A. Ast, J. Becker, R. W. Hartenstein, et al.: Data-procedural Languages for FPL-based Machines; 4rd Int. Workshop On Field Programmable Logic And Applications, FPL'94, Prague, September 7-10, 1994, Lecture Notes in Computer Science, Springer, 1994

Data-procedural Languages for FPL-based Machines

A. Ast, J. Becker, R.W. Hartenstein, R. Kress, H. Reinig, K. Schmidt

Fachbereich Informatik, Universität Kaiserslautern
Postfach 3049, D-67653 Kaiserslautern, Germany
fax: (+49 631) 205-2640, e-mail: abakus@informatik.uni-kl.de

ABSTRACT. This paper introduces a new high level programming language for a novel class of computational devices namely data-procedural machines. These machines are by up to several orders of magnitude more efficient than the von Neumann paradigm of computers and are as flexible and as universal as computers. Their efficiency and flexibility is achieved by using field-programmable logic as the essential technology platform. The paper briefly summarizes and illustrates the essential new features of this language by means of two example programs.

1 Introduction

Usually procedural machines are based on the von Neumann machine paradigm. (Data flow machines are no procedural machines, since the execution order being determined by an arbiter is indeterministic.) Both, von Neumann machines, as well as von Neumann languages (Assembler, C, Pascal, etc.) are based on this paradigm. We call this a *control-procedural paradigm*, since execution order is control-driven. Because in a von Neumann machine the instruction sequencer and the ALU are tightly coupled, it is very difficult to implement a reconfigurable ALU supporting a substantial degree of parallelism.

By turning the von Neumann paradigm's causality chain upside down we obtain a data sequencer instead of an instruction sequencer. We obtain a new machine paradigm called a *data-procedural machine paradigm*. This new paradigm is the root of a new class of procedural languages which we call *data-procedural languages*, since the execution order is deterministically data-driven. This new data-procedural paradigm [1], [4], [5] strongly supports highly flexible FPL-based reconfigurable ALUs (rALUs) permitting very high degrees of intra-rALU parallelism. That's why this paradigm opens up new dimensions of machine architecture, reconfigurability, and hardware efficiency [4].

This paper introduces this new class of languages by using a data-procedural example language. The language MoPL-3 used here is a C extension. Such data-procedural languages support the derivation of FPL-based data path resource configurations and data sequencer code directly from data dependences. The usual detour from data dependences via control flow to data manipulation, as practiced by von Neumann program-

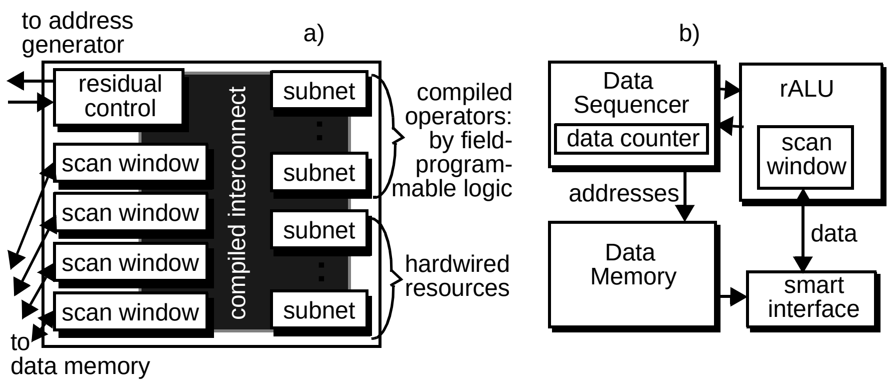


Fig. 1. Basic structures of Xputers and the MoM architecture: a) reconfigurable ALU (rALU) of the MoM, b) basic structure of Xputers

ming, is almost completely avoided. The paper illustrates data-procedural language usage and compilation techniques as well as their application to FPL-based hardware.

2 Summarizing the Xputer

For convenience of the reader this section summarizes the underlying machine paradigm having been published elsewhere [1], [4], [5], [6], [9]. Main stream high level control-procedural programming and compilation techniques are heavily influenced by the underlying von Neumann machine paradigm. Most programmers with more or less awareness need a von-Neumann-like abstract machine model as a guideline to derive executable notations from algorithms, and to understand compilation issues. Also programming and compilation techniques for Xputers need such an underlying model, which, however, is a data-procedural machine paradigm, which we also call *data sequencing paradigm*. This section summarizes and illustrates the basic machine principles of the Xputer paradigm [9]. Later on simple algorithm examples will illustrate MoPL-3, a data-procedural programming language.þ

2.1 Xputer Machine Principles

The main difference to von Neumann computers is, that Xputers have a *data counter* (as part of a data sequencer, see figure 1b) instead of a program counter (part of an instruction sequencer). Two more key differences are: a *reconfigurable ALU* called *rALU* (instead of a hardwired ALU), and *transport-triggered operator activation* ([11], instead of the usual control-flow-triggered activation). Operators are preselected by an *activate* command from a *residual control* unit. Operator activation is transport-triggered. Xputers are data-driven but unlike data flow machines, they operate deterministically by *data sequencing* (no arbitration).

