Joachim Lenzer, Renate Lenzer, Ralf Steinmetz:
[JL84] *Design, Programming and Application of a High-Speed Multi-Processor System for Use in Digital Signal Processing*. In: *Signal Processing, North Holland Publishing Compony, Amsterdam*, vol. 7, no. 7/84, p. 17--30, July 1984.

Signal Processing 7 (1984) 17–30
North-Holland

DESIGN, PROGRAMMING AND APPLICATION OF A HIGH-SPEED MULTI-PROCESSOR SYSTEM FOR USE IN DIGITAL SIGNAL PROCESSING

Joachim LENZER
TE KA DE—Philips Kommunikations Industrie AG, Thurn- und- Taxis- Strasse 10, 8500 Nürnberg 10, FRG

Renate LENZER
Fachgebiet für Graphisch Interaktive Systeme, Fachbereich Informatik, TH Darmstadt, Alexanderstrasse 24, 6100 Darmstadt, FRG

Ralf STEINMETZ
Institut für Übertragungstechnik, Fachbereich Elektrische Nachrichtentechnik, TH Darmstadt, Merckstrasse 25, 6100 Darmstadt, FRG

Received 20 July 1982
Revised 8 February 1984

Abstract. This article introduces a programming and transformation system for describing, optimizing and mapping parallel algorithms onto a highly parallel multiprocessor architecture. The application of the system in digital signal processing, especially digital filtering, has been the main subject of investigation. Consequently both the class of computations schemes as well as the computer organization (MIMD) and the interconnection structure (crossbar) are effected. After presenting a special MIMD-architecture the results concerning concurrent optimized versus pure sequential computing time (speed-up) are delineated.

Zusammenfassung. Dieser Artikel beschreibt eine Methode zur Formulierung paralleler Algorithmen, deren Optimierung und Abbildung auf Multiprozessorsysteme. Der Schwerpunkt liegt dabei auf speziellen Anwendungen aus dem Gebiet der Echtzeitsignalverarbeitung, insbesondere der digitalen Filterung. Dies wirkt sich sowohl auf die Klasse der erfaßbaren Algorithmen als auch der betrachteten Rechnerarchitekturen und der notwendigen Verbindungsstrukturen aus. Dieser Ansatz wird an Hand eines vollständig implementierten Programmier- und Transformationssystems erläutert. Es wird eine spezielle MIMD-Architektur vorgestellt und daran Analysen des Zeitverhaltens gegenüber rein sequentieller Verarbeitung angestellt.

Résumé. Cet article introduit un système de programmation et de transformation à l'aide duquel on peut décrire, améliorer, accelérer et appliquer des algorithmes parallèles dans une architecture d'un système 'multiprocessor'. Le point principal des analyses est l'application en temps réel, spécialement dans le domaine du filtrage numérique. Par consequent on observe une influence sur les classes des algorithmes ainsi que sur l'architecture de l'ordinateur. Cette proposition est décrite à l'aide d'un système complét et achevé de programmation et de transformation. Une MIMD-architecture particulière est présentée. Ensuite, on fait l'étude des temps de l'architecture du MIMD et des méthodes uniquement séquentielles.

Keywords. Multi-processor, parallel algorithms, deterministic scheduling, digital filtering, tree transformations.

1. Introduction

Computer speed is limited by two basic factors. The first one concerns hardware technology: how fast can elementary operations be executed; how long are processor and bus cycle times and memory access time? The second one covers both the aspects of computer architecture and program organization, which form the subject of this paper.

The demand for high computation speed and data throughput in real-time processing requires concepts, dissenting the von-Neumann principle [2, 6, 8, 13], in which a single CPU operates serially on data items that it fetches from and restores to memory via the bottleneck "bus".

Fortunately the advances in VLSI technology have resulted in low cost computing capabilities, which make feasible large-scale systems, including a lot of processing elements PE's, large memory modules and complex interconnection networks.

