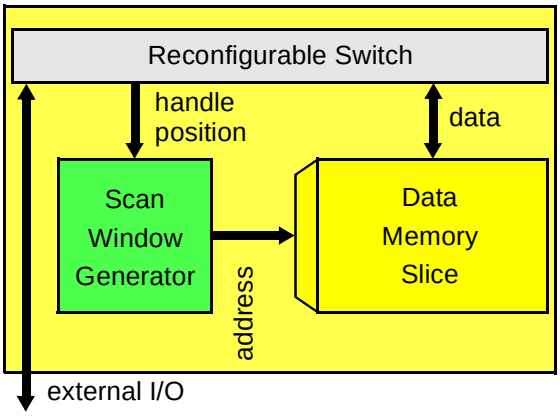


Figure 6-19: Memory unit with integrated scan window generator.

The KressArray Surrounded by Data Memory Only

A KressArray architecture may also be surrounded by data memories only. A homogenous array of datapath units (DPUs) is located at the center of the architecture. All DPUs have nearest neighbor connections in four directions. Further they are connected to long range lines. At the border of the DPU array small memory units are located (see figure 3-2c at page 30). Each memory unit (see figure 6-20 at page 126) spans several rows or columns of DPUs. Inside the memory unit the following components are located:

- a small data memory, and
- a reconfigurable switch, which connects the address- and data lines of the data memory, and the local I/O lines with the local interconnect of the bordering DPUs and the long range lines. Further neighboring reconfigurable switches are directly interconnected as additional routing resources.

In this communication model one or more data sequencers are configured to any location in the reconfigurable array. To route addresses and data, any of the routing resources may be utilized. Similar to the communication model above, no extra routing resources to interface an external memory are needed. But in contrast to the approach before the assignment between scan window generator and memory bank is not fixed. This may need some additional routing resources but allows to map any data sequencer hardware.

6.5.2 The KressArray Implementation of the Data Sequencer

This subsection will present the general data sequencer implementation for the KressArray. While there are multiple KressArray architectures (see section 2.7 at page 15) the basis for the data sequencer implementation is an array with two nearest neighbor connections with programmable direction. Further each datapath unit (DPU)

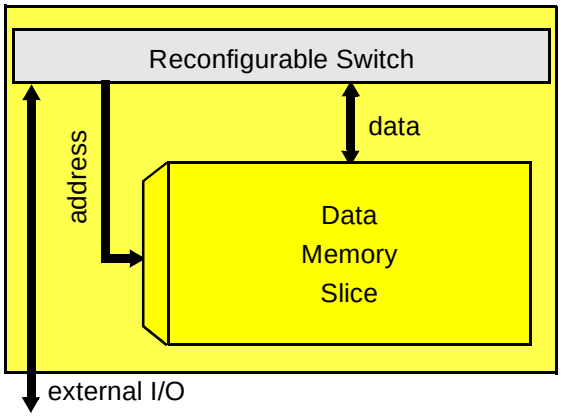


Figure 6-20: Memory unit without scan window generator.

has one reconfigurable ALU. If there are more ALUs inside a DPU, or sequential execution like known from the CHESS array (section 2.9 at page 19) is supported, several operators might be configured to only one DPU (see example in section 8.6.2 at page 215).

The proposed implementation benefits from the inherent pipelining of the KressArray, i.e. all operators work in parallel. In the following the required operators will be explained and after that, a general data sequencer implementation will be presented.

The KressArray Implementation of the Stepper

While the standard operator set of the KressArray provides all functions known from the programming language C [Kre96], for data sequencer implementation some specialized operators are needed. The stepper (see figure 6-21 at page 127) is implemented with two DPUs. The implementation corresponds to the concept presented in section 6.3.1 at page 109.

The address generating part of the stepper is performed by a special step generator also referred to as "St". The step generator calculates *Slider Positions* on the basis of the parameters *Initial Position* and *Step Width*. The result (*Slider Position*) of the step generator has to be checked against the *End Value*. The evaluation of the escape unit is performed by a standard "greater as" ("<") relation. In contrast to the stepper concept presented in section 6.3.1 at page 109, where the performed relation is selected by the sign of the *Step Width*, the performed relation of the KressArray implementation is configured into the escape unit. This simplifies the KressArray implementation of the stepper but requires a reconfiguration, if the sign of the *Step Width* changes. Figure 6-21 at page 127 pictures the partitioning of the stepper onto two DPUs and illustrates its KressArray representation.

