A Novel High-performance Machine Paradigm and ASIC Design Methodology

A. Ast, R. Hartenstein, H. Reinig, K. Schmidt, M. Weber
Fachbereich fuer Informatik, Universität Kaiserslautern
Postfach 3049, W-6750 Kaiserslautern, Germany
fax: (+49 631) 205-2640, *e-mail:* hartenst@rhrk.uni-KL.de

ABSTRACT .This paper illustrates an innovative compilation technique which is important for a novel class of computational devices called Xputers, which are by up to several orders of magnitude more efficient than the von Neumann paradigm of computers. Xputers are as flexible and as universal as computers. The flexibility of Xputers is achieved by using field-programmable logic (interconnect-reprogrammable media) as the essential technology platform (whereas the universality of computers stems from using the RAM). The paper first briefly illustrates the Xputer paradigm as a prerequisite needed to understand the fundamental issues of this new compilation technology.

1. INTRODUCTION

A growing need exists for digital signal processing (*DS processing* or *DSP*) in consumer applications [32] exemplified by compact disk, digital mobile radio, high-definition television, digital compact cassettes, and digital audio broadcasting. Such developments have been encouraged by synergy between VLSI and DSP techniques, and require VLSI circuits commercially feasible in high volumes, i. e. low hardware cost by small chip area. In consumer applications product life time has become very low, sometimes even only a few months. That's why short design time with minimum effort is a must for a product, and, to derive a product family from it. This, however, in ASIC or full-custom design conflicts with the goal of small chip area.

Programmable DS processors with large on-chip memory are a more flexible solution which permits sharing of hardware resources for merging of control functions and various processing functions to achieve low hardware cost by low chip count. The ease to modify program code with its short turn-around debugging supports fast development with low effort. Since in DSP speed is a constraint, rather than just an objective, the inefficiency of the von Neumann paradigm often is a severe problem [7]. Computers have the following bottlenecks and overhead phenomena:

International Design Automation Workshop ("Russian Workshop"), June 29 – 30, 1992, Moscow, Russia

- the ALU bottleneck
- accessing overhead
  - direct (address computation overhead)
  - indirect (data restructuring relocation overhead)
- control flow overhead
- processor-to-memory communication indigestion

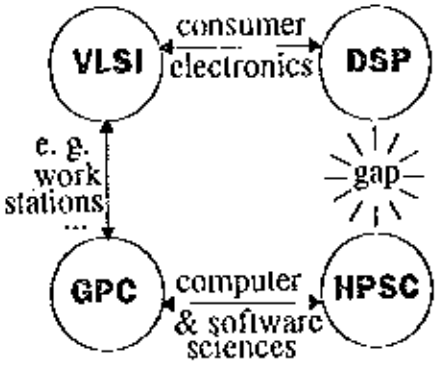
More details on these massive overhead phenomena have been explained and discussed elsewhere ([7],[8],[9]). This paper advocates a different approach, which is based on a more efficient non-von Neumann machine paradigm, which, however, permit to adopt the following high performance features from image processors and DS processors.

Image Processing. High performance of image processors is achieved by minimizing both, direct addressing overhead (address computation) and indirect addressing overhead (rearrangement of data blocks) through hardware-supported windowing, so that for each pixel operation the right data are immediately available at the right time at the right place (in the kernel). In sequential cellular logic machines the 'right time' is achieved by waiting loops (shift registers) for data (Fig. 2 a), such that no address is needed. RAM-based image processors reach this goal by multiple address generators (Fig. 2 b) running in parallel with other hardware and by redundant or interleaved memory: obviously the kernel organization directly points out an efficient storage scheme for redundant or interleaved memory use. Cellular architectures have influenced the development of the first Xputer architecture (MoM-1, formerly called PISA [11]).

DSP. Some digital signal processors provide hardware support to accelerate addressing. For example, the Texas Instruments TMS 320C25 [3] has an additional register arithmetic unit (ARAU) for address computation, supporting auto-increment, auto-decrement addressing, and bit-reversal addressing (useful for fast fourier transforms (FFT)). A *Repeatcounter* supports linear address sequences such, that for repetitive use of the same statement does not require repeated instruction fetch (to avoid control flow overhead).

The DS processor Motorola 56000 [23] uses an address generation unit (AGU) consisting of two address ALUs and several registers, to automatically generate addresses for two operands in parallel. Each ALU is capable to update an address registers in a single machine cycle. This update operation is performed by one of the following add operations: add, two's complement add, increment by one, decrement by one, reverse-carry add, and modulo add. The latter is a special addressing mode useful to built circular buffers, or for sequential addressing within multiple tables or arrays.

The addressing feature of a DS processor, in a wider sense, is comparable to that of vector processors, if we neglect the address dispatcher for interleaved memories in the latter. These address sequences can be used for linear addressing,



Abbreviations

HPSC	High Performance Scientific Computing
GPC	General Purpose Computing

Fig. 1: Relations between electrical engineering and computer science.

modulo addressing, and bit-reverse addressing. These address sequences are generated automatically without use of the main ALU, reducing addressing overhead.

A true general purpose DS processor does not yet exist. But the combination of a much more run-time-efficient machine paradigm with the above features is a promising approach, not only toward a general purpose special processor (general purpose DS processor), but toward a generally general purpose machine. The relatively close relations between xputers and supercomputing (which will be illustrated later) is an opportunity to bridge the gap between DSP and HPSC (see Fig. 1). That's why we would like to propose a new computational approach: the *Xputer* machine paradigm. The term xputer clearly distinguishes his *data-procedural* execution model from the *control-procedural* model of von Neumann computers.

This paper first briefly illustrates the new machine paradigm and its basic execution mechanisms and then illustrates programming and execution of some example algorithms on the MoM xputer architecture.

1.1. What is new?

The xputer paradigm with its new *data-procedural* basic execution mechanisms and all its impacts on technology platforms and basic architectural building blocks, and, on application support techniques like languages, compilers and programming techniques is a major step away from the familiar world of traditional computing based control-driven procedural von-Neumann-based models. The main differences are:

- the ALU of an Xputer is reconfigurable (soft) such, that it does not really have a fixed instruction set, nor a hardwired instruction format
- that's why (procedural) data sequencing is needed, since instruction sequencing is not feasible: a **data counter** is used instead of a program counter
- this leads to a fundamentally new machine paradigm and a new programming paradigm

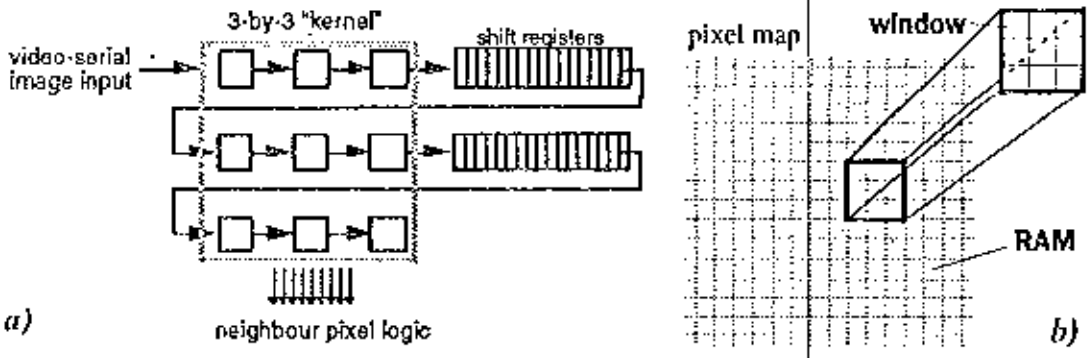


Fig. 2: 3-by-3 kernel: b) a window onto the pixel map, a) its video-serial implementation.

Solutions for all this lead to a novel interdisciplinary approach such, that a reader usually is not completely familiar with all the backgrounds needed by the reader. To achieve a more detailed comprehensibility of all fundamentals and relevant aspects, such as basic execution mechanisms, architectural elements, new programming paradigms, compilation techniques, as well as the feasibility of the high efficiency, a book of several hundred pages would be needed. Since only limited space is available for this paper, major parts of it are organized more as an illustration. The sales pitch which sometimes seems to be visible in the presentation is motivated by our desire to convince other researchers, that the principles of the experimental hardware and software illustrated here, point out a way to a highly promising new R&D area worth to invest major efforts in investigating all its relevant aspects.

2. INTRODUCING the XPUTER

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, also called *data sequencing paradigm*..

This section introduces and illustrates the basic machine principles [15]. Then the MoM-4 architecture is described, which later will be used as a vehicle to illustrate execution mechanisms via simple algorithm examples. Other examples will illustrate MoPL-3, a data-procedural programming language. This paper also tries to show the reasons of the good performance results having been obtained experimentally (e. g. see Fig. 17 a).

2.1. Xputer Machine Principles.

Fig. 3 b illustrates the basic xputer architecture principles. The key difference to computers is, that data sequencer and a reconfigurable ALU replace computers' program store, instruction sequencer and the hardwired ALU (this view is simplified). For operator selection instead of the sequencer another unit is used, which we call *residual control.*

Smart Register File. 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. Such an interface we call a *scan cache*. It implements a hardwired window to a number of adjacent locations in the memory space. Its size is adjustable at run time. Such a scan window may be 'placed' onto a particular location in memory under control a data sequencer. The scan cache is a generalization of the 3-by-3 or hexagonal kernel used by cellular or pseudo-cellular image processors (compare paragraph "Image processing" in section 1.). A sequence of locations we call a *scan path* or *scan pattern* (for examples see later).