The design of a parallel computation scheme

However, parallel processing as the most powerful architectural concept to reduce computation time [3] raises more sophisticated questions to a designer than a sequential approach does.

Sequential approach

S1: Design of the sequential algorithm
S2: Translation into machine language
S3: Tuning of the algorithm by realizing time consuming functions by firmware or hardware

Parallel approach

P1.1: Design of the parallel algorithm
P1.2: Design of the corresponding parallel architecture
P2: Scheduling of the basic algorithmic components and determining the number of necessary processing elements
P3: Translation into machine language

Ad P1.1

Each parallel algorithm (PA) consists of a set of basic components called "Task-Set" T. Between each pair $t(i)$, $t(j)$ of elements of T a precedence relation $PR(i,j)$ has to be defined.

$\mathrm{PR}(i,j) = "\langle"$: $t(i)$ has to be finished before $t(j)$ will be started

$\mathrm{PR}(i,j) = "\rangle"$: $t(j)$ has to be finished before $t(i)$ will be started

$\mathrm{PR}(i,j) = "\langle\rangle"$: $t(i)$ and $t(j)$ can be executed simultaneously

So the design of a PA is equivalent to the definition of the tasks $t(i)$, their data interfaces to other tasks, their semantical behaviour as well as the set of precedence relations.

Ad P1.2

Close to the PA the machine architecture has to be defined. The tasks correspond to the PE's, which have to be connected by some form of interconnection network [1] in order to exchange data or messages. The number of data paths between the PE's and as a consequence the organization of the interconnection network itself depends on the PA, which has to be computed.

Ad P2

After the definition of both the computation scheme and the machine architecture the scheduling of the tasks to the PE's has to be done. The problem is to find an optimal schedule, i.e. a schedule with

minimal computation time and minimal number of PE's and additional hardware resources. However, most of the optimal scheduling algorithms are NP-complete, even for very strong simplifications of both the computation scheme and the machine architecture [5].

Statement of the problem

It is evident that solving all the problems, which have been mentioned so far depends on the question: Which kind of computation schemes do we want to analyse? This question requires investigations on program analysis, including data and control abstraction mechanisms in program specification. Since this is not the intended subject of this publication, the result will be given without detailed justification: all computation schemes which can be scheduled deterministically [5, 14]. A formal definition as well as implications of this assumption will be given later on. For this class of computation schemes the above mentioned design steps (P1–P3) will be investigated in detail in this article leading to a programming and transformation system for a special multi-processor architecture to be used in digital signal processing, especially digital filtering.

2. Description of parallel algorithms

Most of the problems in digital signal processing—as digital filtering, DFT, eigenvalue problems, vector operations etc.—can be expressed by sequences of arithmetic expressions [4] and simple assignments. Constructions for describing the data dependent control flow of programs—as IF-THEN-ELSE, WHILE, REPEAT, FOR, CASE etc—used in high level von-Neumann styled languages are not allowed due to the assumption of deterministic scheduling. The reason for this will become clearer in Section 4.

Abstraction mechanisms for composition of data—ARRAY, RECORD, POINTER, ENUMERATION TYPES, etc—and consequently sophisticated access paths to data items are avoided due to a more simple language implementation.

Input–output relation

Each program maps input objects I to output objects O by means of a function T (Fig. 1):

$$O = T(I),$$

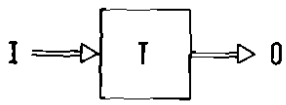

Fig. 1. $O = T(I)$.

with $O = (o_1, o_2, \ldots o_n)$ and $I = (i_1, i_2, \ldots i_m)$. Two rules can be applied, combining existing functions to form new ones:

$$O = T(I_1, I_2) \tag{1}$$

$$O = T_2(T_1(I)), \tag{2}$$

(resp. Fig. 2, Fig. 3). To define a functional form in a linear notation the Deterministic Computing Scheme Language DCSL [15] has been defined and implemented, by which the syntactic interface of a function

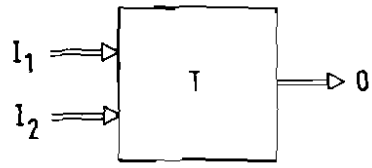

Fig. 2. $O = T(I_1, I_2)$.