Scan Window. Due to their higher flexibility (in contrast to computers) Xputers may have completely different processor-to-memory interfaces which efficiently support the exploitation of parallelism within the rALU. Throughout this paper, however, we use an Xputer architecture supported by smart register files, which provide a 2-dimensional scan windows (e.g. figure 2b shows one of size 2-by-2). A scan window gives

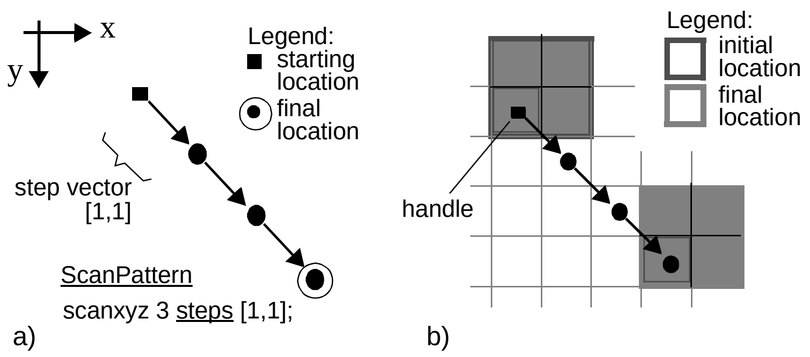


Fig. 2. Simple scan pattern example: a) source text and illustration, b) scan pattern moves a scan window by its handle

rALU access to a rectangular section of adjacent locations in data memory space. Its size is adjustable at run time.

Scan Pattern. A scan window is placed at a particular point in data memory space according to an address hold by a data counter (within a data sequencer). A data sequencer generates sequences of such addresses, so that the scan window controlled by it travels along a path which we call scan pattern. Figure 2a shows a scan pattern example with four addresses, where figure 2b shows the first and fourth location of the scan window. Figure 3 shows sequential scan pattern examples, and figure 9c a compound (parallel) scan pattern example.

Data Sequencer. The hardwired data sequencer features a rich and flexible repertory of scan patterns [4] for moving scan windows along scan paths within memory space. Address sequences needed are generated by hardwired address generators having a powerful repertory of generic address sequences [2], [13]. After having received a scan pattern code a data sequencer runs in parallel to the rest of the hardware without stealing memory cycles. This accelerates Xputer operation, since it avoids performance degradation by addressing overhead.

Reconfigurable ALU. Xputers have a reconfigurable ALU (rALU), which usually consists of global field-programmable interconnect (for reconfiguration), hardwired logic (a repertory of arithmetic, relational operators), and field-programmable logic (for additional problem-specific operators) [3]. Figure 1a shows an example: the rALU of the MoM-3 Xputer architecture: 4 smart register files provide 4 scan windows. A rALU has a hidden RAM (hidden inside the field-programmable integrated circuits used) to store the configuration code.

rALU Configuration is no Microprogramming. Also microprogrammable von Neumann processors have a kind of reconfigurable ALU which, however, is highly bus-oriented. Buses are a major source of overhead [7], especially in microprogram execution, where buses reach extremely high switching rates at run time. The intension of rALU use in Xputers, however, is to push overhead-driven switching activities away from run time, over to loading time as much as possible, in order to save the much more precious run time.