The Data Sequencer. The hardwired data sequencer features a rich and flexible repertory of *scan patterns*, for moving scan caches along scan paths within memory space (e. g. see Fig. 3 c). Address sequences needed are generated by hardwired *address generators* having a powerful repertory of generic address sequences. For more details see later. 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.

Similar acceleration features are known from instruction sets with auto-increment features, from a digital signal processor with a bit reversal addressing feature [23] (see paragraphs "DSP" in section 1.), and from DMA controllers with very simple linear address sequences, mainly as needed for block transfers. The above control-procedural processor [23] even has an auto-increment feature for instruction iteration, where even instruction fetch iteration is suppressed (a pseudo-data-procedural mode, which avoids control flow overhead). For the xputer, however, the data sequencer is general purpose device covering the entire domain of generic scan paths, which directly maps the rich repertory of generic interconnect patterns (see [29] et al.) from space into time to obtain wide varieties of scan patterns, like e. g. video scan sequences, shuffle sequences, butterfly sequences, trellis sequences, data-driven sequences, even nested sequences, and many others. Instead of being a special feature it is an essential for xputers: the basis of the general purpose machine paradigm.

Reconfigurable ALU. Xputers (Fig. 3a) have a reconfigurable ALU (rALU), partly using the technology of field-programmable logic. Fig. 3a shows an example: the rALU of the MoM-4 Xputer architecture. The four smart register files called scan caches are explained later (lower left side in Fig. 3a). The MoM-

4 rALU has a repertory of hardwired operator subnets (see lower right side in Fig. 3a). Within the field-programmable part of the rALU additional operators needed for a particular application may be compiled by logic synthesis techniques (upper right in Fig. 3a) A global interconnect-programmable structure (centre in Fig. 3a) is the basis of connecting these operators to form one or more problem-specific compound operators, what will be illustrated later by a simple algorithm implementation example.

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 [10], especially in microprogram execution, where buses reach extremely high switching rates at run time. The intension of rALU use in xputers, however, is compound operator configuration at compile time (downloaded at loading time) as much as possible, so that path switching activities at are minimized and the underlying organizational overhead is pushed into compile time to save the much more precious run time.

Compound Operators. The rALU may be configured such a way, that one or more sets of parallel data paths form powerful compound operators which need only a single basic clock cycle to be executed. This rALU uses no fixed instruction set: compound operators are user-defined. Since their combinational machine code is loaded directly into the rALU, xputers do not have a program store nor an instruction sequencer. Instead a data sequencer is used which steps through the data memory to access the operands via register files called *data scan caches*. Xputers operate data-driven but unlike data flow machines, they feature deterministic principles of operation called *data sequencing*.

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* (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. (For more details see later.)

3. THE MoM XPUTER ARCHITECTURE

To use a practical and comprehensible example for illustration of execution mechanisms, programming techniques, and the novel task of compilers for xputers a simple algorithm example implementation on the MoM Xputer architecture will be used. This MoM (Map-oriented Machine) has a two-dimensional memory organization and uses some extra features which further support optimization efforts of the compiler.

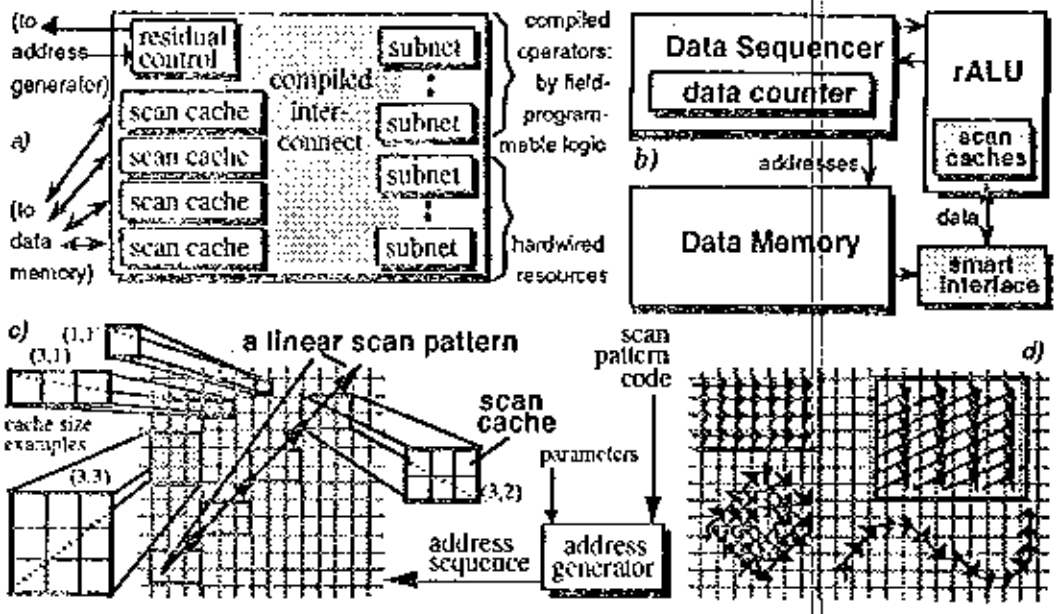
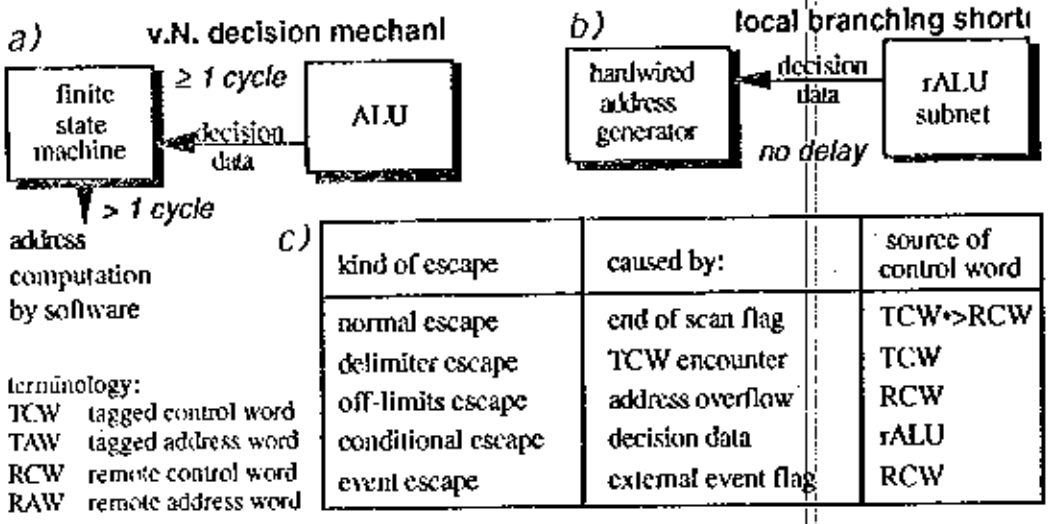


Fig. 3: Basic structures of Xputers and the MoM architecture: a) reconfigurable ALU (rALU) of the MoM, b) basic structure of Xputers, c) MoM cache size examples (left side) and a scan pattern example, d) a few other scan pattern examples.

Scan Cache. The MoM-4 has 4 scan caches (Fig. 3b and 2c) operating as two-dimensional windows, adjustable at run time (some size examples in Fig. 3c) up to a maximum size (5 by 5). Each cache can be used to read, write, or read and write data from and to the data memory. The MoM scan cache is a generalization of concepts having been used earlier [31], but never as an essential of a machine paradigm [9], [15] like the xputer. The MoM-4 scan cache is a smart register file featuring some hardwired smartness to save memory cycle time: 1) access mode flags [7] for each cache word (read-write, read-only, write-only, ignore), 2) shift paths being a generalization of the paths within the kernel shown in Fig. 2a (to avoid repeated access to the same words during smallest step cache movements [7]). The hardwired scan cache manager recognizes all occasions for using these features and activates them, so that no overhead is caused on the software side.

Parallel Data Sequences. Since the MoM-4 has *multiple scan caches*, several such data scan caches may be connected to the same compound operator, so that they may run in parallel (e. g. see Fig. 6c and line (33) thru (36) in Fig. 11) communicate with each other through the rALU (Fig. 6e). By redundant multiple memory (like in [15] and [17]) or interleaved memory use such parallel data sequencing provides a substantial throughput improvement.

Hardwired Data Address Generator. For the sequencer of the MoM a parameter-driven powerful hardwired address generator has been developed [8]. Examples of such data scan patterns are single steps as well as longer generic scan



kind of escape	caused by:	source of control word
normal escape	end of scan flag	TCW•>RCW
delimiter escape	TCW encounter	TCW
off-limits escape	address overflow	RCW
conditional escape	decision data	rALU
event escape	external event flag	RCW

Fig. 4: MoM Decision mechanisms: b) branching hardware more efficient than a) von Neumann branching, c) type of escapes from scan patterns

sequences, such as *video scan sequences*, *shuffle sequences*, *butterfly sequences*, *trellis sequences* and others (for a very few examples see Fig. 3d). The data scan patterns can be adjusted in parameter registers of the data sequencer or they can be evoked by the decision data feedback loop from the r-ALU (Fig. 4b). With this feedback loop data-dependent cache movements can be performed (e. g. lower left scan pattern in Fig. 3d).