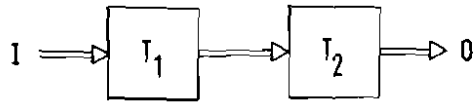

Fig. 3. $O = T_2(T_1(I))$.

T is written as:

```
FUNCTION T
INPUT i1, i2, ... im
OUTPUT o1, o2, ... on
..... Definition of the relation
END T;
```

History sensitivity

Up to now, however, it is not possible to define input–output relations, which are history sensitive as illustrated in Fig. 4. This means that an output vector O depends on both an input vector I and a state vector S, which describes the history of computation.

$$O = F(I, S), \qquad S = G(I, S).$$

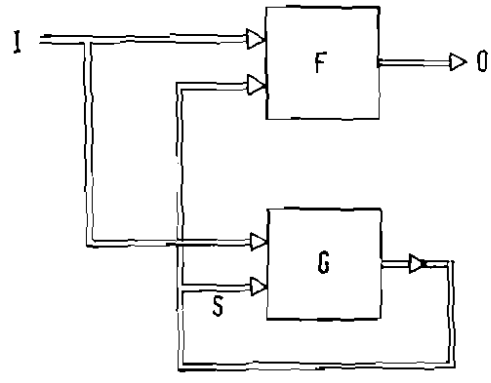

Fig. 4. $O = F(I, S)$ $S = G(I, S)$.

In DCSL this is written

```
MODULE M
INPUT i1, i2, ..., im
OUTPUT o1, o2, ..., on
STATE s1, s2, ..., st = G(I, S)
O - F(I, S)
END M;
```

The functions F and G are of the form defined above. Each computation scheme which can be described by these rules will be called a deterministic computation scheme DCS.

Arithmetic expressions

The functional behaviour is defined by arithmetic expressions. An arithmetic expression is any well formed string composed by operators $(+, -, *, /)$, left and right parenthesis and data objects, which are constants, input- or state-variables.

To include more powerful instructions of the processor system for the description of input–output relations, the user is allowed to define machine dependent operators with the same syntactic interface as functions. A typical example is a butterfly operator, which could be realized beyond the architectural level by hardware or firmware. Thus the semantical meaning of such a hardware operator, i.e. the input–output relation, is not relevant at this point of view.

The following example and Fig. 5 demonstrate some features of DCSL. A detailed description of DCSL is given in [15].

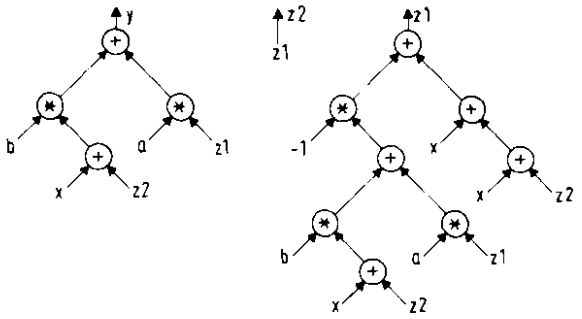

Fig. 5. Tree oriented computation scheme for the notch-filter.

```
MODULE notch_filter
INPUT x
OUTPUT y
CONST a = 0.1,      b = 0.2;
STATE z1, z2 = -1*(b*(x+z2)+a*z1)+(x+(x+z2)), z1
           y = b*(x+z2)+a*z1
END notch_filter;
```

Representation of a DCS

A DCS will be represented by a Directed Acyclic Graph (DAG) [14].