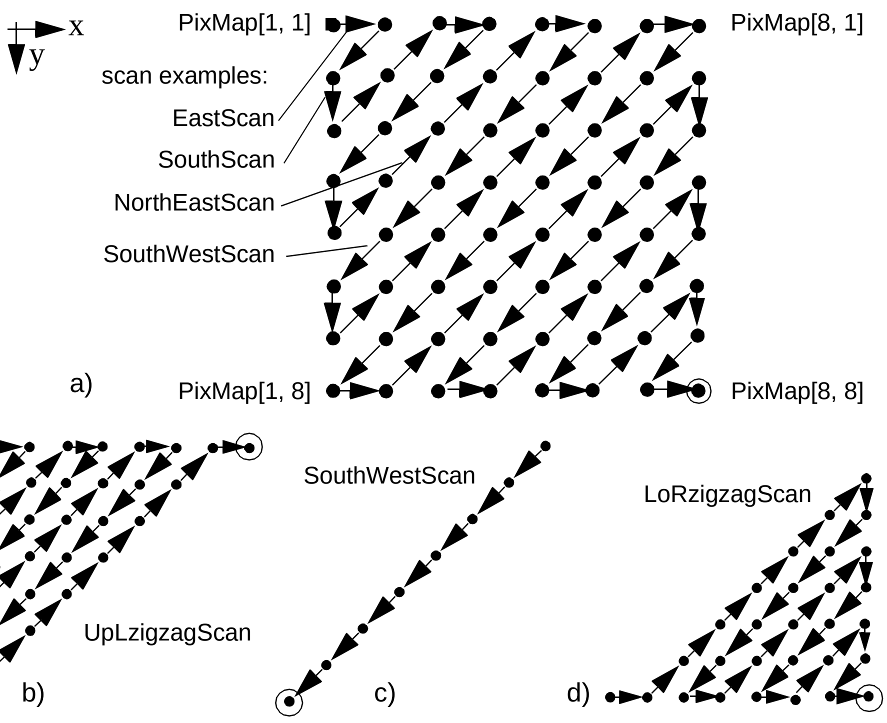


Fig. 3. JPEG Zig-Zag scan pattern for array PixMap [1:8,1:8], a) and its subpatterns: b) upper left triangle UpLzigzagScan, d) lower right LoRzigzagScan, c) full SouthWestScan

Compound Operators. An Xputer may execute expressions (which we call *compound operators*) within a single machine step, whereas computers can execute only a single operation at a time. The rALU may be configured in such a way, that one or more sets of parallel data paths form powerful compound operators connected to one or more scan windows (example in figure 7).

Execution triggering. A compound operator may be activated (sensitized) by setting a flag bit (and passivated by resetting this flag bit). Each operator currently being active is automatically executed whenever the scan windows connected to it are moved to a new location. E.g. during stepping through a scan pattern of length n this operator is executed n times.

Summary of Xputer Principles. The fundamental operational principles of Xputers are based on *data auto sequencing* mechanisms with only sparse control, so that Xputers are deterministically data-driven (in contrast to data flow machines, which are indeterministically data-driven by arbitration and thus are not debuggable). Xputer hardware supports some fine granularity parallelism (below instruction set level: at data path or gate level) in such a way that internal communication mechanisms are more simple than known from parallel computer systems (figure 9d and e, for more details about Xputer see [4], [5]).

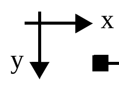
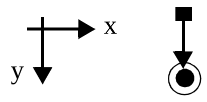
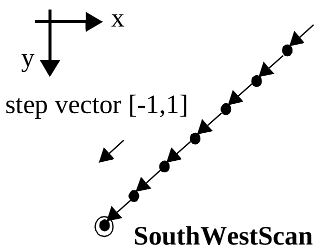
Source code example (see figure 5)	**Action described**
`ScanPattern` (see line (2)) `EastScan is 1 step [1,0]`	single step scan pattern **EastScan**:  step vector: [1,0]
`ScanPattern` (see line (3)) `SouthScan is 1 step [0,1]`	single step scan pattern **SouthScan**:  step vector: [0,1]
`ScanPattern` (see line (4)) `SouthWestScan is 7 steps [-1,1]`	multiple step scan pattern:  **SouthWestScan**
`ScanPattern` (see line (5)) `NorthEastScan is 7 steps [1,-1]`	like SouthWestScan (see above), but reversed order sequence

Fig. 4. Scan patterns declared for the JPEG example (see also figure 5)

3 A Programming Language for Xputers

This section introduces the high level Xputer programming language MoPL-3 (Map-oriented Programming Language) which is easy enough to learn, but which also is sufficiently powerful to explicitly exploit the hardware resources offered by the Xputer. For an earlier version of this language we have developed a compiler [15]. MoPL-3 is a C extension, including primitives for data sequencing and hardware reconfiguration.