Residual Control. Also *tagged control words* [7] having been inserted sparsely into the data memory map (Fig. 4c) can be recognized and decoded within the rALU to derive suitable decision data to select the next scan pattern. This kind of control we call *residual control*, because the decoder within the rALU is called only upon request and does not steal cycles from primary memory. By this means a sequence of several data scan patterns can be executed. Also residual control is a source of improved efficiency, what will be illustrated in section 3.1.

Efficient Branching Mechanisms. The MoM architecture provides branching mechanisms which are more efficient than those known from von Neumann architectures [7]. Von-Neumann-type branching requires one or more control accesses to primary memory, because only after the decision the next control state is known (Fig. 4). Depending on the kind of loop exit control code the number of memory accesses may be higher, even if no address computation is involved (which would cost further memory address cycles). For a number of cases such as e.g. nearest neighbor transitions in data memory space (in curve following, for example).

The MoM architecture provides more efficient branching modes, where usually no control action at all is needed, what saves memory cycles (Fig. 4b). For data-

dependent scan patterns this is achieved by direct manipulation of the least significant data address bits by decision data bits. Because this decision data bypasses the sequencer we call this mechanism a *local branching short-cut*.. Other branching modes (called *escape* modes) also avoiding control flow overhead, are handled within the address generator (see next paragraph). For instance, lines (48), (50), (52), (58), (60), and (62) in Fig. 15 (`while` and `until` clauses) refer to the 4 array limit parameter registers (within the address generator) specifying the 4 borders of the array `PixMap` (compare Fig. 13). Fig. 4c lists all types of escapes available. For more details see section 4.2.3.

3.1. Illustration of the MoM execution mechanism

The following algorithm execution example demonstrates the essentials of the Xputer execution mechanism and illustration the task of the new kind of compilers needed [31]: a kind of fine granularity scheduling of caches, rALU subnets, and of data words. Fig. 5a shows the algorithm in a textual high level language notation, Fig. 5b its graphical representation: a signal flow graph (SFG). Fig. 5d illustrates, how this algorithm is executed on the MoM Xputer architecture. The upper side of Fig. 5d shows the scan cache (format: 1 by 4 words), the rALU subnet for the compound operator (also compare Fig. 5a and Fig. 5b) and the interconnect between cache and subnet. The register inside the rALU subnet saves memory accesses, because the intermediate operands c(0) through c(7) do not need to move to memory. The bottom of Fig. 5d shows the data map (location of operands in memory.

The execution will run as follows. Starting at the left end of the data map The scan cache. The scan cache scans the data map area from left end (location shown in Fig. 5d) to the right end (shaded rectangle at the right of the data map in Fig. 5d). The sequence of arrows below the data map shows the scan pattern having.

Note, that no control action is needed because the *auto-execute* mode, where, whenever the scan cache is placed somewhere in the memory space (i. e. at each step of a scan), two things are carried out automatically (i.e. without having been called explicitly): the movement of data between scan cache and memory (*auto-xfer* mode) as well as the application of the marked rALU subnet to the variables held by the scan cache (what we call *auto-apply* mode). Note that by access mode tags only a minimum of memory semi cycles is carried out: read-only tags for all 4 words by this example. In our example 8 steps (x width=1, y with = 0) are carried out (Fig. 5d shows initial and final cache locations). From this example the task of the compiler may be summarized:

- define a data map (storage scheme)
- select a scan cache size and format
- define a compound operator and its scan cache interconnect
- select a suitable scan pattern available from address generators
- for linkage select a TCW and place it into the data map

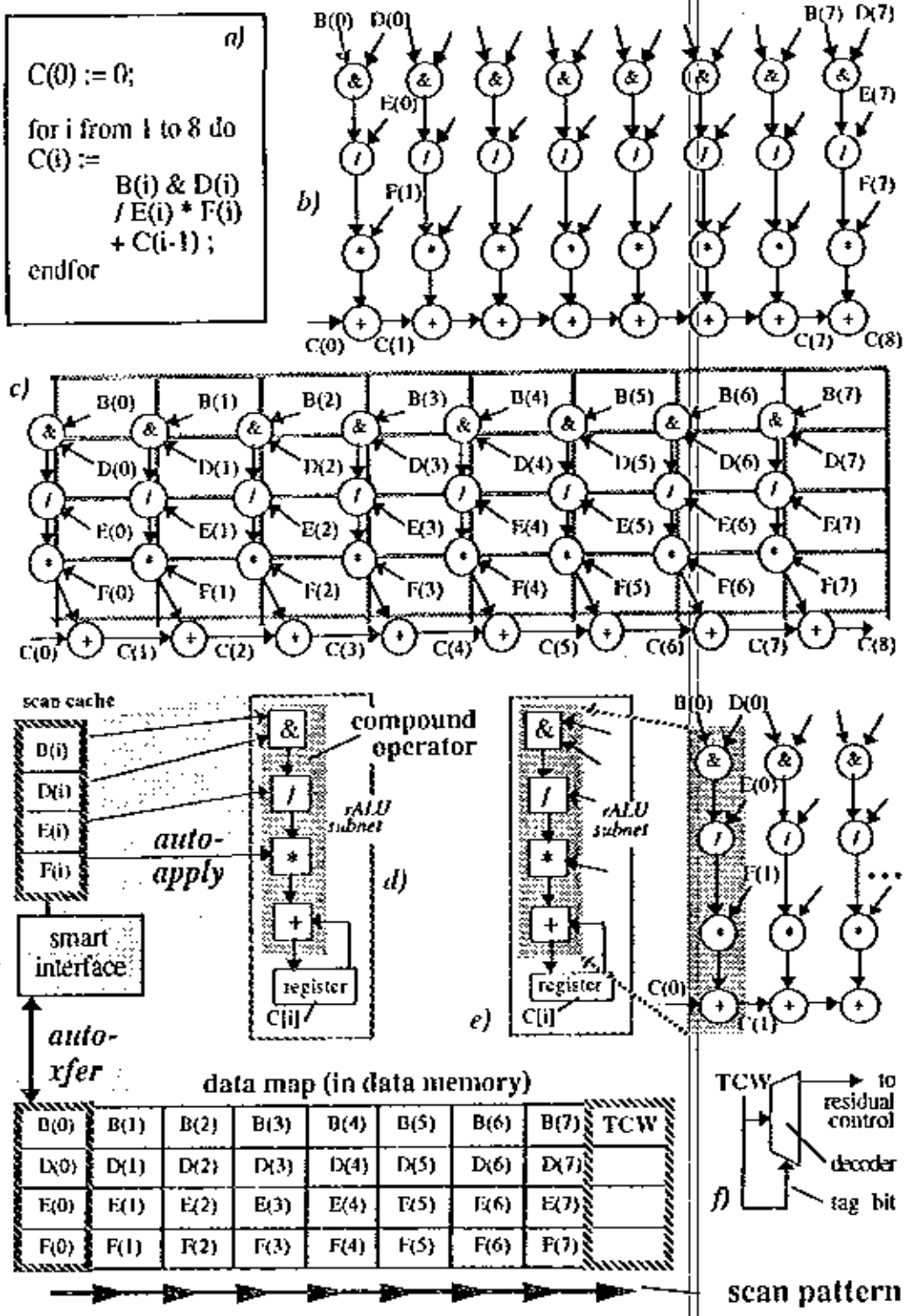


Fig. 5: Simple systolizable algorithm MoM execution example illustrating the compilation task: a) textual algorithm specification, b) graphic version of specification: signal flow graph (SFG), c) deriving a data map from SFG, e) deriving a compound operator for rALU from SFG, d) deriving scan cache size, rALU interconnect and scan pattern (also illustrating auto-apply and auto-xfer operation: needed for data-procedural machine principles)