Definition. A directed acyclic graph (DAG) is a 2-tuple DAG $= (N, E)$, where N (nodes) and E (edges) are disjunct finite sets with $E \to N \times N$. Iff for $e \in E$ $e = (n1, n2)$; $n1, n2 \in N$ then e is an edge from $n1$ to $n2$. We say that $n1$ is a predecessor of $n2$, and $n2$ is a successor of $n1$. There exist no subset of nodes $(n1, n2, \ldots, nm)$, where each pair $(ni, ni+1)$, i in $(1, \ldots, m-1)$, is connected by an edge and $n1 = nm$.

Definition. The number of edges directed into node n is called the inner degree of n and is denoted by

in(n). The number of edges directed away from node n is called the outer degree of n and is denoted by out(n). All nodes n with in(n) $\neq 0$ are called inner nodes, those with in(n) $= 0$ outer nodes.

Definition. Any DAG is a DCS, where the inner nodes of the graph represent arithmetic operations, whereas the outer nodes mean data objects, i.e. constants, input objects or state variables.

The inner nodes (tasks) of a DAG correspond to processing elements (PE's) of the real executing machine, whereas the edges can be interpreted as data links for passing data objects from the output of one PE to the input of another one.

The representation of DCSL-expressions are trees.

Definition. A DAG T is a tree, iff

1) exactly one node $r \in N$ exists with out(r) $= 0$, called root
2) forall $n \in N - \{r\} \Rightarrow$ out(n) $= 1$

3. Parallelization and optimization of algorithms

In this section we will illustrate the question of how to get the most parallel computation scheme from any algorithmic specification described above.

The assumed executive machine model hereby implies an unrestricted number of PE's available for evaluation, each of those being able to perform any task in maybe different time periods. Data can be passed at any time between input and output data links of the PE's via a complete interconnection network. In this section, however, the time required for passing data between the processing elements is ignored. Those more realistic assumptions of the computer architecture will be added in Section 4, when the problem of scheduling will be introduced.

Tree height reduction

The height of an expression tree, i.e. the maximum path length from the root to a leaf, determines the number of computation steps which are necessary for evaluation, whereas the width of the tree determines the number of PE's required for fully parallel computation. Any expression tree, however, which is generated during parsing a DCSL-expression is not the only possible representation. For example the expression $(a+b+c+d)$ is algebraic equivalent to the expression $((a+b)+(c+d))$, although the two parse trees (Fig. 6) are different.

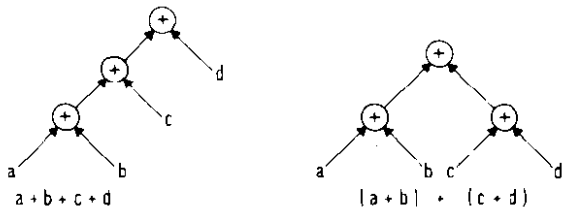

Fig. 6. Different parse trees for two algebraic equivalent expressions.

Often algebraic laws—commutativity, associativity, distributivity—applicable to certain arithmetic operators, can reduce the height of an expression tree. For computing the height H of a tree T, it is first necessary to introduce the cost C of a tree node. This means, that evaluating a node t by one of the PE's takes $C(t)$ elementary time units.

Definition. The height H of a tree T with root t is defined as

$$H(T)=\begin{cases}0 & \text{if } \text{in}(t)=0\\ \max(H(t(i)))+C(t) & 1\leq i\leq \text{in}(t)\end{cases} \qquad t(i) \text{ means the } i\text{-th subtree of } t$$

Definition. Given a set L of transformation rules, we say that two expression trees T_1 and T_2 are equivalent under L, if there exists a sequence of transformations derived from L, which will transform T_1 into T_2.

We will not discuss the problem, that those transformations are not equivalent with regard to numerical stability, which may lead to rounding errors [4]. Thus, applying the rules to a given expression tree T_1 an equivalent tree T_2 with minimal height can be found.