3.1 MoPL-3: A Data-procedural Programming Language

This section introduces the essential parts of the language MoPL-3 and illustrates its semantics by means of two program text examples (see figure 5 and figure 8): the constant geometry FFT algorithm, and the data sequencing part for the JPEG zig-zag scan being part of a proposed picture data compression standard. MoPL-3 is an improved version of MoPL-2 having been implemented at Kaiserslautern as a syntax-directed editor [15].
From the von Neumann paradigm we are familiar with the concept of the *control state* (current location of control), where control statements at source program level are

translated into program counter manipulations at hardware machine level. The main extension issue in MoPL compared to other programming languages is the additional concept of *data location* or *data state* in such a way, that we now have simultaneously two different kinds of state sequences: a single sequential control state sequence and one (or more concurrent) data state sequence(s). The control flow notation does not model the underlying Xputer hardware very well, since it has been adopted from C to give priority to acceptance by programmers. The purpose of this extension is the easy programming of sequences of data addresses (scan patterns) to prepare code generation for the data sequencer. The familiar notation of these MoPL-3 constructs is easy to learn by the programmer.

Function Name	Corresponding Operation
rotl	turn left
rotr	turn right
rotu	turn 180^{o}
mirx	flip x
miry	flip y
reverse	reversed order sequence

Table 1. Transformation functions for scan patterns

3.2 Declarations and Statements

The following Xputer-specific items have to be predeclared: scan windows (by window declarations), rALU configurations (by rALUsubnet declarations), and scan patterns (by ScanPattern declarations). Later a rALU subnet (a compound operator) or a scan pattern may be called by their names having been assigned at declaration. Scan windows may be referenced within a rALU subnet declaration.

Scan Window Declarations. They have the form: window <group_name> is <window_specs>';'. Each window specification has the form: <window_name(s)> <window_size> handle <point>. Figure 6 shows an example, where a 2-dimensional window named 'SW1' with a size of 2 by 2 64-bit-words, and two windows named 'SW2' and 'SW3' with the size of a single 64-bit-word each, are declared. The <point> behind handle specifies the word location inside the window, which is referenced by scan patterns. The order of windows within a group refers to physical *window numbers* within the hardware platform. E.g. the above windows 'SW1' through 'SW3' are assigned to window numbers 1 through 3.

rALU Configuration. A compound operator is declared by a rALUsubnet declaration of the following form: rALUsubnet <group_name> is <expression_ assignment(s)>, where the compound operators are described by expressions. All operands referenced must be words within one or more scan windows. Figure 7 illustrates an example of a group 'FFT' which consists of two compound operators with destination window 'SW2', or 'SW3', respectively, and a common source window 'SW1'.

Scan Pattern Declarations. Scan patterns may be declared hierarchically *(nested scan patterns)*, where a higher level scan pattern may call lower level scan patterns by their names. Parallel scan patterns *(compound scan patterns)* may be declared, where several scan patterns are to be executed synchronously in parallel. Scan pattern declarations are relative to the current data state(s). A scan pattern declaration section has the form ScanPattern <declaration_item(s)> ';'. We distinguish two types of declaration items: simple scan pattern specifications <simple_spec> (linear scan patterns only: examples in figure 4) and procedural scan pattern specifications <proc_spec>. More details will be given within the explanation of the following two algorithm examples.

Activations. Predeclared rALU subnets (compound operators) may be activated by apply statements (example in line (44) of figure 10, where group 'FFT' is activated), passivated by passivate statements, and removed by remove statements (to save programmable interconnect space within the rALU). Scan window group definitions can be activated by adjust statements (example in line (43) of figure 10). Such adjustments are effective until another adjust statement is encountered.

Parallel Scan Patterns. For parallel execution (compound) scan patterns are called by a name list within a parbegin block. See example in line (46) of figure 10, where the scan patterns 'SP1', 'SP23' and 'SP23' are executed in parallel (which implies, that three different data states are manipulated in parallel). Pattern 'SP23' is listed twice to indicate, that two different scan windows are moved by scan patterns having the same specification. The order of patterns within the parbegin list corresponds to the order of windows within the adjustment currently effective (ThreeW, see line (43) in figure 10). E.g. scan pattern 'SP1' moves window no. 1, and 'SP2' moves windows no. 2 and 3. Each scan pattern starts at current data state, evokes a sequence of data state transitions. The data state after termination of a scan pattern remains unchanged, until it is modified by a moveto instruction or another scan pattern.

Nested Scan Patterns. Predeclared scan patterns may be called by their names. A scan pattern may call another scan pattern. Such nested calls have the following form: <pattern_name> '(' <pattern_definition>')' ';'. An example is shown in line (46) of figure 10, where scan pattern 'HLScan' calls the compound scan pattern definition formed by the parbegin block explained above. The entire scan operation is described as follows (for illustration see figure 9). Window group ThreeW is moved to starting points [0,0], [2,0], and [2,8] within array CGFFT by line (45) - see initial locations in figure 9c. Then the (inner loop) compound scan pattern (parbegin group in line (46)) is executed once. Then the (outer loop) scan pattern 'HLScan' executes a single step, where its step vectors move the window group ThreeW to new starting points. Now again the entire inner loop is executed. Finally the inner loop is executed from starting points being identical to the end points of the outer loop scan pattern. After last execution of the inner loop the windows have arrived at final locations shown in figure 9c.