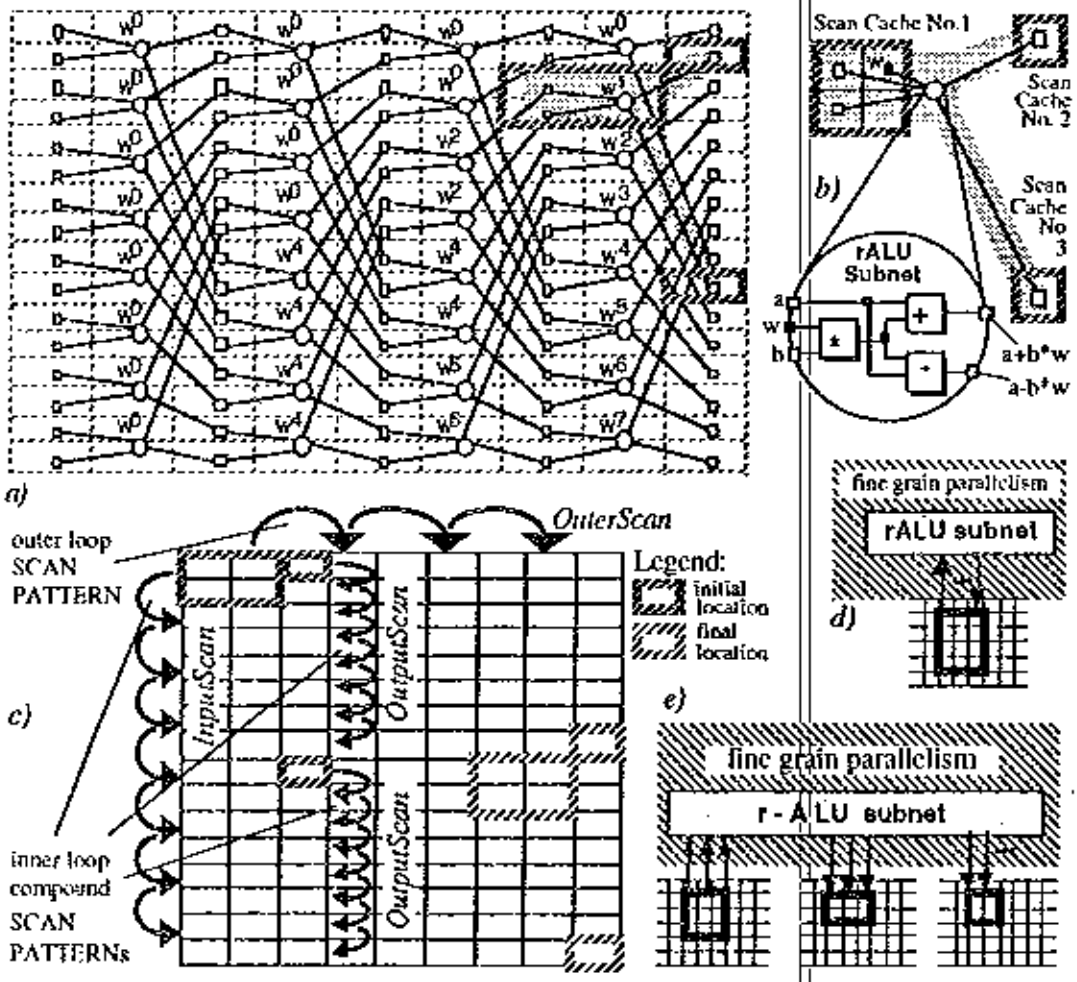


Fig. 6: Constant geometry FFT algorithm 16 point example using 3 scan caches synchronously in parallel: a) signal flow graph with data map grid and a scan cache location snapshot example, b) deriving rALU subnet, scan cache sizes and interconnect from compound operator, c) nested scan pattern illustration, d) illustration of fine grain parallelism: single cache use, e) multiple cache fine grain parallelism: field-programmable rALU as a communication mechanism.

At the end of the above data sequence example the cache finds a *tagged control word* (TCW) which then is decoded (right side of the map in Fig. 5d) to change the state of the *residual control logic* to select further actions of the Xputer. This sparse TCW insertion into data maps we call *sparse control*. Note that the control state changes occur only after many data operations (driven by the data sequencer).

3.2. A Multi-Cache Example

Fig. 6 shows an algorithm implementation example, a 16 point constant geometry FFT, where three scan caches run in parallel. Fig. 6 a shows the signal flow graph and the storage scheme (the grid in the background). The 16 input data points are

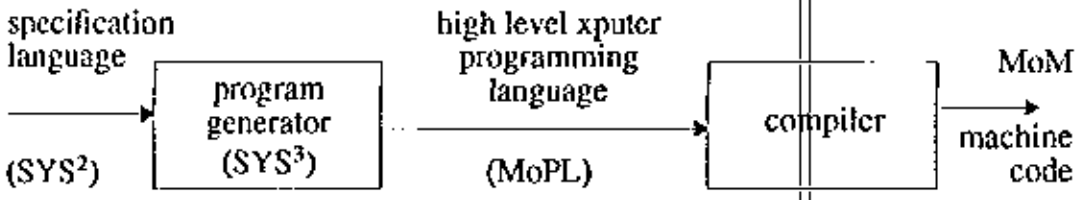


Fig. 7: MoM xputer language levels and translators

stored in the leftmost column. Weights w are stored in every second column, where each second memory location is empty (for regularity reasons). Fig. 6 b shows the cache adjustments: the 2-by-2 cache no. 1 is the input cache reading the operands a and b, and the weight w. Caches no. 2 and 3 are single-word result caches. Fig. 6 b also shows the compound operator and its interconnect to the three caches. This is an example of fine granularity parallelism, as modeled by Fig. 6 e, where several caches communicate with each other through a common rALU. Fig. 6 c illustrates the nested compound scan patterns for this example. Note, that with respect to performance this parallelism of scan caches makes sense only, if interleaving memory access is used, which is supported by the regularity of the storage scheme and the scan patterns.

4. A PROGRAMMING LANGUAGES FOR XPUTERS

This section introduces two languages (also see Fig. 7): a specification language SYS² and a 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 the xputer offers. For an earlier version of this language we have developed a compiler [31].

4.1. SYS²: Mapping systolic arrays onto xputers.

We have experimented with an approach using SYS² very high level specifications as a source input. With this approach we have examined some program generation techniques, which transform high level specifications into an equivalent high level xputer program (MoPL-1). Since an xputer scan cache provides neighborhood communication very efficiently [7], it is most promising to adapt techniques from the area of automatic synthesis of systolic arrays [20], briefly called *systolic synthesis* ([5], [19], [22] et al.); for an introduction to systolic arrays see [17], [24], [27], or others. Systolic Synthesis makes use of nearest neighbor communication within a VLSI processor array by projecting the data dependence graph of an algorithm into time and physical processor space. Systolic synthesis can handle only *systolic algorithms* or *systolizable algorithms* (i. e. algorithms which can be converted into systolic algorithms), which are algorithms with regular data dependencies. (For a survey on systolizable algorithms see [17].)

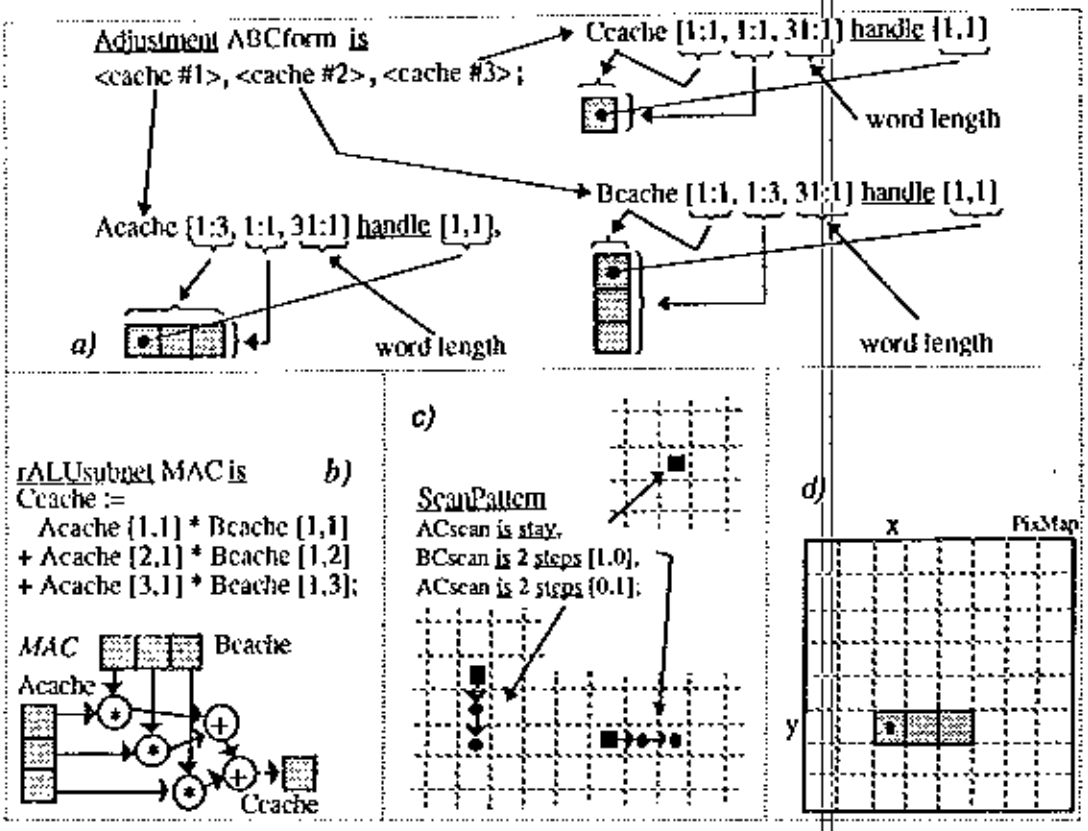


Fig. 8: Illustrating the declaration part of the MoPL program for the multiple-cache 3-by-3 matrix multiplication: a) scan cache format adjustments ABCforms, b) compound operator MAC, three scan patterns Ascan, BCscan, and ACscan.

Projection techniques from systolic synthesis have been adapted for parallelizing compilers for parallel computer systems. In the scene of parallel computing such techniques are called *systolizing compilation*, where the usual processes are modeled by the processing elements known from systolic synthesis, so that a concurrent implementation is derived. An xputer, however, is a monoprocessor. The problem therefore is to map the spatially distributed parallelism onto a data sequencing scheme suitable for xputers. For illustration let us use a 3 by 3 matrix multiplication example:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj} \qquad (1)$$

Several systolic synthesis systems, which have been implemented recently, usually accept nested loop notations as high level specifications ([5], [19], [22]). Such specifications look procedural, but the semantics is quite different: the intension is to express data dependencies, but not an order of execution (compare

```
Array           A, B, C [1:3, 1:3, 31:0];                              (5)
                                                                       (6)
Adjustment  ABCform  is                                                (7)
               Acache [1:3, 1:1, 31:1] handle [1,1],                   (8)
               Bcache [1:1, 1:3, 31:1] handle [1,1],                   (9)
               Ccache [1:1, 1:1, 31:1] handle [1,1];                  (10)
                                                                      (11)
rALUsubnet MAC is  Ccache = Acache [1,1] * Bcache [1,1]               (12)
                          + Acache [2,1] * Bcache [1,2]               (13)
                          + Acache [3,1] * Bcache [1,3];              (14)
                                                                      (15)
ScanPattern is  Ascan  stay,                                          (16)
                BCscan 2 steps [1,0],                                 (17)
                ACscan 2 steps [0,1];                                 (18)
```

Fig. 9: MoPL declaration part for matrix multiplication example in Fig. 8

Fig. 5 a and b). Expressed in SYS^2, the source language for the SYS^3 systolic synthesis system [19], the above matrix multiplication example is specified by the following source text:

```
for I := 1 to 3 do                                                    (1)
  for J := 1 to 3 do                                                  (2)
    for K := 1 to 3 do                                                (3)
      C[I,J] := C[I,J] + A[I,K] * B[K,J];                             (4)
```

The program generator having been implemented at Kaiserslautern generates a MoPL-1 program (MoPL-1 is an earlier version of MoPL-3). Next section describes a MoPL-3 program solution of this algorithm example.