Each transformation rule consists of one source tree template and one or more target templates in the sense of [12]. So the tree transformation can be explained as a process of laying all the source templates over the actual point of the program tree and selecting the 'best fitting' one.

However, out of the set of possible target templates only this template with lowest height will be selected. This is a necessary condition to avoid transformation cycles. The realization of the recognition and transformation process of tree templates is described in [12].

Tree width reduction

When performing a transformation, the height of the tree decreases, whereas the width increases. Consequently more PE's have to be available in order to execute this expression in parallel. If the PE's are functionally identical, i.e. if each PE is able to evaluate each tree node, the total number is equivalent to the width W. The determination of the upper bound of necessary PE's for parallel evaluation is quite similar to the register allocation problem within code generation for high level languages.

The form of optimization, which reduces the width of a tree and as a consequence the number of processors necessary for the evaluation of a given set of arithmetic expressions, is the recognition and elimination of common subexpressions. Two subexpressions S and T are common, if the corresponding expression trees fulfil the following condition:

```
FUNCTION common (S, T:tree) RETURNS boolean;
BEGIN
  IF root (S) = root (T) THEN
  IF S and T are leafs THEN RETURN true
  ELSE IF root (S) in '+', '*'
  /* commutative operators */
    THEN RETURN common (S(1), T(1)) AND common (S(2), T(2))
                OR common (S(2), T(1)) AND common (S(1), T(2))
  /* non-commutative operators */
    ELSE RETURN common (S(1), T(1)) AND common (S(2), T(2))
  ELSE RETURN false
END common;
```

If two members of a given set of trees or subtrees are common, they need to be evaluated only once. By application of these optimizations a tree is transformed into a DAG. In Fig. 7 the completely optimized graph for the notch filter is illustrated.

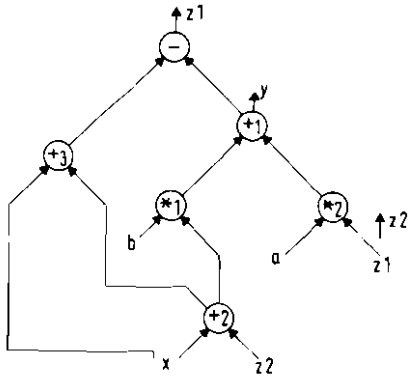

Fig. 7. Optimized computation scheme for the notch-filter.

4. Scheduling and code generation

Introduction to the scheduling problem

One rather general definition of scheduling is

– the allocation of processors and resources over time to perform a collection of tasks.

An individual task is characterized by its processor type, its resource requirements and its duration. In addition a collection of tasks may be described by the algorithmic constraints (precedence restrictions) that exist among its elements. One possible representation is the taskgraph, which is equivalent to a DAG of Section 2. At this point of view, however, the relation $E \to N \times N$ is named precedence relation (PR) or predecessor relation and defines for each task $t(i)$ $(1 \leqslant i \leqslant n)$ the earliest execution data.

The solution to a scheduling problem is any feasible resolution of three types of constraints:

– sequencing order of the tasks
– allocation of processors and
– allocation of resources.

Under these assumptions we can define a scheduling system S as a triple $S = (\mathrm{DAG}, P, R)$, where DAG is a directed acyclic graph in the sense of Section 2, with the possibility of nonuniform duration times $td(i)$ $(1 \leqslant i \leqslant n)$ for the tasks $t(i)(1 \leqslant i \leqslant n)$. P is a set of not necessarily homogeneous processor types $p(i)$ $(1 \leqslant i \leqslant m)$. R is a set of resources $r(i)(1 \leqslant i \leqslant s)$. The term resource is specified as the amount of data links between the PE's and the state variables.

The system S will generate a schedule s, which will be depicted using one of the most widely accepted graphical representation models: the Gantt-chart. In our case the chart shows the processor-task allocation over time, with the specified PE's along the vertical axis and the task dispatching together with the time scale along the horizontal axis.