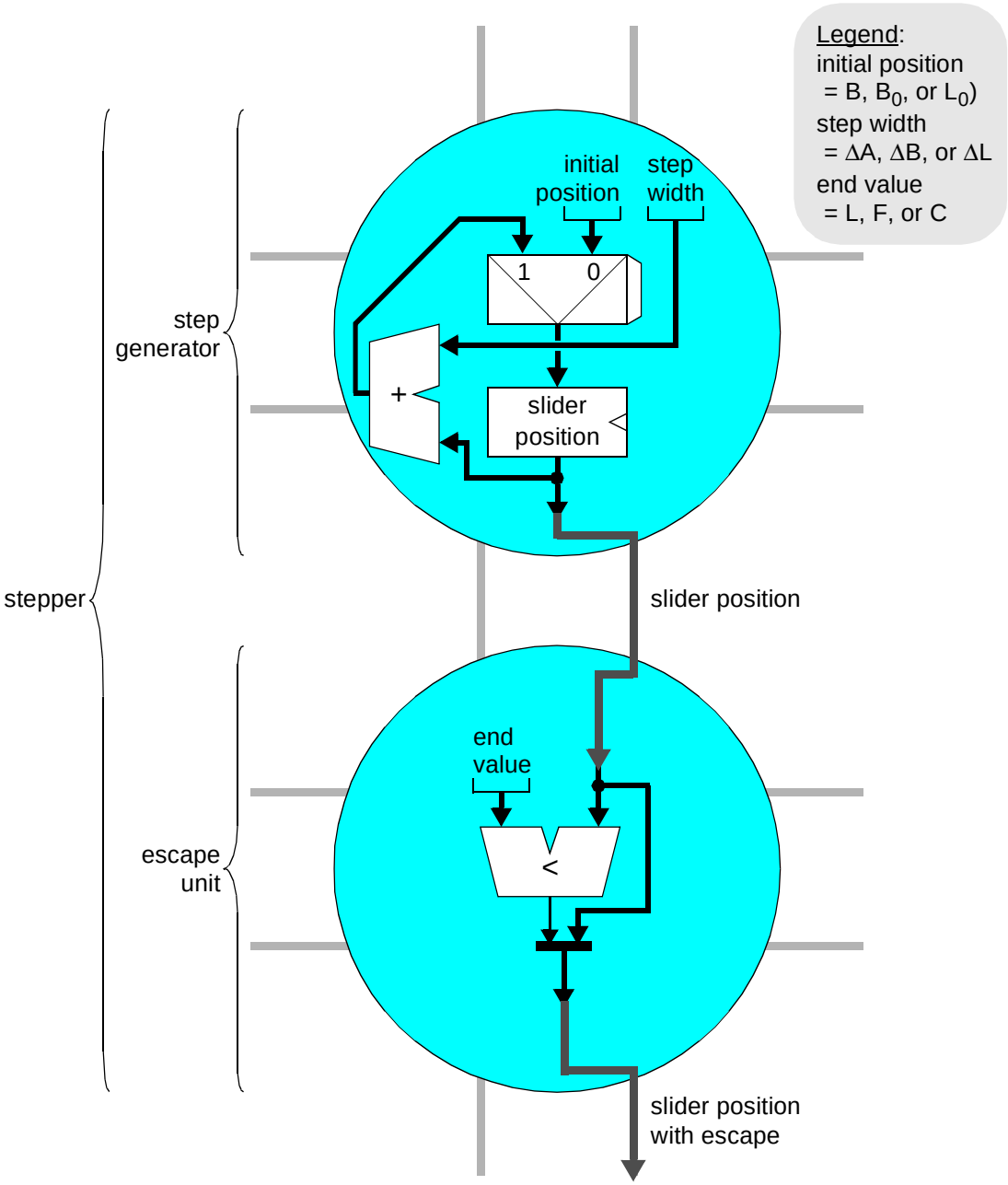


Figure 6-21: The KressArray implementation of a stepper (see also fi gure6-9 at page 110 and figure 6-10 at page 111).

The 2-dimensional Video Scan Generator

As described in section 6.3.3 at page 113 the 2-dimensional video scan generator consists of 6 steppers, three for each dimension. While the Base- and Limit-steppers operate on configured constant parameters only, the Address-steppers use the results generated by the Base- and Limit-steppers as an input value.

After the handle position generation the memory mapping as described in section 7.1.2.3 at page 151 is performed by a mapping shifter operator. The mapping shifter operator is programmed by a constant, which specifies the address mapping. It reads two addresses (the x- and y-parts of the handle position) A_x and A_y with the attached overflow bits. If an overflow bit has been set, no output is generated. In that case the next input values are read. Otherwise A_x and A_y are merged to one address depending on the memory width constant (see also figure D-12 at page 276). This value forms the output.

The low level data sequencing (see figure 6-2 at page 92) is look-up table-based, which can be integrated into a single DPU, called "SWG". The scan window generator (SWG, see figure 6-4 at page 96) holds a look-up table with the definition of all scan window positions. It has one input, where it reads a handle position, and one address output, where a memory bank is connected. If parallel memory banks have to be driven, one "SWG" is required for each memory bank. In such a case the address outputs of all SWGs have to be assigned dynamically to the memory banks according to the scheme presented in figure 7-4 at page 151. Such an implementation is shown in figure 8-18 at page 210.

Figure 6-22 at page 129 illustrates the schematic of the basic KressArray implementation of a data sequencer capable to execute single video scans. This general data sequencer implementation may serve as a template to generate application specific data sequencers, which are still programmable for different data sets (see section 6.5.3 at page 133). The basic data sequencer has been mapped to the KressArray architecture, where the KressArray is surrounded by memory units, which integrate already a SWG. Therefore the SWG as shown in figure6-22 at page129 is not mapped to a DPU.

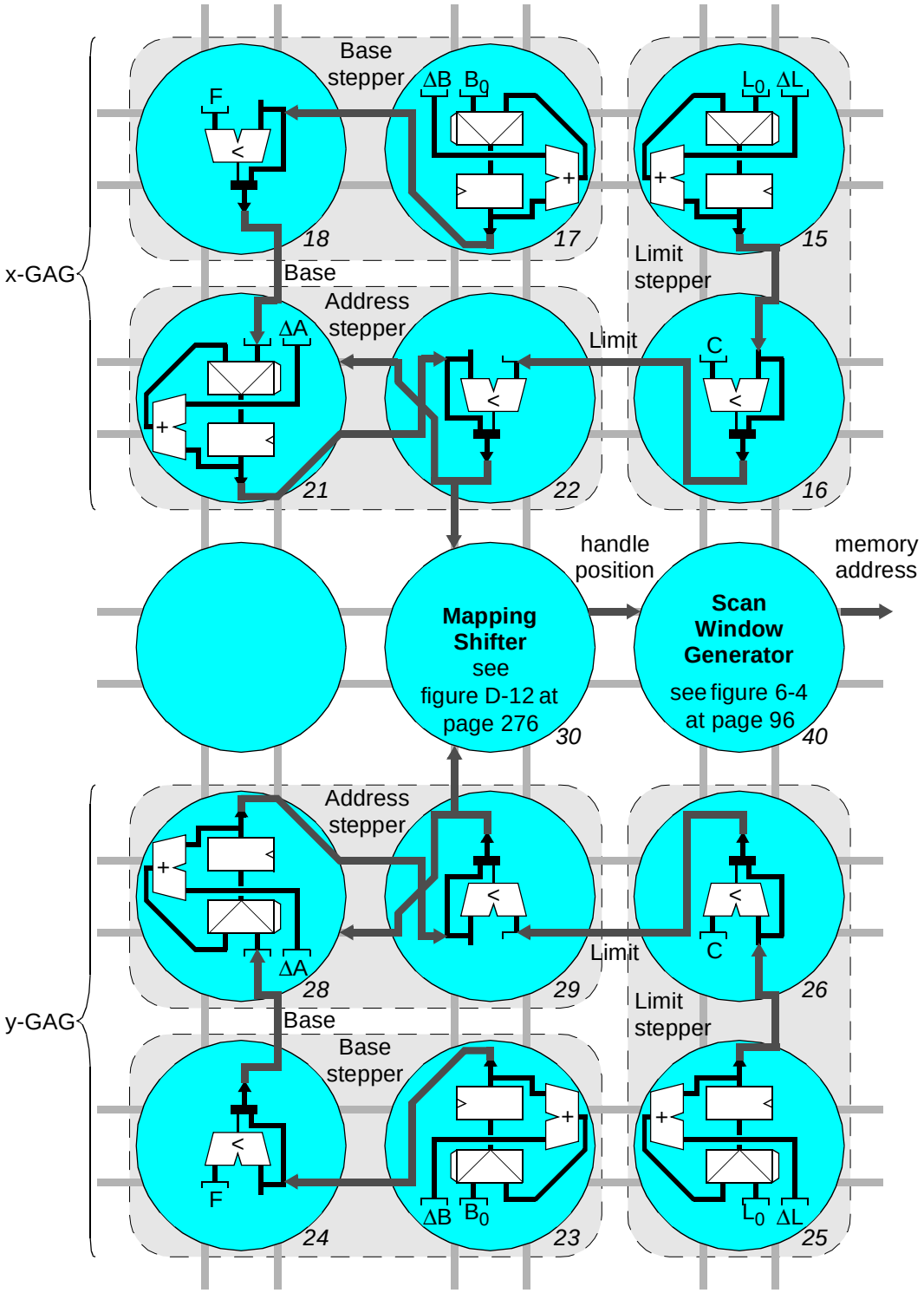


Figure 6-22: The basic KressArray implementation of the data sequencer. For details on the stepper implementation see figure 6-21 at page 127.