4.2. 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 three program text examples: the above 3-by-3 matrix multiplication, the constant geometry FFT algorithm from Fig. 6, and the data sequencing part for the JPEG zigzag 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 [30].

The Language MoPL-3 is an extended dialect of the programming language C. The main extension issue is the *data location* or *data state* such, that we simultaneously have two different kinds of location or state. There is the familiar von-Neumann-type *control state (current location of control)*, which e. g. is handled by *goto* statements referencing control label locations within the program text, or, by other control statements. During execution of xputer programs such a *control state* is coexisting with one or more *data location states*,

```
begin                                                              (19)
      adjust ABCforms;                                             (20)
      apply  MAC;                                                  (21)
      moveto A[1,1], B[1,1], C[1,1];                               (22)
      fork                                                         (23)
        ACscan(Ascan), Ascan(BCscan), ACscan(BCscan);              (24)
      join                                                         (25)
end                                                                (26)
```

Fig. 10: MoPL statement part for matrix multiplication example in Fig. 8.

what will be illustrated subsequently. (The *control flow* notation does not model the underlying xputer hardware very well, since it has been adopted from C for compatibility reasons to minimize programmer training efforts.)

4.2.1. Matrix multiplication example

In addition to current control locations MoPL-3 programs also have *current data locations*, which are manipulated by `moveto` statements and scan patterns. Such a current data location is the current location of the scan cache. The statement `moveto A[1,1]`, for instance, says: move the cache to the location, where the variable `A[1,1]` is stored. A current data location does not change unless a data flow statement is encountered. I. e. after completion of a scan pattern the cache does not change its location, until another scan pattern or a `moveto` statement is encountered. In case of multiple scan cache use the MoPL program has multiple current data locations. For example the statement `moveto A[1,1], B[1,1], C[1,1]` says: move physical cache no. 1 to `A[1,1]`, cache no. 2 to `B[1,1]`, and, cache no. 3 to `C[1,1]`. The following two MoPL program examples illustrate the issue of current data location.

Lines (5) thru (18) in Fig. 9 show the declaration part of the matrix multiplication example. In line (5) the operand matrixes (arrays) `A` and `B`, and the result matrix `C` are declared. In line (7) thru (10) the size adjustments are declared for the physical scan caches number 1 thru 3 (also see Fig. 8 a). The handle point preceded by the keyword `handle` indicates the particular word location within the cache, which defines *current cache location* for address generator and user. See example in Fig. 8d, where the current data location is `PixMap[x,y]`. In line (12) thru (14) the compound operator named `MAC` is declared (see Fig. 8 b). In lines (16) thru (18) three scan patterns are declared which are named `Ascan`, `BCscan`, and `ACscan` (see Fig. 8 c). At declaration time scan patterns are not yet assigned to a physical scan cache, nor a starting point is defined.

Lines (19) thru (37) in Fig. 10 show the statement part of the MoPL matrix multiplication program. The `adjust` statement in line (20) assigns a predeclared format or format list (here: `ABCforms`) to physical scan caches. This adjustment remains effective until another adjustment statement is encountered. The `apply`

```
array                  CGFFT [1:9,1:16,31:0]                          (27)
ScanPattern            InputScan  is  7 steps   [0,2];                (28)
                       OutputScan is  7 steps   [0,1];                (29)
                       OuterScan  is  3 steps   [2,0];                (30)
                        :                                             (31)
                        :                                             (32)
                        :                                             (33)
     moveto CGFFT[1,1],[3,1],[3,9];                                   (34)
     OuterScan (  fork                                                (35)
                    InputScan,OutputScan,OutputScan;
                  join   )                                            (36)
                                                                      (37)
end
```

Fig. 11: MoPL program of constant geometry FFT scan from Fig. 6.

statement in line (21) activates the predeclared compound operator named MAC and keeps it effective until another apply statement is activated. The moveto statement in line (33) is the kind of *data goto*, which makes the caches no. 1 thru 3 jump into a the particular locations indicated within this statement (also see first paragraph of this section).

Line (35) shows calls to predeclared scan patterns named ACscan etc. (compare line (16) - (18)). These calls activate scanning actions starting from the current data locations. Note, that a call to a scan patterns is a call to a loop. The expression ACscan(Ascan) in line (35) indicates a call to a nested scan pattern, where the scan pattern Ascan is called by the scan pattern ACscan. This means to call a loop by a loop, i. e. to call nested loops. The fork / join brackets (23) (25) around these three scan calls show, that the three scan actions run in parallel synchronously. Due to MoPL semantics the sequence of scan calls within the fork list refers to the order of physical caches no. 1, 2, and 3.

Fig. 12 illustrates the hierarchy of nested scan patterns of our example algorithm (rows 1 thru 3: outer loop, rows 4 - 6: inner loop) and shows the snapshots (rows 7 thru 9) of the sequence of triple cache locations created by these nested scan patterns. Whenever by SP1(SP2) a scan pattern SP1 calls a scan pattern SP2 this means, that SP1 determines nothing else than the sequence of start locations of SP2. The list of scan names within the fork/join clause refers to the (by lines (7) thru (10)) predeclared numbering scheme of physical caches: ACscan(Ascan) is applied to scan cache no. 1, Ascan(BCscan) to scan cache no. 2, etc.

4.2.2. Constant geometry FFT example

Next MoPL text sample shows in lines (33) thru (36) the nested scans of the FFT algorithm example in Fig. 6c (scan pattern declarations in line (38) thru (30).

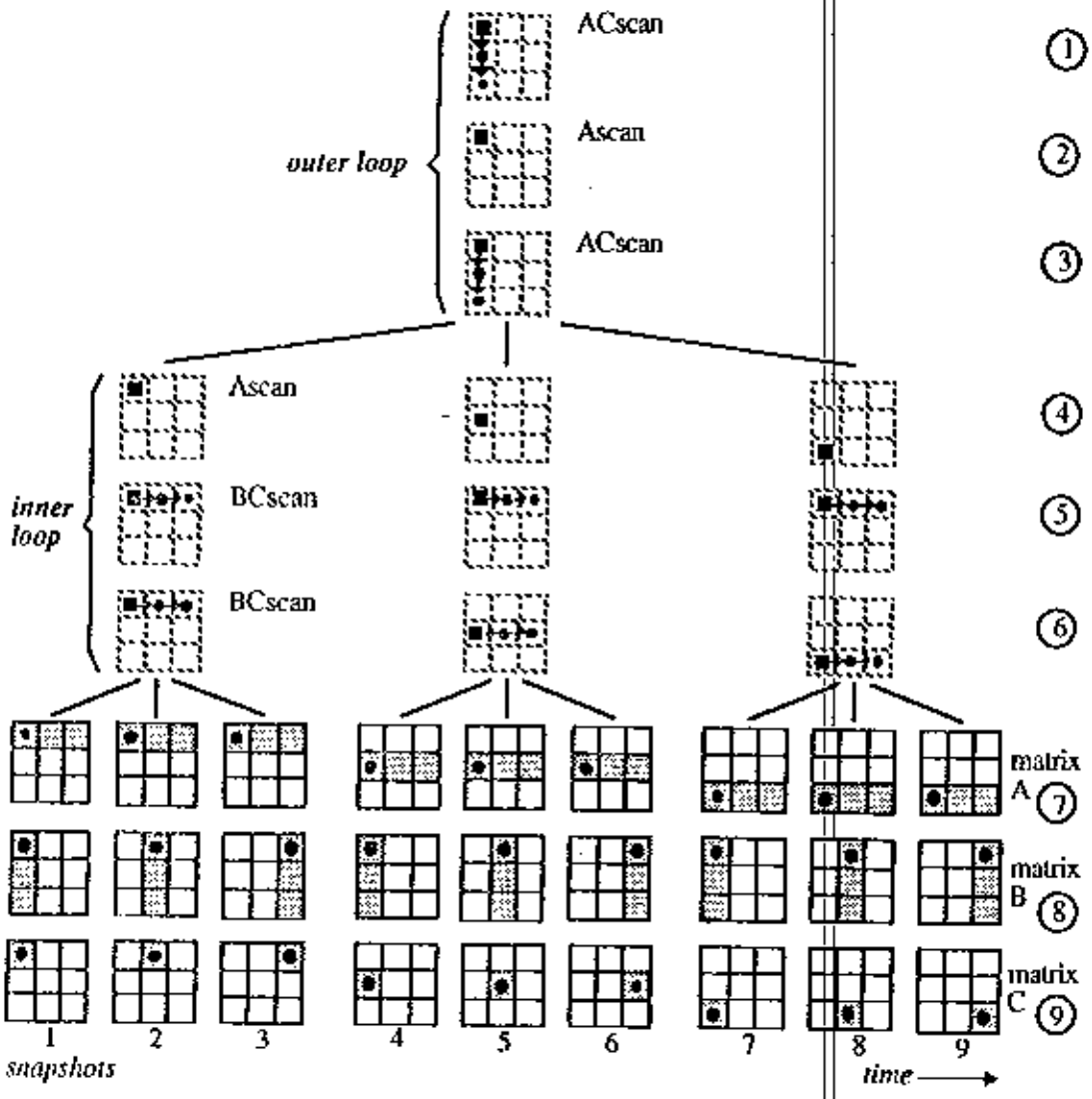


Fig. 12: Illustration of scan pattern execution for the 3 by 3 matrix multiplication example: the hierarchy of nested scan patterns (row 1 - 3: outer loop, row 4 - 6: inner loop, row 7 - 9: snapshot sequence of scan cache locations within matrixes).