Restrictions in real-time signal-processing

Algorithms for scheduling tasks for a multi-processor system are well known [5]. However, optimal solutions exist only for a small subset of problems. Fortunately in the field of digital signal processing it

is possible to make some assumptions, e.g. nonpreemptive tasks which simplify the complexity of these problems [9].

There exists two different approaches to find a schedule for a given problem

- the heuristic solution and
- the optimal solution.

One advantage of the heuristic solution is the fact of polynomial time-complexity, for the optimal solution suffers from exponential time-complexity. The optimal strategy allows only some special problem specifications to be solved whereas the heuristic approach guarantees always a solution. The terms 'optimal' and 'heuristic' imply the disadvantage of the heuristic approach: the completion time of the generated schedule.

Although the field of optimal solutions is quite interesting the heuristic approach is most widely used for its polynomial behaviour.

The generation of a schedule based on heuristic algorithms consists of two phases [9]:

1) The determination of parallel working task sets PTS(i) ($1 \leq i \leq n$) according to the precedence relationship PR: All tasks of a PTS(i) can be executed in parallel. Between the diffcrent PTS's there exist dependencies in a way that all tasks, which are element of PTS(i) have to be finished before starting a task, which is element of PTS(j) with $i < j$. Out of the PTS's the number of necessary processors per processor type can easily be determined as the maximum number of tasks of a certain type over all PTS's.

2) The computation of a priority number for every task $t(i)$ ($1 \leq i \leq n$) according to the applied heuristic algorithm: The computation of the priority number for the tasks differs from one heuristic algorithm to thc other. They are determined either by

- the task duration,
- the processor-type constraints,
- the resource constraints or
- the precedence structure.

This often leads to quite different schedule completion times, though the applied algorithm have to be chosen carefully, depending on the input structure (e.g. tree, anti-tree, etc.) [9].

A task sequence then is generated by attaching the task with the highest priority out of the set of the selected PTS to the next free processor. A random selection will be done if more than one possible task is having the highest priority number.

To complete the example from Fig. 7 the result of the scheduling system for the notch filter is represented in Fig. 8. The td's for the different operations are: $td(*) = 3$, $td(+) = 1$, $td(-) = 1$.

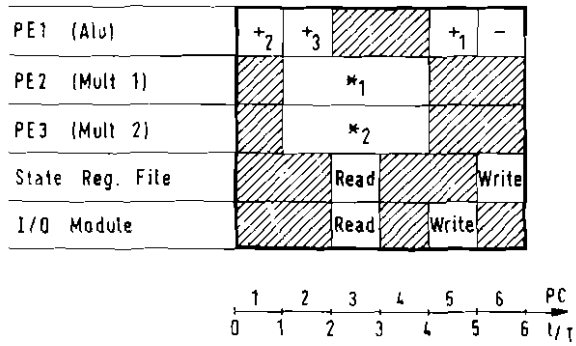

Fig. 8. Gantt-chart for the notch-filter.

Structure of the design and programming system

The various phases of the transformation process and consequently the organization of the transformation system is shown in Fig. 9. Starting at the point of DCSL-program definition, after the conventional lexical, syntactic and semantic analysis a program tree is built up, which is transformed into a tree with minimal height, using tree transformation rules. The so transformed tree is optimized by recognition and elimination of common subexpressions, leading to a DAG. The DAG is converted into precedence relations, which are the input for the scheduling system.