3.3 JPEG ZIG-ZAG SCAN EXAMPLE

The MoPL-3 program in figure 5 illustrates programming the JPEG Zig-Zag scan pattern (named JPEGzigzagScan, see figure 3) being part of the JPEG data compression algorithm [10], [12], [14].þThe problem is to program a scan pattern for scanning 64 locations of the array PixMap declared in line (1) of figure 5 according to the sequence shown in figure 3a. Note the performance benefit from generating the 64 addresses needed by the hardwired address generator such, that no time consuming memory

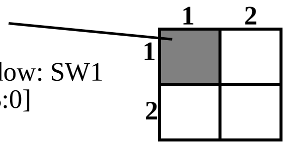
```
/* assuming, that rALU configuration has been declared and set-up */

Array       PixMap [1:8,1:8,15:0];                              (1)
ScanPattern EastScan      is  1 step  [ 1, 0],                  (2)
            SouthScan     is  1 step  [ 0, 1],                  (3)
            SouthWestScan is  7 steps [-1, 1],                  (4)
            NorthEastScan is  7 steps [ 1,-1],                  (5)
                                                                (6)
            UpLzigzagScan is                                    (7)
            begin                                               (8)
                 while (@[<8,])                                 (9)
                 begin  Eastscan;                              (10)
                      SouthWestScan until @[ð1,];              (11)
                      SouthScan;                               (12)
                      NorthEastScan until @[,ð1];              (13)
                 end                                           (14)
            end UpLzigzagScan;                                 (15)
                                                               (16)
                                                               (17)
            JPEGzigzagScan is                                  (18)
            begin                                              (19)
                 UpLzigzagScan;                                (20)
                 SouthWestScan;                                (21)
                 rotu (reverse(UpLzigzagScan));                (22)
            end JPEGzigzagScan;                                (23)
       /* end of declaration part*/                            (24)
                 •                                             (25)
                 •                                             (26)
begin  /*statement part*/                                      (27)
       moveto PixMap [1,1];                                    (28)
       JPEGzigzagScan;                                         (29)
end                                                            (30)
```

Fig. 5. MoPL program of the JPEG scan pattern shown in figure 3

access cycles are needed for address computation. Figure 3 shows, that the JPEG scan pattern may be partitioned into the three subsequences shown by figure 3b through d. Lines (2) thru (5) in figure 5 declare four scan patterns used later to synthesize the scan patterns shown in figure 3 and explained in figure 4. Scan pattern declaration statements have the following form:
<name_of_scan_pattern> <maximum_length_of_loop> STEPs <step_vector>.

Scan Pattern Declaration. The step vector specifies the next data location relative to the current data location (data state) before executing a step of the scan sequence. A positive integer specifies the maximum length (maximum step count) of the scan pat-

Source code example:	**Size of the scan windows:**

window ThreeW is
`SW1 [1:2,1:2,63:0]`
`handle [1,1],`
	handle [1,1]
	name of window: SW1
	word size [63:0]
	

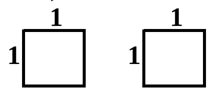
`SW2, SW3`
`[1:1,1:1,63:0]`
`handle [1,1];`
	names of windows: SW2, SW3
	word size [63:0]
	

Fig. 6. Scan window declaration of the FFT example (see also figure 8)

tern. A scan pattern may be terminated earlier than predeclared when an escape clause has become true (which will be explained later).

Calling Scan Patterns. Predeclared scan patterns may be called by statements. The 4 declared scan patterns (figure 4), which are needed for the JPEG zig-zag scan, are called in the two while loops at lines (9) thru (14) and at lines (18) thru (23) in figure 5, e.g. see line (10), where the scan pattern 'EastScan' is called (similar to a procedure call in C). By an escape a scan may also be terminated before <maximum_length_of_loop> is reached.

Escapes. In this case there will be an escape from the scan pattern, when the boundary of the data map is reached or exceeded. E.g. see the until clause (escape clause) in line (11) indicating an escape on having reached a leftmost word within the 'PixMap' array (see figure 3: the first execution of 'SouthWestScan' at top left corner of the array reaches only a loop length of 1). The condition @[ð1,] says: escape if within current array a data location with an x subscript ð1 has been reached. The empty position behind the comma says: ignore the y subscript.