4.2.3. JPEG zigzag scan example

The MoPL text from Fig. 15 illustrates programming the JPEG zigzag scan pattern (Fig. 13 [21]) named `JPEGzigzagScan` for scanning the array `PixMap` declared in line (40). This example uses a single 1-by-1 scan cache (adjusted as a single word buffer), which illustrates, that the performance benefit by the address generator can be obtained also for accessing long sequences of single memory locations. Lines (38) thru (30) declare four scan patterns (also see Fig. 13), where the statements have the form:

<name_of_scan_pattern> <maximum_length_of_loop> STEPs <step_vector>.

The step vector specifies the next data location relative to the current data location (before executing a step of the scan sequence). By an escape a scan may also be terminated before <maximum_length_of_loop> is reached. E. g. see the **until** clause in line (50) indicating an escape on reaching a leftmost word within the `PixMap` array (see Fig. 13: the first execution of `SouthWestScan` at top left corner of the array reaches only a loop length of 1). The condition `@[≤1, ]` says: escape if within current array a data location with an x subscript ≤1 is reached. The empty position behind the comma says: ignore the y subscript).

Hardware-supported Escapes. To avoid overhead for efficiency the **until** clauses are directly supported by MoM hardware features of escape execution [7] (also see Fig. 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

The above program covers the following strategy. The first **while** loop at lines (48) thru (53) iterates the sequence of the 4 scan calls `EastScan` thru `NorthEastScan` fortheupperleftriangle of theJPEGscan,from `PixMap[1,1]` to `PixMap[8,1]` (see Fig. 13). The second **while** loop at lines (58) - (63) covers the lower right triangle from `PixMap[8,1]` to `PixMap[8,8]`. The `SouthWestScan` between both **while** loops at line (69) from `PixMap[8,1]` to `PixMap[1,8]` connects both triangular scans to obtain the total JPEG pattern.

4.3. Geometric Transformation of Scan Patterns

Because of symmetry properties of many scan patterns it is useful to have the following transforms of the form *transformation (<scan_path>)* as part of the MoPL language:

name of transformation	transformation carried out
reverse	reverse scan direction and step sequence (starting point becomes final location, and vice versa, ...)
mirx	flip horizontally
miry	flip vertically
rotl	rotate left by 90°
rotr	rotate right by 90°
rotU	rotate by 180° ("U" reminds to "U-turn")

These transformations have been adopted from the KARL-3 hardware description language [12] [13]. The application is illustrated by the following declaration of the scan pattern `LoRzigzagScan` in Fig. 13d:

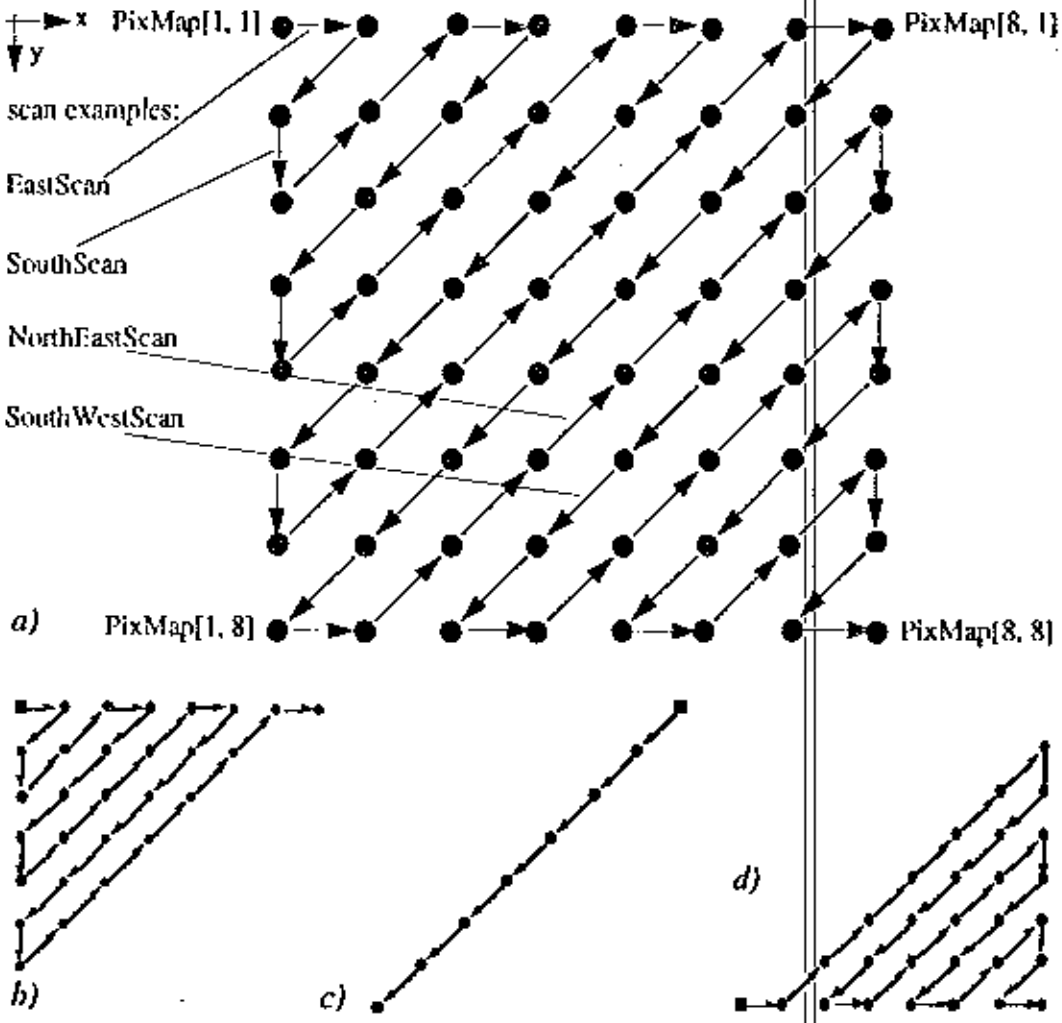


Fig. 13: JPEG zigzag scan pattern scanning an array PixMap [1:8,1:8] (a) and its subpatterns: b) upper left triangle UpLzigzagScan, d) lower right LoRzigzagScan, c) full SouthWestScan.

```
ScanPattern LoRzigzagScan is                                    (38)
                    reverse ( rotU (UpRzigzagScan));            (39)
```

which is equivalent to lines (56) thru (64) in Fig. 15, but derives this scan pattern by transforming the pattern `UpRzigzagScan`. Substantially reduction of the description length and program development time is obtained by reusing already existing declarations.

4.4. Section Summary

This section has introduced the essentials of the language MoPL-3, a C extension, by means of three 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.

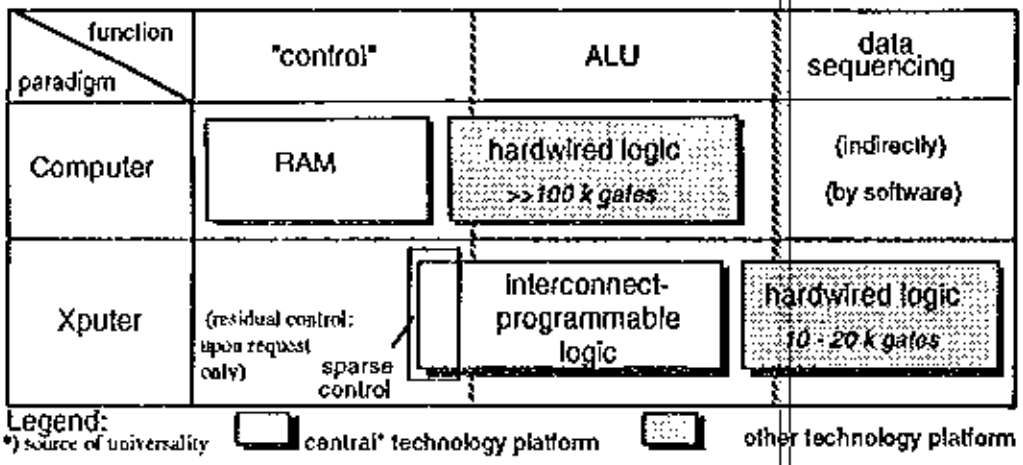


Fig. 14: The difference of the central technology platforms: computers and Xputers

5. A NEW TECHNOLOGY PLATFORM

For algorithms with regular data dependencies the Xputer paradigm is by several orders of magnitude more efficient than the von Neumann paradigm. Even for unstructured spaghetti-type sources the Xputer paradigm is at least half an order of magnitude more efficient. The high efficiency of Xputers has several reasons:

- Their data-procedural operational principles cope much better with most kinds of overhead and bottlenecks being typical to the von Neumann machine paradigm:
  - address computation overhead
  - control flow overhead,
  - ALU (multiplexing) bottleneck
  - processor-to-memory communication bottleneck
- Xputers support some compiled fine granularity parallelism inside their reconfigurable ALU (rALU).
- *A smart register file* with a *smart memory interface* contributes to further reduction of memory bandwidth requirements.
- Xputers are highly compiler-friendly by supporting more efficient optimizing compilation techniques, than possible for compilers for computers.