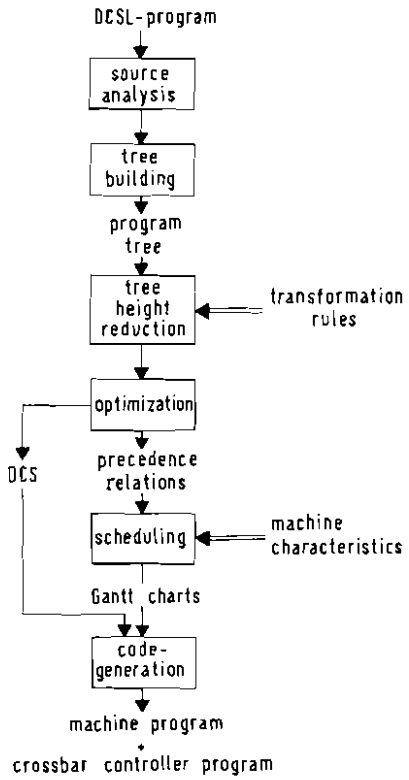

Fig. 9. Structure of the programming and transformation system.

Code generation

The generated Gantt-charts as well as the DAG are used as input for the final code generation phase, in which the programs for each PE and the informations for the interconnection network are generated.

Each PE consists of two register files, one for both input operands, an arithmetic unit to perform the operations and a local control store containing the instructions. Each instruction for a PE looks like

⟨PE_op_code⟩⟨source 1⟩⟨source 2⟩⟨reg 1⟩⟨reg 2⟩,

where ⟨PE_op_code⟩ determines the arithmetic operation including the states 'busy' and 'idle'. ⟨source 1⟩ and ⟨source 2⟩ represent the input operands for the operation, which can be either the contents of a register (*R*(*address*)) *or a constant operand* (*C*(value)) located in the control store.

The interconnection network realizes the data links between the outputs of the PE's, their inputs, I/O-processor, and a common register file, containing the state variables. The network is controlled by a local controller and a local control store, which contains the connection data for each time period in the following form:

(input → output) . . . (input → output)

Each input and output link is described by an integer number. In Fig. 10 the machine programs for the notch-filter are shown, corresponding to the Gantt-chart of Fig. 8.

5. The multi-processor-architecture

Classification of architectures

In this chapter a multiprocessor architecture for use in digital filtering is introduced (Fig. 11). The PE's are not identical, because they usually perform different operations as addition, subtraction, multiplication and I/O-operations. Therefore the multiprocessor system can be classified as inhomogeneously. Different operations generally last different time periods to be executed. For example a multiplication takes more time as an addition, if we use today's conventional hardward components like a Shottky-TTL adder (50–80 ns) and a 16-bit hardware multiplier (150–200 ns).

Processing elements (PE)

The system includes n PE's of different types, which execute the elementary operations as multiplication, addition and subtraction. Each processor contains one output and two input data links, which are completely connected by a crossbar switch matrix.

Input data are buffered in random accessible register files for the case that intermediate results can not be processed immediately. So to each PE a local control store is attached, containing the program instructions (operation, input operands) as well as the addresses of registers in which data has to be stored for further requirements. Data input and output is handled by an I/O-processor.

Synchronous communication

The elementary operations executed by each processor take an integer multiple of the basic time cycle. Therefore data exchanging between the processors has to be performed in fixed time intervals. So the interconnection network can operate synchronously.

Centralized controlling

After each operation cycle the computed intermediate results as outputs of the processors (k) have to be passed to the inputs (m) of the processors. So the communication links of the interconnection network have to be rearranged for each connecting phase. This rearrangement must be performed by a centralized network controller.

Network topology

Each processor output (k) must have the chance of linking to each of the processor (m) inputs the network must be able to handle all possible connections $k \times m$. The generalized connection network which

CROSSBAR

PC	Transmission
1	(1 → 2) (1 → 3)
2	(1 → 1)
3	(4 → 2) (5 → 1)
4	(2 → 1) (3 → 2)
5	(1 → 2) (1 → 8)
6	(1 → 5) (1 → 7)

PE_1 (ALU): input-link 1,2
output-link 1

PC	PU-OP-code	OP.1	OP.2	R.B.1	R.B.2
1	$+_2$	R1	R1		R2
2	$+_3$	R1	R2		
3	NOP			R1	R1
4	NOP			R3	R3
5	$+_1$	R3	R3		R4
6	–	R2	R4		