Data State Initialization. Before the execution of the first scan pattern, you have to specify the starting point in the data map. For this purpose we use another data state manipulation statement, the moveto statement. With this statement (a data goto) you are able to realize absolute jumps of the scan window inside the data map. E.g. see line (28) in figure 5, where the scan window is moved to the upper left corner of the array 'PixMap', which is the starting point of scan pattern 'JPEGzigzagScan' call at line (29).

Hardware-supported Escapes. To avoid overhead for efficiency the until clauses are directly supported by the MoM hardware features of escape execution [4]. To support the until @ clauses by off-limits escape the address generator provides for each dimension (x, y) two comparators, an upper limit register, and a lower limit register.

Structured Scan Pattern. The above MoPL-3 program (figure 5) covers the following strategy. The first while loop at lines (9) thru (14) iterates the sequence of the 4 scan calls 'EastScan' thru NorthEastScan for the upper left triangle of the JPEG scan, fromþPixMap [1,1] to PixMap [8,1] (figure 3). The second while loop at lines (19) - (23) covers the lower right triangle from PixMap [8,1] to PixMap [8,8]. The 'South-

Source code example	`rALUsubnet FFT is`
	`SW2 = SW1 [1,1] + SW1 [2,1] * SW1 [1,2],`
	`SW3 = SW1 [1,1] + SW1 [2,1] * SW1 [1,2];`

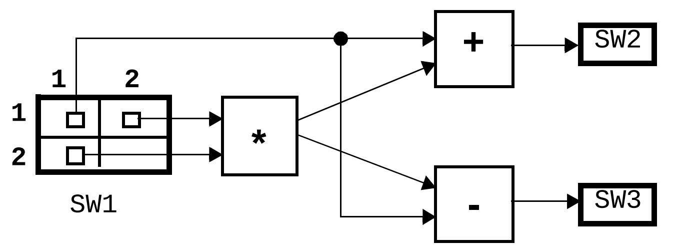
rALU configuration	

Fig. 7. Declaration of the 2 compound operators of the FFT example (see figure 8)

WestScan’þbetweenþboth while loops at line (30)þfromþPixMap [8,1] to PixMap [1,8] connects both triangular scans to obtain the total JPEG pattern. In line (22) two scan pattern transformation functions (rotu, reverse) are used. With these functions (see table 1) one can easily realize new scan patterns by using predeclared scan pattern, which structure is similar to the newer ones.

3.4 CONSTANT GEOMETRY FFT EXAMPLE

The second example illustrates parallelism by running several windows synchronously. It is the constant geometry Fast Fourier Transform (FFT) illustrated by figure 9, with a data map (CGFFT) size of 9 by 16 words (figure 9a). Figure 8 shows the MoPL-3 section declaring the scan patterns and the rALU configuration. The declaration of the scan pattern starts with the keyword ScanPattern. ’HLScan’ is the outer loop, whereas ’SP1’ and ’SP23’ are used for inner loops running in parallel (compound scan pattern, see Figure 9c). ’SP1’ is used for scan window ’SW1’ and ’SP23’ is used for two scan windows ’SW2’ and ’SW3’. The configuration of the rALU is specified in two parts: the window declaration (size declaration, see Figure 6) and the declaration of the rALUsubnet referencing the window group (see Figure 7).
The declaration of the scan windows and their sizes starts with the keyword window. With handle you can specify the reference point of a window (see Figure 6). This point will be needed, when you move a scan window to a specific place in the data memory. The window group named ’ThreeW’ (see above: including 3 scan windows with the names ’SW1’, ’SW2’ and ’SW3’) is declared by the keyword window followed by the name of the group and the keyword is etc. (see line (36) in Figure 8).
Problem-specific Compound Operators. Their declaration starts with rALUsubnet followed by the name of the rALU subnet group (here: ’FFT’) and the keyword is. Thereafter the compound operators are specified (see figure 7 and also lines (39) thru (41) in figure 8). Having been activated by adjust and apply (line (43) and (44) in figure 10) these operators are executed automatically in every single step of a scan pattern.

```
Array       CGFFT  [ 1:9, 1:16, 63:0];                      (31)
ScanPattern                                                 (32)
            SP1    is  7  steps  [0,2],                     (33)
            SP23   is  7  steps  [0,1],                     (34)
            HLScan is  3  steps  [2,0];                     (35)

window      ThreeW  is                                      (36)
            SW1       [1:2, 1:2, 63:0] handle [1,1],        (37)
            SW2, SW3 [1:1, 1:1, 63:0] handle [1,1];         (38)

rALUsubnet  FFT is                                          (39)
            SW2 = SW1[1,1] + SW1[2,1] * SW1[1,2],           (40)
            SW3 = SW1[1,1] + SW1[2,1] * SW1[1,2],           (41)
```