Commercial exploitation of Xputers is now becoming feasible by the progress and commercial availability of modern field-programmable technology [6]. Seen from a global point of view the universality of the Xputer paradigm is based on a new essential technology platform, which will be explained by next paragraphs.

RAM-based hardware universality. Von Neumann machine principles are based on *sequential code*, which is laid down in a RAM and which, at run time is scanned from there by an instruction sequencer. That is why the RAM is a central

```
Array           PixMap [1:8,1:8,15:0]                                  (40)
ScanPattern     EastScan      is  1 step  [ 1, 0];                     (41)
                SouthScan     is  1 step  [ 0, 1];                     (42)
                SouthWestScan is  7 steps [-1, 1];                     (43)
                NorthEastScan is  7 steps [ 1,-1];                     (44)
                                                                       (45)
                UpLzigzagScan     is                                   (46)
                begin                                                  (47)
                     while (@[<8,])                                    (48)
                     begin  Eastscan;                                  (49)
                            SouthWestScan until @[≤1,];                (50)
                            SouthScan;                                 (51)
                            NorthEastScan until @[,≤1];                (52)
                     end)                                              (53)
                end UpLzigzagScan ;                                    (54)
                                                                       (55)
                LoRzigzagScan     is                                   (56)
                begin                                                  (57)
                     while (! @[8,8])                                  (58)
                     begin  EastScan;                                  (59)
                            NorthEastScan until @[8,];                 (60)
                            Eastscan;                                  (61)
                            SouthWestScan until @[,8];                 (62)
                     end                                               (63)
                end LoRzigzagScan ;                                    (64)
                                                                       (65)
                JPEGzigzagScan    is                                   (66)
                begin                                                  (67)
                            UpLzigzagScan                              (68)
                            SouthWestScan;                             (69)
                            LoRzigzagScan                              (70)
                end JPEGzigzagScan ;                                   (71)
endScanPattern ;          (* end of declaration part*)                (72)
                     :                                                 (73)
                     :                                                 (74)
                     .
begin                     (* statement part*)                         (75)
     moveto PixMap [1,1];                                              (76)
     JPEGzigzagScan ;                                                  (77)
end                                                                    (78)
```

Fig. 15: MoPL program of the JPEG scan pattern shown in Fig. 13.

technology platform for computers already for several decades. Such RAM use is a source of the elegance and the universality of the von Neumann machine paradigm. Practically all of the user-specific problem complexity is pushed into the RAM. Although the processor is hardwired an extremely high flexibility is obtained. Due to the RAM-based executions mechanism the machine code needed is highly overhead-prone [15]. This is one of the reasons of the high complexity of von Neumann machine code and the high demand of RAM space.

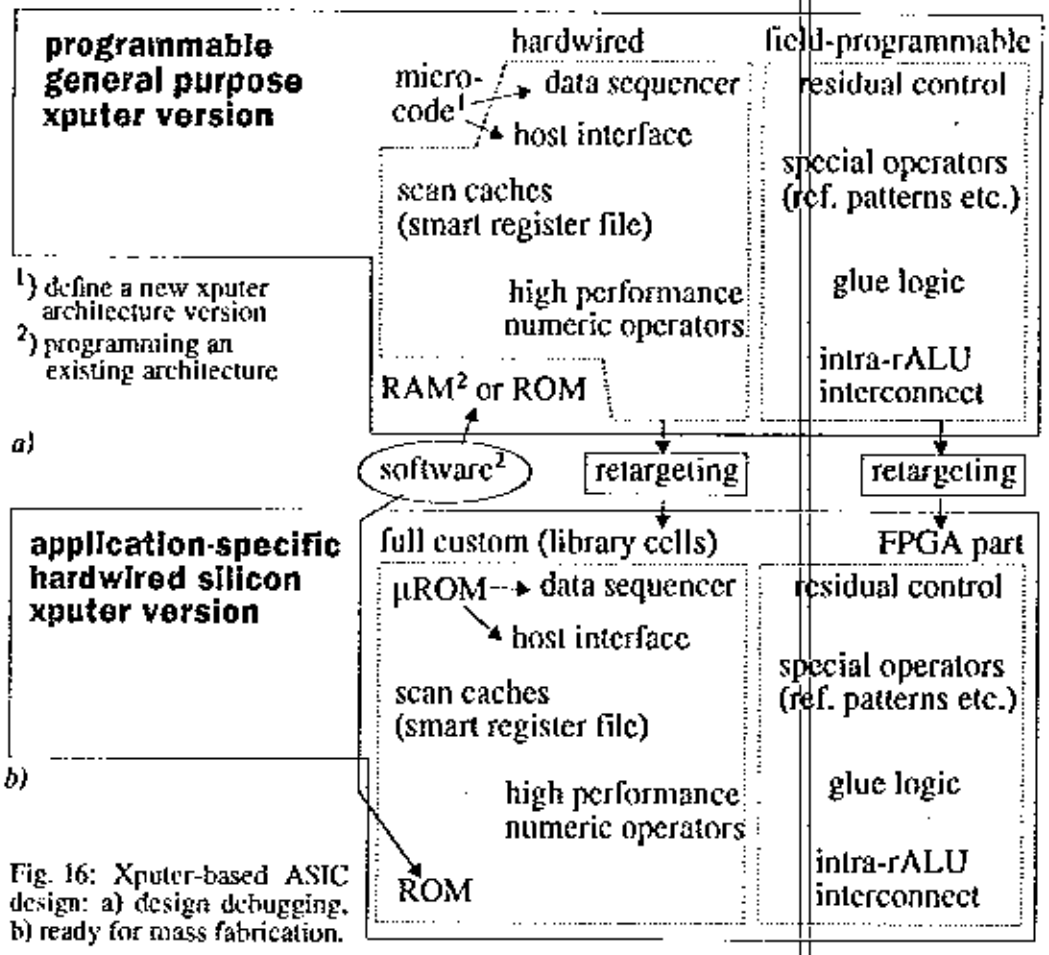


Fig. 16: Xputer-based ASIC design: a) design debugging, b) ready for mass fabrication.

General hardware universality by a new Technology Platform. Until recently field-programmable logic and related technologies have been used mainly for prototyping of relatively simple hardware. But now some researchers have recognized, that on the basis of such a technology platform completely new computational paradigms can be developed, which cannot be obtained from RAM-based technology platforms. An example is the PAM (Programmable Active Memory) concept by Vuillemin [2]. ASIC emulators (e. g. [4],[16]), also called hardware modelers (used for simulation acceleration), are examples being available as products already for a couple of years. But all those innovative uses of field-programmable logic do not provide a procedural machine paradigm, comparable to von Neumann processors.

Machine universality by programmable interconnect. Also to obtain universal *machine* paradigms, field-programmable logic is an alternative. Non-hardwired processors by using field-programmable technology platforms permit much more efficient machine paradigms. Field-programmable logic, or more precisely *interconnect-reprogrammable media* (irM), are programmable by effectively

non-sequential code, quickly alterable electrically, incrementally programmable. In this context we have experimented with several implementations of the Xputer machine paradigm [7], [8], [9], using an interconnect-reprogrammable ALU (rALU). This means flexibility (universality) by adaptable processor hardware instead of RAM use. The integration density and switching speed of such media are interesting already now for a number of applications. This is just the beginning: much more can be expected in the near future from this niche of the semiconductor market. For Xputers irM (instead of the RAM) is the central technology platform (see lower row in Fig. 14), source of simplicity, universality and elegance of hardware principles.

A new area of R&D in Programming Languages and Compilers. This new platform offers an implementation basis for a new class of programming languages and programming methods. Most of the von Neumann bottlenecks can be avoided by machine principles based on this alternative technology platform. Of course, also new compilation techniques are needed in such a fundamentally different target technology. For short term technology transfer reasons new cross compilation techniques are also needed to bridge the gap between both classes of computational paradigms.

Fast turn-around ASIC design. Recently field-programmable gate arrays have become available, which are compatible to particular real (mask-programmable) gate arrays. Due to code compatibility the personalization code of a field-programmable version can be easily translated into that of a real gate array (being faster and of higher integration density). Such conversions are carried out by *retargeting* software (e. g. [25]), which also provides an efficient bridge between computational paradigms and ASIC design. Compared to conventional ASIC this has the benefit, that simulation is replaced by execution being several orders of magnitude more efficient. Recently considerable attention has turned over to the topic of retargeting. This indicates that the high significance of retargeting for the future trends has been widely recognized.

6. APPLICATION AREAS FOR XPUTERS

Xputers like the MoM-3 and MoM-4 architectures are as universal as computers. A general competition between xputers and computers in all possible application areas would be unrealistic. This section briefly discusses suitable application areas for xputers from different points of view: for which algorithms and problem areas most benefits are to be expected - from which application environments least technology transfer problems will arise - in which application scenarios most cost/performance benefits can be expected.