Required Routing Resources for the KressArray Mapping of the 2-dimensional Video Scan Generator

This subsection analyses the required routing resources needed for a satisfactory KressArray mapping of the 2-dimensional video scan generator. Therefore the data sequencer shown in figure 6-22 at page 129 is mapped to several KressArray architectures of the same functionality but with different numbers and types of nearest neighbor (NN) connections. The mappings have been performed using the MA-DPSS (see [HHH99a] or [HHH00]). Four different architectures have been explored.

The operators pictured in figure 6-22 at page 129, figure 6-23 at page 131, and figure 6-24 at page 132 are numbered to support the reader to follow up the mapping by the MA-DPSS. Since the layout generator of the MA-DPSS supports only textual operator naming, the step generator is named "St", and the escape unit is named "<".

- KressArray with single directed NN port:
  This local routing scheme corresponds to the KressArray-1 infrastructure (see section 2.2 at page 7). The mapping is illustrated in figure 6-23a at page 131. To achieve a full routing the MA-DPSS used three global bus connections. The rare use of local routing may be caused by the interconnect structure of the data sequencer. Since global bus connections are much slower than NN connections, this array structure is unsuited to map the data sequencer.
- KressArray with single bidirectional NN port:
  The mapping is illustrated in figure 6-23b at page 131. To achieve a full routing the MA-DPSS used one global bus connection. Even the intensive utilization of local routing fabrics could not avoid a global bus cycle. This is caused by the rare availability of NN ports, which are exhausted for nearly all DPUs. Since the global bus connection is much slower than NN connections, this array structure is also unsuited to map the data sequencer.
- KressArray with two bidirectional NN ports:
  The mapping is illustrated in figure 6-24a at page 132. For this array structure the MA-DPSS found a good mapping. The mapper managed to unscramble the operators, and thus many NN ports are still free for other connections. The basis for this is the second NN port, which is needed in the interconnect intensive parts of the data sequencer design.
- KressArray with three bidirectional NN ports:
  The mapping is illustrated in figure 6-24b at page 132. Also for three NN ports the MA-DPSS found a good mapping. Similar to the array with two NN ports the operators have been unscrambled, but here an even more uniform distribution of the connections is found. This distribution leaves more fabrics for routing of application datapaths. But compared to the mapping achieved with the KressArray with two NN ports this architecture provides no important improvement.

All mapping results are summarized in figur e6-25 at page133.

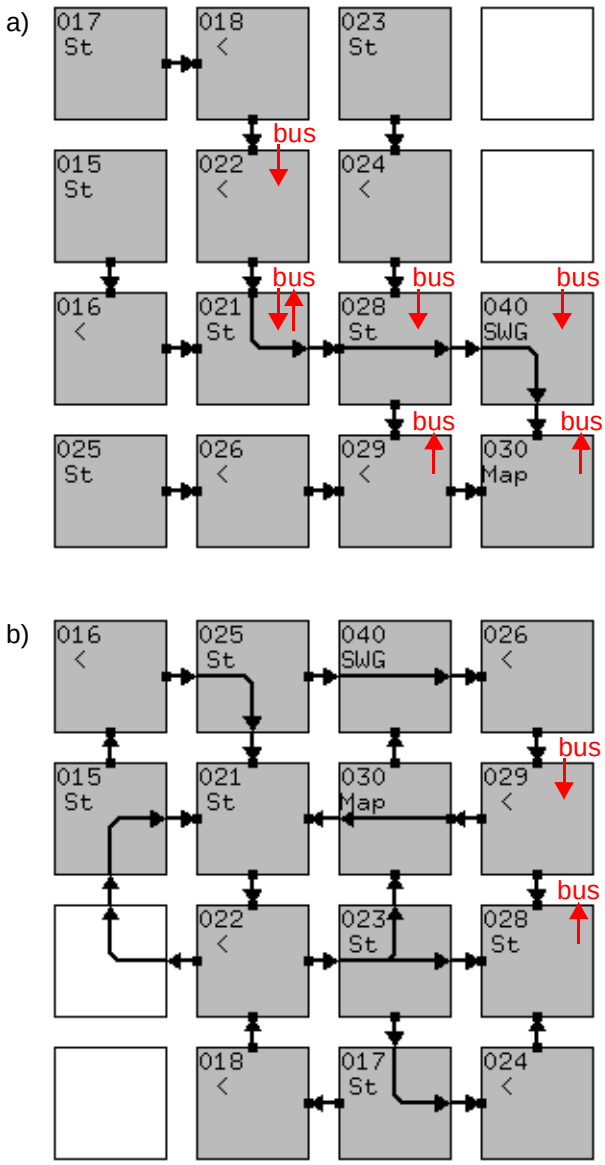


Figure 6-23: KressArray mappings generated by the MA-DPSS (see [HHH99a] or [HHH00]) of the data sequencer shown in figur e6-22 at page 129:
(a) using an array with one directed nearest neighbor (NN) port, and
(b) using an array with one bidirectional nearest neighbor (NN) port.