Fig. 8. Declaration part of the FFT example (operator definition omitted)

Execution of the Scan Pattern. Figure 9 shows an algorithm implementation example, a 16 point constant geometry FFT, where three scan windows run in parallel. Figure 9a shows the signal flow graph and the storage scheme (the grid in the background). The 16 input data points are stored in the leftmost column. Figure 8 shows the MoPL-3 program section, which moves the scan windows to proper starting points and calls the nested compound scan pattern (such as illustrated in figure 9c). Line (45) in figure 10 makes the handles of windows number one through three move to the 3 starting points of scan patterns (line (46)) within the array CGFFT:
Weights w are stored in every second column, where each second memory location is empty (for regularity reasons). Figure 7 shows the window adjustments: the 2-by-2 window no. 1 is the input window reading the operands a and b, and the weight w. Windows no. 2 and 3 are single-word result windows. Figure 7 also shows the compound operator and its interconnect to the three windows.
This is an example of fine granularity parallelism, as modelled by figure 9e, where several windows communicate with each other through a common rALU. Figure 9c illustrates the nested compound scan patterns for this example. Note, that with respect to performance this parallelism of scan windows makes sense only, if interleaving memory access is used, which is supported by the regularity of the storage scheme and the scan patterns.
This section has introduced the essentials of the language MoPL-3 by means of two algorithm implementation examples. The main objective of this section has been the illustration of the language elements for data sequencing programs and the illustration of its comprehensibility and the ease of its use.

4 CONCLUSIONS

The paper has briefly summarized the new Xputer machine paradigm, has demonstrated its basic execution mechanisms, and, has discussed its high efficiency having been published earlier. The paper has introduced a new high level Xputer programming language MoPL-3 and has illustrated its conciseness, comprehensibility and the