General purpose DS and image processor. Xputers are not competitive to computers in general, since cross compilers, and application software environments are not available commercially. Competitiveness, however, is be

Algorithm	6800 16 MHz / millisec	MoM2 10 MHz / millisec	Acceleration factor
CMOS Design Rule Check	91330.20	39.0300	2340
Digital Filter	9126.40	29.4400	310
Lee Routing: seek S	42.50	0.0625	680
wavefront	70.00	0.3750	186
backtracking	23.25	0.1250	186

a)

Algorithm	Data Manipulation	Addressing	Control
CMOS Design Rule Check	7 %	93 %	<1 %
Digital Filter	28 %	58 %	14 %
Lee Routing: seek S	14 %	74 %	12 %
wavefront	6 %	92 %	6 %
backtracking	17 %	67 %	17 %

b)

Fig. 17: Performance Analysis: a) MoM-2 acceleration factors compared to Motorola 6800, b) overhead analysis on DEC VAX-11/750

expected for particular niches of application markets, such as image processing, digital signal processing, computer graphics, multi media applications, scientific computing, and others, where higher performance is needed at low hardware cost.

General purpose accelerator. From a technology transfer point o view, and, for utilization of existing utilities, interfaces and application software an good symbiosis would be using the xputer as a universal accelerator co-processor, hosted by a von Neumann computer, e. g. as an extension board within a workstation. Only those critical algorithms, which exceed the power of the host, are candidates for running on the co-processor, mostly only a few lines of source code.

New directions in supercomputing. Because of high acceleration factors in algorithms, which are subjects of supercomputing efforts, xputers, their compilers, and their applications are a source of ideas for new directions in supercomputing research - also in compilation techniques because of the paradigm's close relations to data dependency analysis. The xputer execution mechanism supported by scan caches and their address generators is a generalization of vectorization. With xputers the storage schemes for interleaved access memories are derived more easily and can be used for a wider variety of algorithms than with traditional supercomputers. (also see paragraph "Image Processing" in section I.).

6.1. Rapid turn-around ASIC synthesis.

Because the rALU and the use of field-programmable logic the xputer has close relations to ASIC design methods (also see last paragraph in section 6.). That's why with xputer parts held in cell libraries a new approach to ASIC design could be created. Debugging is by orders of magnitude faster than in traditional ASIC design, since execution is used instead of simulation. By retargeting this programmable version could be converted into a hardwired gate array version for fabrication.

Fig. 16 illustrates the xputer-based ASIC design process. Algorithm capture and optimization as well as debugging is carried out on a programmable version of the xputer platform (Fig. 16a). All "standard circuits" are hardwired, where address generators and external interfaces are microprogrammable (see box named "hardwired in Fig. 16a), and primary memory is a RAM. By microcode modification the architecture of the target machine may be optimized to a given application area. Machine code generated by the xputer's compiler is loaded into the RAM and into the field-programmable part (see Fig. 16a).

After debugging an application-specific hardwired silicon version of the machine is derived by a retargeting tool from the hardwired machine version and its machine code (Fig. 16b). The "standard circuits" (full-custom, if high performance is needed) are fetched from a cell library and their microcode is loaded from machine code of the programmable machine version. The FPGA part (see Fig. 16b) is derived from the field-programmable part of the programmable machine version (Fig. 16a) by retargeting. Let us summarize the advantages of such a design methodology over traditional ASIC design:

- fast turn-around debugging (simulation replaced by debugging),
- designer guidance by a simple machine paradigm as a model behind a high level programming language,
- very high source level: much less complex description input at (e. g. MoPL, or even SYS^2: much higher than e. g. VHDL).
- drastically reduced design effort: the major part of a design consists of general purpose cells from library
- high performance: due to the efficient paradigm high performance results are obtained, although procedural (sequential) methods are used.

Fig. 17 a gives some performance results having been obtained earlier [9]: acceleration factors, compared to von Neumann implementations of the same algorithms. Fig. 17b shows some overhead figures obtained experimentally, which partly explain the high acceleration factors.

7. CONCLUSIONS

The paper has briefly summarized the new xputer machine paradigm, has demonstrated its basic execution mechanisms, and, has shown its very high efficiency and the reasons for it. The paper has introduced a new high level xputer programming language MoPL-3 being an extension of the language C and has

illustrated its comprehensibility and the 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. For systolizable algorithms a program generator has been implemented as a front end, which generates MoPL programs by using modified versions of projection techniques known from systolic synthesis. 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. We have illustrated, that xputers, their languages and compilers open up several promising new directions in research and development - academic and industrial. We have shown, that also xputer-based ASIC design is a highly promising new direction of research and development.

8. REFERENCES

[1] A. Ast, R. W. Hartenstein, H. Reinig, K. Schmidt, M. Weber: ; in: (ed.: M. A. Bayoumi) VLSI Design Methodologies for DSP Architectures and Applications; Kluwer Academic Publishers, 1993

[2] P. Bertin, D. Roncin, J. Vuillemin: Introduction to Programmable Active Memories; Int'l Conf. on Systolic Arrays, Kilarney, Ireland, 1989

[3] M. Christ: Texas Instruments TMS 320C25; Signalprozessoren 3; Oldenbourg-Verlag 1988

[4] M. D'Amour, et al.: ASIC Emulation cuts Design Risk; High Performance Systems, Oct. 1989

[5] J. A. B. Fortes, K. S. Fu, B.J. Wah, "Systematic Approaches for Algorithmically Specified Systolic Arrays", in Computer Architecture: Concepts and Systems, (ed.: V. Milutinovic), North Holland, 1988.

[6] R. Freeman: User-Programmable Gate Arrays; IEEE Spectrum, Dec.1988.

[7] R. W. Hartenstein, A. G. Hirschbiel, K. Lemmert, 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.

[8] R. W. Hartenstein, A. G. Hirschbiel, K. Lemmert, K. Schmidt, M. Weber, "The Machine Paradigm of Xputers and its Application to Digital Signal Processing", Proc. of 1990 International Conference on Parallel Processing, St. Charles, Oct. 1990.

[9] R. Hartenstein, A. Hirschbiel, M. Weber: MoM - Map Oriented Machine; in: Ambler et al.: Hardware Accelerators, Adam Hilger, Bristol 1988.

[10] R. Hartenstein, G. Koch: The universal bus considered harmful; in: (eds.) R. Hartenstein, R. Zaks: Microarchitecture of Computer Systems, North Holland, 1975

[11] R.W. Hartenstein, R. Hauck, A.G. Hirschbiel, W. Nebel, M. Weber: PISA - A CAD package and special hardware for pixel oriented layout analysis, Proc. ICCAD 1984,

[12] R. W. Hartenstein, E. v.Puttkamer: KARL - a Hardware Description Language as a part of a CAD tool for VLSI; CHDL'79, Int'l Symp. on Computer Hardware Description Languages and their Applications, Palo Alto, California, USA, 1979; IEEE New York, 1979

[13] R. Hartenstein, K. Lemmert: KARL-III reference manual; CVT report, Universität Kaiserslautern; March 1984

[14] J. M. Herron, J. Farley, K. Preston Jr., H. Sellner: A General-Purpose High-Speed Logical Transform Image Processor; IEEE Transactions on Computers, C-31(8), 1982.

[15] A. G. Hirschbiel: A Novel Processor Architecture based on Auto Data Sequencing and Low Level Parallelism; Ph. D. dissertation, Universität Kaiserslautern, 1991

[16] P. A. Kaufmann: Wanted: Tools for Validation, Iteration; Computer Design, Dec.1989

[17] S.-Y. Kung, VLSI Array Processors; Prentice-Hall, 1988.

[18] E. A. Lee: Programmable DSPs: an Overview; IEEE Micro 10,5, Oct.'90

[19] K. Lemmert, SYS3 - a Systolic Synthesis System around KARL, Ph. D. dissertation, Kaiserslautern University, 1989.

[20] C. Lengauer, On the Projection Problem in Systolic Design; report, Carnegie-Mellon University, CMU-CS-88-102, Pittsburgh, 1988

[21] 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

[22] D. I. Moldovan, "ADVIS: A Software Package for the Design of Systolic Arrays", IEEE Transactions on Computer Aided Design, pp. 33-40, Jan., 1987.

[23] N. N. (Motorola): DSP 56000 / 56001 Digital Signal Processor User's Manual; Motorola Corp., 1989.

[24] N. Petkov: Systolische Algorithmen und Arrays, Akademischer Verlag 1989.

[25] N. N. (Plessey): Quickgate (Product Overview); Plessey Semiconductors, Swindon, U.K., May, 1990.

[26] K. Preston Jr.: The CELLSCAN system - A leucocyte pattern analyzer, Proc. Western Joint Comput. Conf., 1961

[27] P. Quinton, Y. Robert: Systolic Algorithms & Architectures, Prentice Hall 1989.

[28] H. J. Siegel: Interconnection networks for large-scale parallel processing; McGraw-Hill, New York, 1990

[29] Thomas W. Starnes: MC68000: Philosophie und praktische Realisierung einer 16/32-Bit_Mikroprozessorfamilie; Das 68000-Sonderheft, Franzis-Verlag, München 1985.

[30] M. Weber: An Application Development Method for Xputers; Ph. D. dissertation, Kaiserslautern University, 1990.

[31] R. Woudsma, J. L. van Meerbergen: Consumer Applications: A Driving Force for High-Level Synthesis of Signal Processing Applications; IEEE Micro 11,4, Aug. 1992