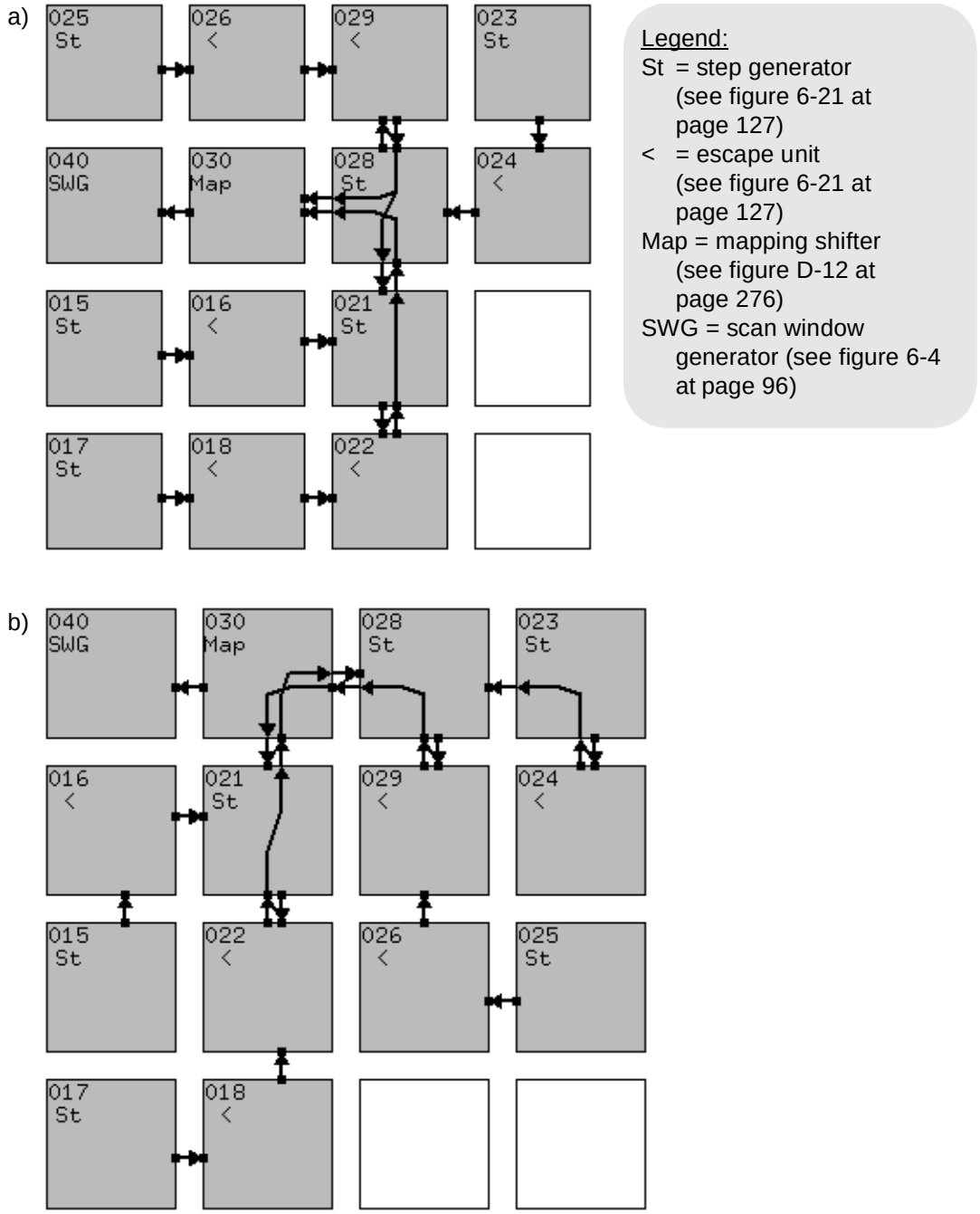


Figure 6-24: KressArray mappings generated by the MA-DPSS (see [HHH99a] or [HHH00]) of the data sequencer shown in figure 6-22 at page 129:
(a) using an array with two bidirectional nearest neighbor (NN) ports, and
(b) using an array with three nearest neighbor (NN) ports.

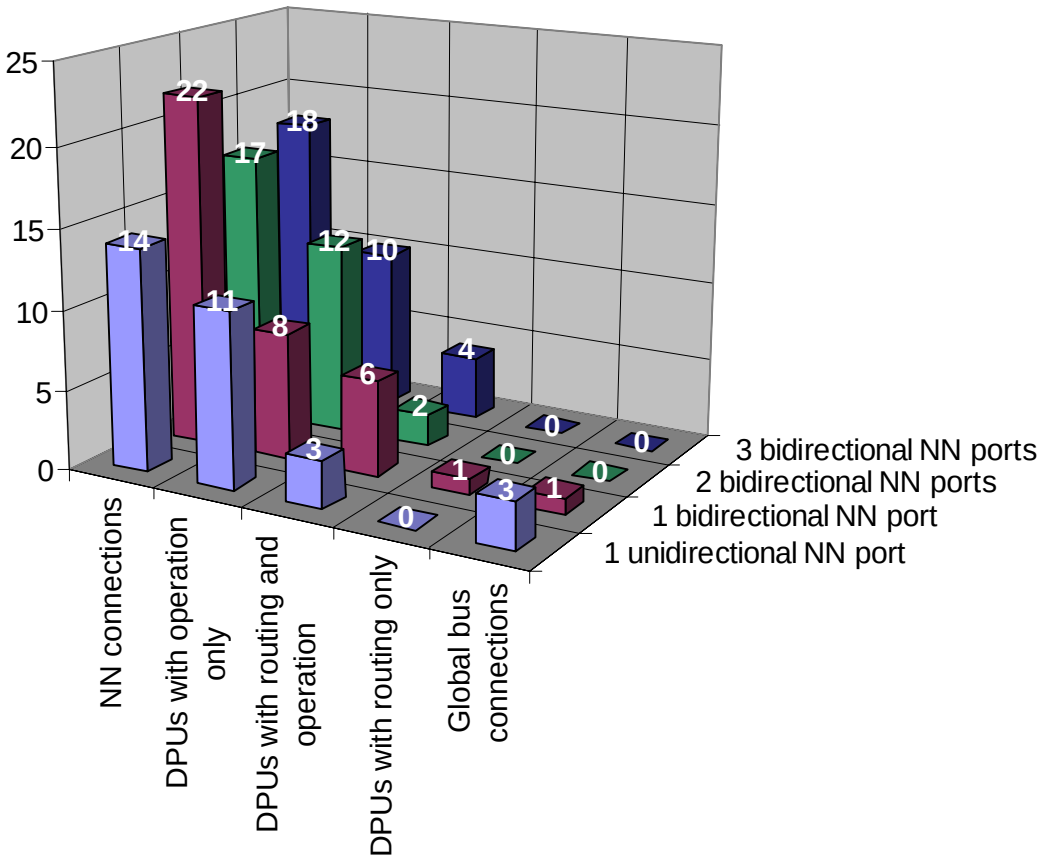


Figure 6-25: Routing requirements needed to map the data sequencer shown in figure 6-22 at page 129 to KressArrays with different fabrics.

6.5.3 The Generation of Application Specific Data Sequencers

In this subsection a method will be presented, which allows to save reconfigurable resources by generating application specific data sequencers. Since a reconfigurable device has to be configured to fix the computational datapath, also the data sequencer mapped to the same device may be reconfigured. Using this programming model it is no restriction if an application specific data sequencer is used.

The universal data sequencer design serves as a template for the generation of application specific data sequencers. A given application is examined and parts of the universal data sequencer are identified to be not needed for that particular application. The resulting application specific data sequencer still benefits from the generic data sequencer concept, because it is still programmable by the same parameters. Therefore it may be programmed for different data sets of the same shape.

In the following the application specific data sequencer implementations for

- linear scans,
- rectangular video scans, and
- triangle, trapezium, and parallelogram shaped video scans

are presented. A detailed discourse on this topic can be found in [Ero99].

Linear Scan Generation

Figure 6-26 illustrates different linear scan examples. To specify an application specific data sequencer for linear scans two types of linear scans are identified:

- linear scans parallel to a coordinate axis, and
- diagonal video scans.

A diagonal linear scan is depicted in figur e6-26b. It starts handle position generation at $\overline{B_0}$ position and ends at $\overline{L_0}$ position. Further equation 6-1, equation 6-2, and equation 6-3 are generally valid.

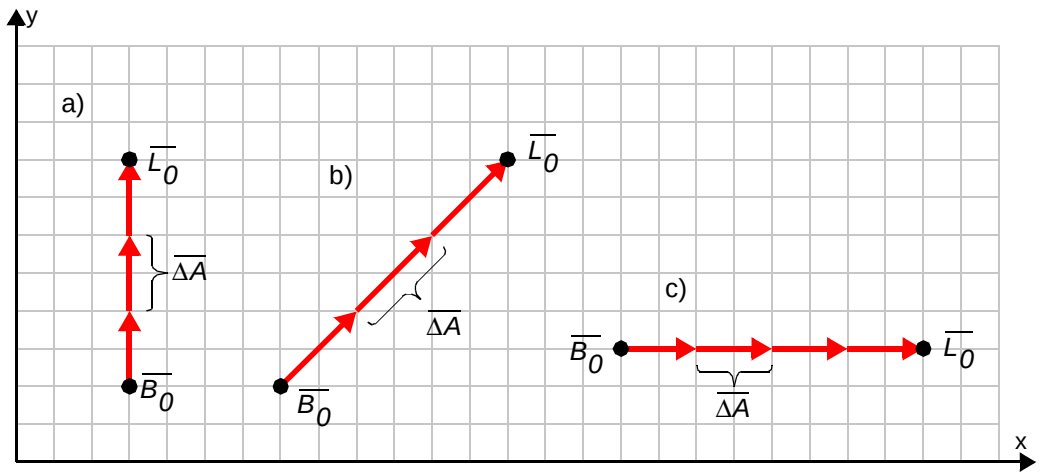


Figure 6-26: Handle position sequences of linear scan examples:
(a) parallel to the y-axis,
(b) diagonal, and
(c) parallel to the x-axis.