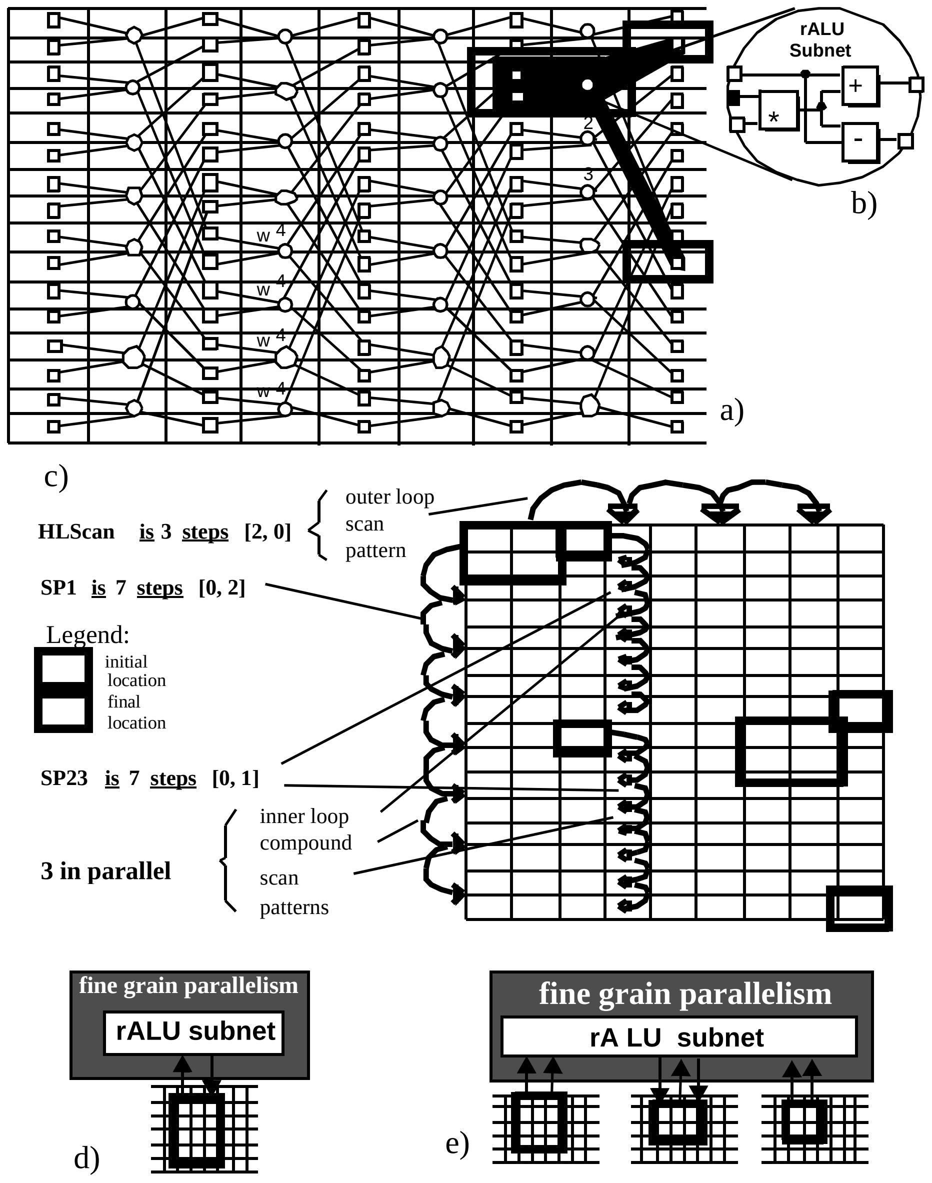


Fig. 9. Constant geometry FFT algorithm 16 point example using 3 scan windows synchronously in parallel: a) signal flow graph with data map grid and a scan window location snapshot example, b) rALU subnet, c) nested scan pattern illustration (also see figure 8), d) illustration of fine grain parallelism: single window use, e) multiple window use.

```
begin                                                  (42)
adjust ThreeW;                                         (43)
apply  FFT ;                                           (44)
moveto CGFTT [0,0], [2,0], [2,8] ;                     (45)
HLScan ( parbegin SP1, SP23, SP23 parend ) ;           (46)
end                                                    (47)
```

Fig. 10. Statement part of the FFT example

ease of its use in data-procedural programming for Xputers. An earlier version of the language (MoPL-2) has been implemented at Kaiserslautern on VAX station under ULTRIX. It is an essential new aspect of this new computational methodology, that it is the consequence of the impact of field-programmable logic and features from DSP and image processing on basic computational paradigms. Xputers, their languages and compilers open up several promising new directions in research and development - academic and industrial.

5 REFERENCES

1. A. Ast, R.W. Hartenstein, H. Reinig, K. Schmidt, M. Weber: A General Purpose Xputer Architecture Derived from DSP & Image Proceeing; in: (ed.: M.A. Bayoumi) VLSI Design Methodologies for DSP Architectures; Kluwer, 1994.
2. M. Christ: Texas Instruments TMS 320C25; Signalprozessoren 3; Oldenbourg, 1988.
3. R. Freeman: User-Programmable Gate Arrays; IEEE Spectrum, December 1988
4. R.W. Hartenstein, A.G. Hirschbiel, K. Schmidt, M. Weber: A Novel Paradigm of Parallel Computation and its Use to Implement Simple High Performance Hardware; Int'l Conf. on Information Technology, Tokyo, Japan, Oct. 1990.
5. R.W. Hartenstein, A.G. Hirschbiel, M. Weber, The Machine Paradigm of Xputers and its Application to Digital Signal Processing Acceleration; Proc. of 1990 International Conference on Parallel Processing, St. Charles, Oct. 1990.
6. R.W. Hartenstein, A.G. Hirschbiel, M. Riedmüller, K. Schmidt, M. Weber: A Novel ASIC Design Approach Based on a New Machine Paradigm; IEEE Journal of Solid-State Circuits, Vol. 26, No. 7, July 1991.
7. R.W. Hartenstein, G. Koch: The Universal Bus Considered Harmful; in [8].
8. R.W. Hartenstein, R. Zaks: Microarchitecture of Computer Systems, North Holland, 1975.
9. A.G. Hirschbiel: A Novel Processor Architecture Based on Auto Data Sequencing and Low Level Parallelism; Ph.D. Thesis, Kaiserslautern University, 1991.
10. J. Hoffmann: Redundanz raus; Computer Time, Heft 6, 1991.
11. G. J. Lipovski: A Stack Organization for Microcomputers; in [8].
12. L. Matterne et al.: A Flexible High-performance 2-D Discrete Cosine Transform IC; Proc. Int'l Symp on Circuits and Systems, Vol. 2, IEEE New York, 1989.
13. N.N.: DSP 56000/56001 Digital Signal Processor User's Manual; Motorola, '89.
14. G.K. Wallace: The JPEG Still Picture Compression Standard; CACM 34,4, April 1991.
15. M. Weber: An Application Development Method for Xputers; Ph.D. Thesis, Kaiserslautern University, 1990.