$$\overline{B_0} = \overline{F} \qquad \textit{(Eq. 6-1)}$$

$$\overline{L_0} = \overline{C} \qquad \textit{(Eq. 6-2)}$$

$$\overline{\Delta B} = \overline{\Delta L} = \overline{0} \qquad \textit{(Eq. 6-3)}$$

Since Base (*B*) and Limit (*L*) never change (because of equation 6-3 at page 134) the Base-, and Limit Steppers are not needed. Figure 6-28b at page 136 illustrates the application specific data sequencer for diagonal linear scans.

A linear scan parallel to the x-axis is shown in figure 6-26c at page 134, and a linear scan which is parallel to the y-axis is pictured in figure 6-26a at page 134. The optimization of the data sequencer for this type of scan pattern equivalent. Only x and y change. Here the application specific data sequencer for a linear scan parallel to the y-axis is described.

A linear scan parallel to the y-axis never moves in the x-direction, therefore equation 6-4, and equation 6-5 hold in addition to equation 6-1 at page 134, equation 6-2 at page 134, and equation 6-3 at page 134.

$$B_{0x} = L_{0x} \qquad \textit{(Eq. 6-4)}$$

$$\Delta A_x = 0 \qquad \textit{(Eq. 6-5)}$$

As a result the x-part of the handle position is constant B_{0x}. The data sequencer may be optimized as illustrated in figure 6-28a at page 136. The constant part of the handle position is configured to the mapping shifter operator for this task.

Rectangular Video Scan Generation

Figure 6-27 shows two rectangular video scans which are parallel to the x- and y axis. For both video scans the data sequencer may be optimized in the same way only x and y have to be exchanged. Therefore here only the application specific data sequencer for a video scan, where the scan lines are parallel to the y-axis (figure 6-27a), will be explained.

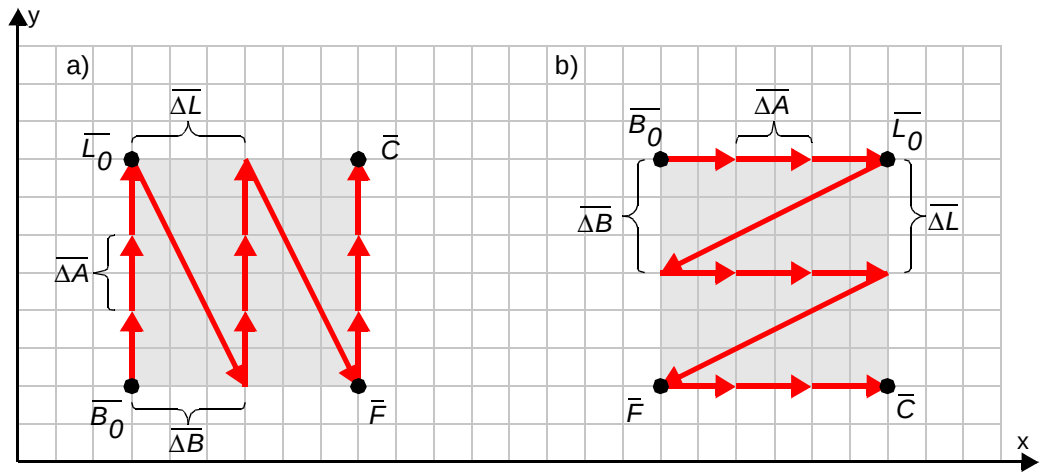


Figure 6-27: Handle position sequences of rectangular video scan examples:
(a) scan line parallel to the y-axis, and
(b) scan line parallel to the x-axis.

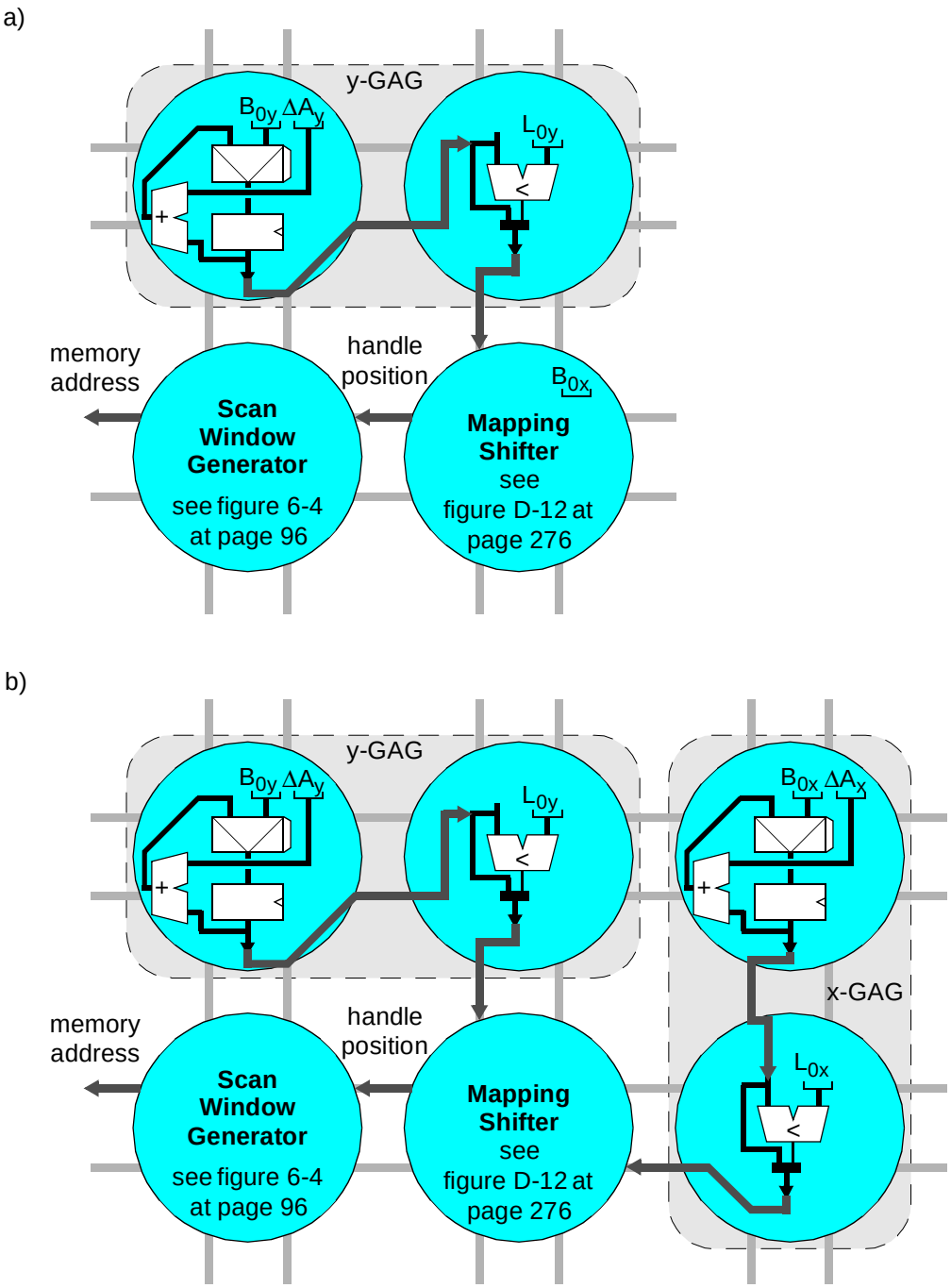


Figure 6-28: Application specific data sequencers:
(a) to generate a linear scan parallel to the y-axis as pictured at figure 6-26a at page 134, and
(b) to generate a diagonal scan as shown at fi gure6-26b at page 134.

Because of the video scan shape the following conditions hold for all rectangular video scans, where the scan lines are parallel to the y-axis:

$$B_{0x} = L_{0x} \qquad \textit{(Eq. 6-6)}$$

$$F_x = C_x \qquad \textit{(Eq. 6-7)}$$

$$B_{0y} = F_y \qquad \textit{(Eq. 6-8)}$$

$$L_{0y} = C_y \qquad \textit{(Eq. 6-9)}$$

$$\Delta A_x = 0 \qquad \textit{(Eq. 6-10)}$$

$$\Delta B_y = 0 \qquad \textit{(Eq. 6-11)}$$

$$\Delta L_y = 0 \qquad \textit{(Eq. 6-12)}$$

The performed optimizations are based on equation 6-10, equation 6-11, and equation 6-12. From equation 6-10 can be derived, that the Address stepper for the x-dimension is not active during scan line generation. Also the Base-, and Limit steppers of the y-dimension are inactive (equation 6-11 and equation 6-12). Therefore the parts of the universal data sequencer (see figure 6-22 at page 129), which generate these values may be omitted. The application specific data sequencer is pictured in figure 6-29.

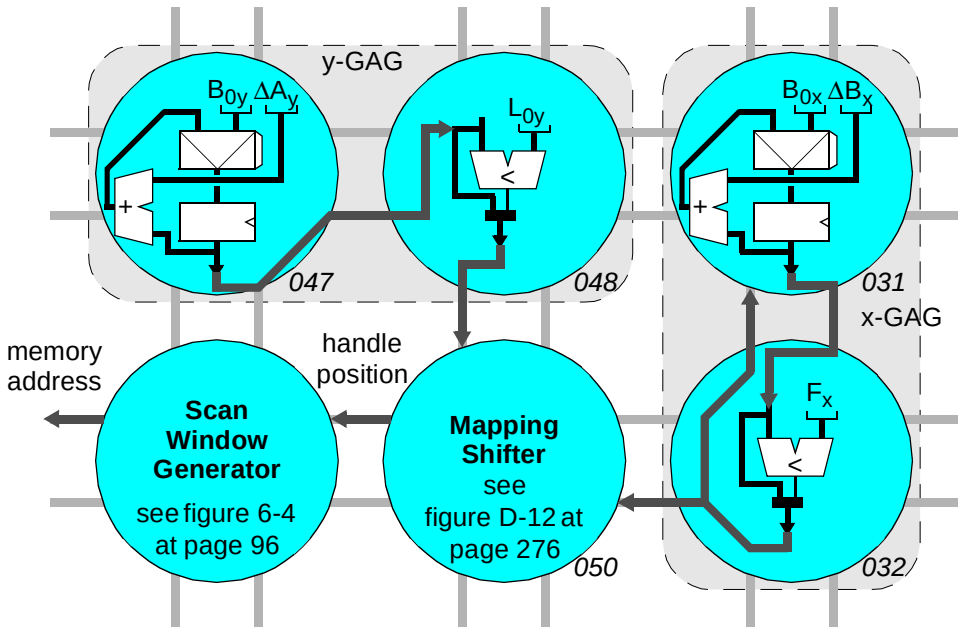


Figure 6-29: Data sequencer for a rectangular video scan which is parallel to the x- and y-axis (see figure 6-27a at page 135). The operators numbers are used in the layout in figure 8-21 at page 214.

The y-GAG generates scan lines, which start always at B_{0y} and end always at L_{0y}. The stepwidth is ΔA_y. During scan line generation the x-part of the handle position is constant. The x-GAG generates the x-part of the handle positions, which is valid for a complete scan line. It starts at B_{0x} and runs in ΔB_x steps until F_x is reached.

Triangle, Trapezium, and Parallelogram Video Scan Generation

The application specific data sequencer for triangle, trapezium, or parallelogram video scans is more complicated and achieves less good optimization results in area requirements. The best optimization of the data sequencer for such applications is achieved for scans which are parallel to an axis and the other edges are isosceles. Examples for the three video scan shapes are shown in figure 6-30, figure 6-32 at page 139, and figure 6-31 at page 139. The scan lines of all examples are parallel to the x-axis but the same optimization may be performed to the data sequencer for video scan parallel to the y-axis. Only the x- and y- parameters have to be exchanged.

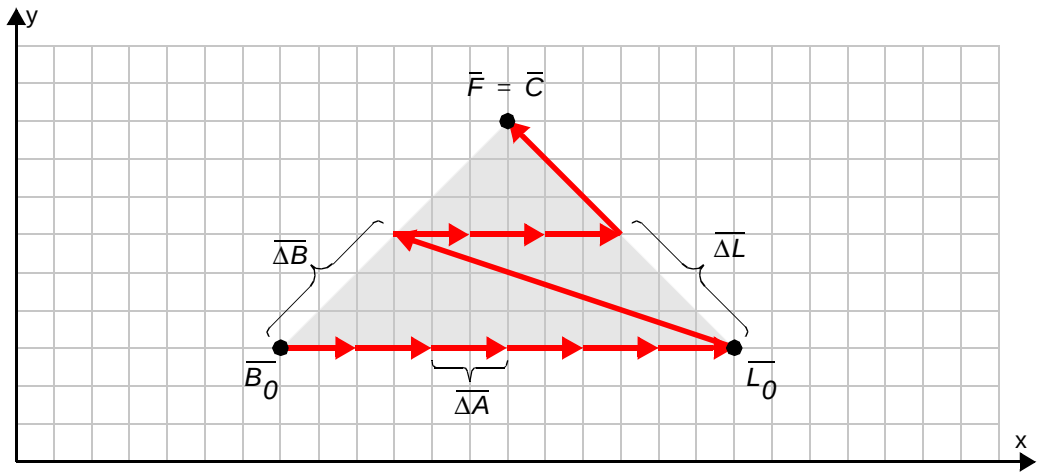


Figure 6-30: Handle position sequence of a triangular video scan example.

All video scans of the three shapes, where the scan lines are parallel to the x-axis, have the following properties:

$$B_{0y} = L_{0y} \qquad \textit{(Eq. 6-13)}$$

$$F_y = C_y \qquad \textit{(Eq. 6-14)}$$

$$\Delta A_y = 0 \qquad \textit{(Eq. 6-15)}$$

Here equation 6-15 is the main source of data sequencer optimization. Because of scan line movement is only performed in x-direction the most parts of the y-Stepper may be omitted as already described above for rectangular video scans.

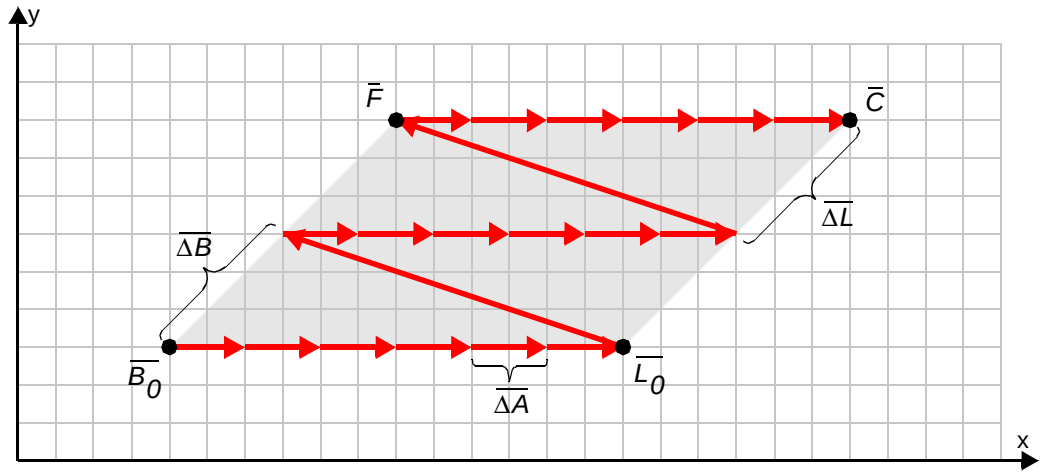


Figure 6-31: Handle position sequence of a parallelogram video scan example.

For triangular and trapezium scans the address generation in x-direction has the property that

$$\Delta B_x = -\Delta L_x \qquad \textit{(Eq. 6-16)}$$

The reason for this property is the isoscelesness of the video scans. Based equation 6-16 all x-parts of the Limit can be expressed by the following equation:

$$L_x = (2 \cdot B_{0x} - (2 \cdot (C_x - L_{0x})) + (C_x - F_x)) - B_x \qquad \textit{(Eq. 6-17)}$$

Since all values in equation 6-17 except the actual Base position are constants the Limit stepper may be replaced by a constant subtraction. The application specific data sequencer is pictured in figure 6-33 at page 140. The constant *c* is calculated as follows:

$$c = 2 \cdot B_{0x} - 2 \cdot (C_x - L_{0x}) + C_x - F_x \qquad \textit{(Eq. 6-18)}$$

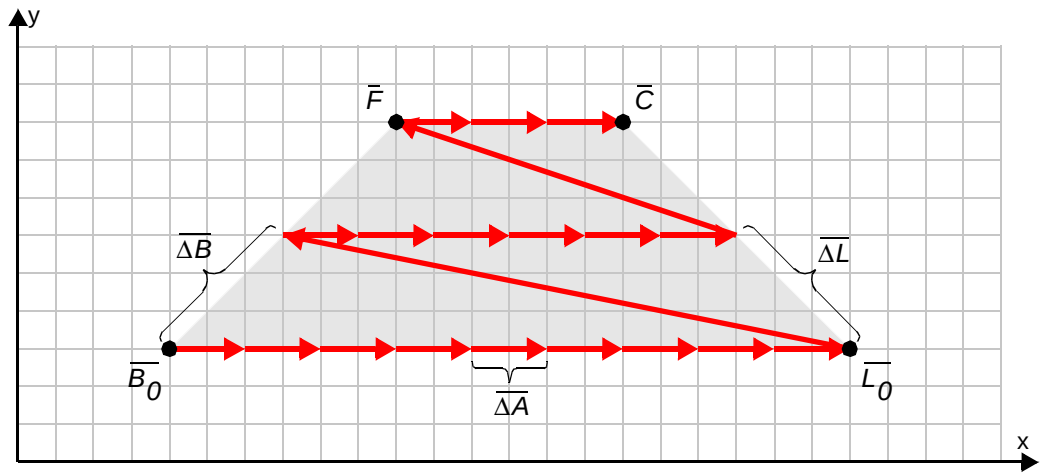


Figure 6-32: Handle position sequence of a trapezium video scan example.

In contrast to triangular and trapezium scans (equation 6-16 at page 139), parallelograms have the following property (see also figu re6-31 at pa ge139):

$$\Delta B_x = \Delta L_x \qquad \textit{(Eq. 6-19)}$$

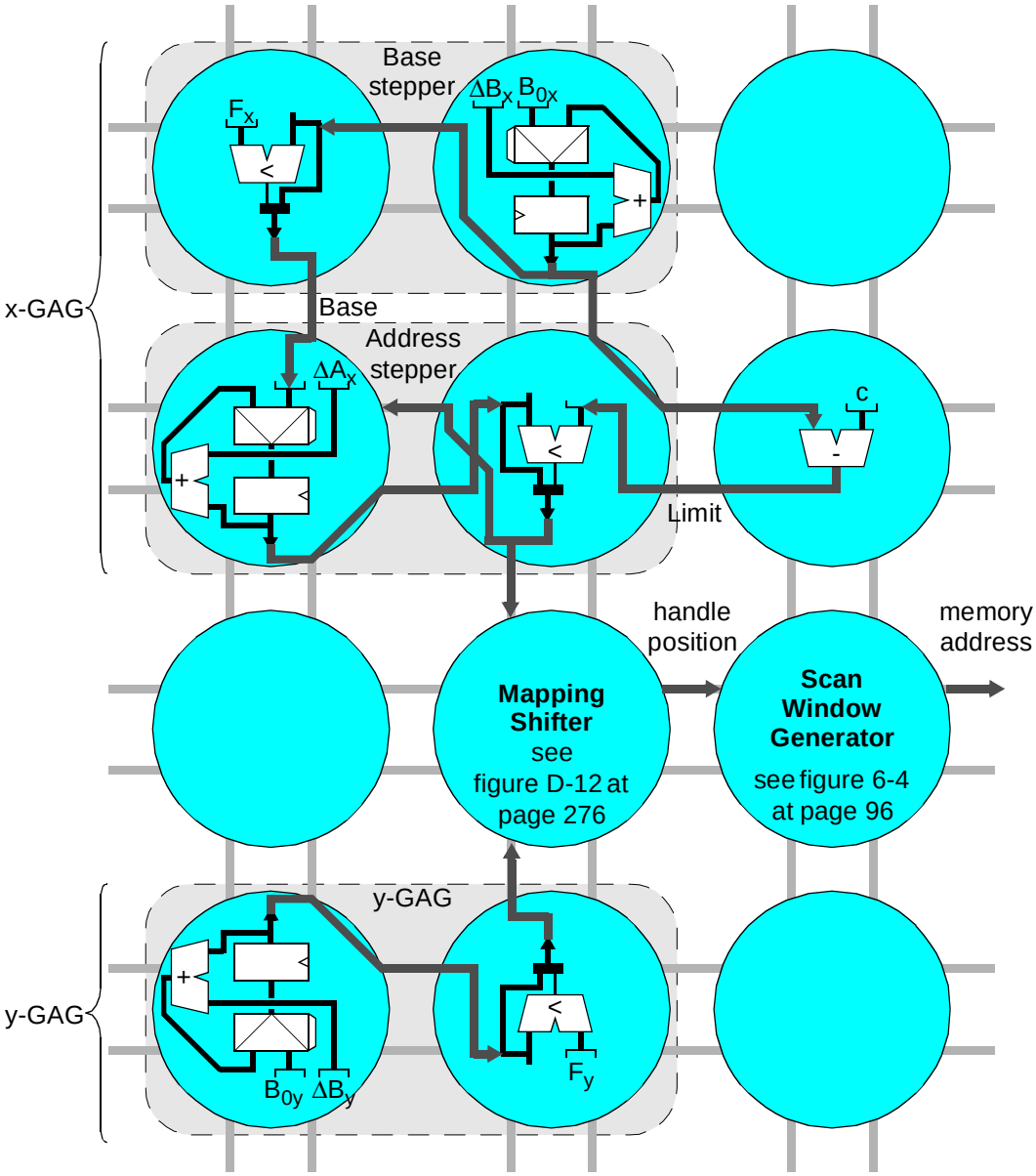


Figure 6-33: Application specific data sequencer to generate triangle (see figure 6-30 at page 138), trapezium (see figure 6-32 at page 139), and parallelogram (see figur e6-31 at page139) video scans, which are parallel to the y-axis.

Based on this property the x-part of the Limit for parallelograms may be expressed by the following equation:

$$L_x = (L_{0x} - B_{0x}) + B_x \qquad \textit{(Eq. 6-20)}$$

Since all values in equation 6-20 except the actual Base position are constants the Limit Stepper may be replaced by a constant addition. The required constant is calculated as follows:

$$c = L_{0x} - B_{0x} \qquad \textit{(Eq. 6-21)}$$

The application specific address generator is the same as for triangular and trapezium video scan (see figure 6-33 at page 140) except that the constant subtraction is replaced by a constant addition.

Optimization Results of Application Specific Data Sequencers

Figure 6-34 summarizes the hardware requirements of the different data sequencers. The diagram illustrates the number of DPUs and the number of local routing resources needed by the different implementations.

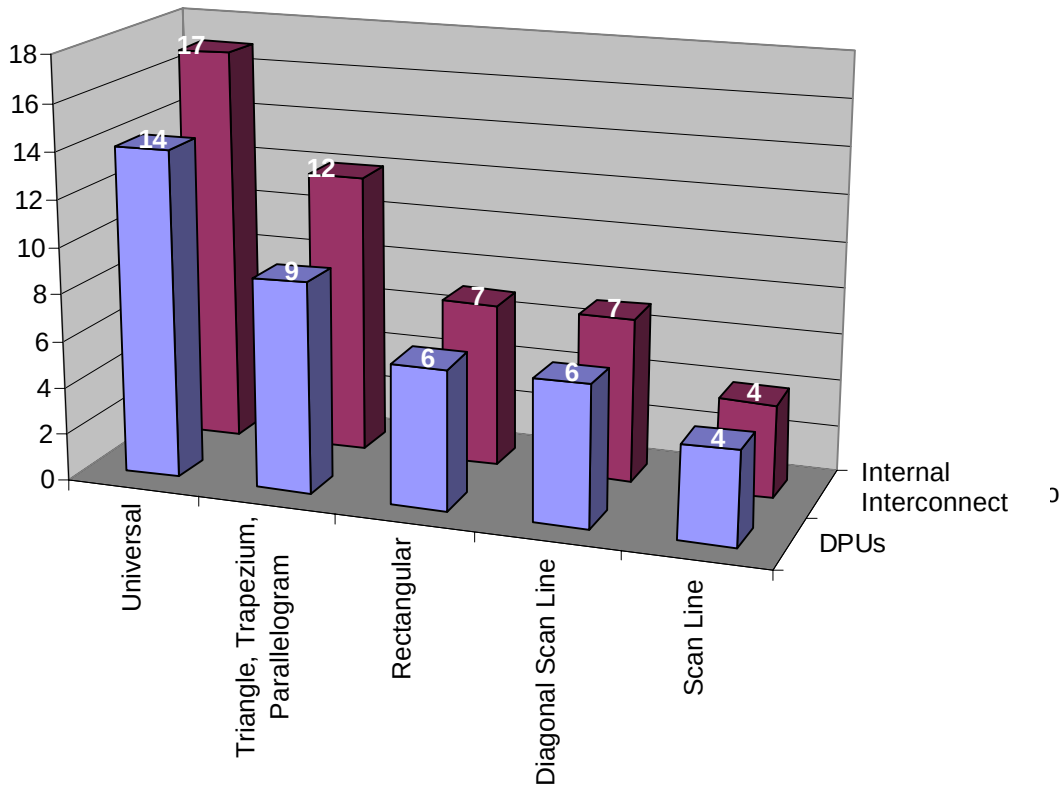


Figure 6-34: Area requirements of the presented application specific data sequencers.

The figures for the universal data sequencer have been taken from the mapping shown in figure 6-24a at page 132. The exact figures for the application specific data sequencer implementations are taken from [Ero99].

Some application specific data sequencers require much less hardware resources than the universal data sequencer. But the highly optimized data sequencers for scan lines, and single points are needed only exceptionally. But even for more common video scans like rectangular scans, good optimizations have been obtained. More than 50% of the initial hardware resources can be saved.

6.6 Chapter Summary

In this chapter a novel data sequencing concept for reconfigurable computing has been presented. The well defined Xputer paradigm with its two-level data sequencing forms a solid basis for the new data sequencing methodology. With the video scan as the atomic element a classification of access patterns provides powerful constructors for complex scan patterns. The video scan can be generated generically by only a few parameters using the slider method. On this basis a new stack-based control mechanism has been introduced, which enables flexible address generation without relying on software based sequencing methods. The presented concept addresses the several aspects of reconfigurable computing. Further it will be the basis for the subsequent access optimization techniques (chapter 7), which improve the memory communication performance.

The Overall Design Time

The overall design time can be reduced by the introduction of the common concept for data sequencing. By using the presented methodology during application implementation, not a data sequencer has to be designed, but a flexible data sequencer is programmed. This brings application development for reconfigurable systems closer to the programming style of conventional computers.

The Configuration Load

Ad hoc data sequencer implementations would require the reconfiguration of the complete data sequencer design. In contrast to this the presented method requires only the reprogramming of parameters to switch between applications. The generic address generator is capable to perform complex memory accesses on the basis of only a few parameters. The number of theses parameters may also be decreased by application specific address generator implementations.

Mapping Onto the KressArray

The new sequencing concept provides substantial flexibility to fit to different KressArray applications featuring a wide variety of memory bandwidth requirements. It supports also the generation of application specific sequencers based on a general template. This proceeding allows to perform a trade-off between flexibility and area requirements.