PE_2 (MULTIPLIER 1): input-link 3,4
output-link 2
constant value: b

PC	PU-OP-code	OP 1	OP 2	R.B.
1	NOP			R1
2	$*_1$	R1	R(b)	
3	BUSY			
4	BUSY			
5	NOP			
6	NOP			

PE_3 (MULTIPLIER 2): input-link 5,6
output-link 3
constant value: a

PC	PU-OP-code	OP 1	OP 2	R.B.
1	NOP			
2	$*_2$	R1	R(a)	
3	BUSY			
4	BUSY			
5	NOP			
6	NOP			R1

STATE REGISTER FILE : input-link 7
output-link 4

PC	PU-OP-code
1	
2	
3	Read
4	
5	
6	Write

I/O MODULE : input-link 8
output-link 5

PC	PU-OP-code
1	
2	
3	Read
4	
5	Write
6	

Fig. 10. Machine programs for the notch-filter.

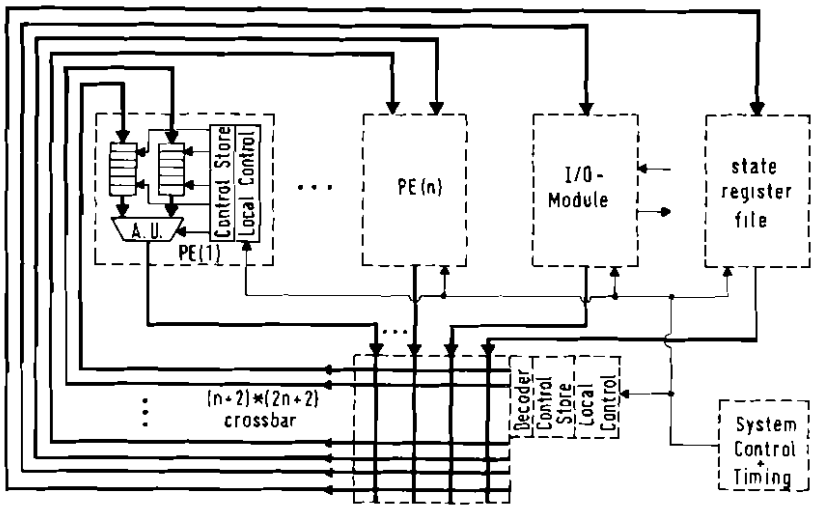

Fig. 11. Machine architecture.

is capable to do so is a crossbar. This is the fastest architectural concept, because all communication links can be generated simultaneously. It has to be outlined that it is also the most expensive concept due to necessary hardware [1, 7, 10, 11].

6. Results

So far different examples of real-time processing have been investigated, the results are tabulated below. The speed-up factor demonstrates the advantage of parallel processing.

Explanation to Table 1:

1: Amount of leaves ($\text{in}(t) = 0$)

2: Amount of inner nodes ($\text{in}(t) > 0$)

3: Available system components (I/O processors + state memories/data processors)

Table 1

Examples

	1	2	3	4	5	6	7
2×2 *Matrix Multipl.*	16	12	10/10	8/8	5	36	7.2
5×5 *Inner Product*	10	9	10/10	5/10	8	29	3.6
3×3 *Outer product*	12	9	10/10	6/6	5	27	5.4
3×3 *Tridiag. Matrix Multipl.*	54	45	10/10	10/10	7	60	8.6
FFT-Radix-2-4.	40	33	10/10	8/10	8	53	6.6
Butterfly-Op.	6	4	10/10	2/3	5	8	1.6
Notch-filter	14	12	10/10	2/3	7	15	2.1
Dig. Filter 1	28	25	10/10	3/1	15	24	1.6
Dig. Filter 2	22	19	10/10	5/9	14	49	3.5
Dig. Filter 3	28	25	10/10	3/7	13	31	2.4
Notch-filter with bus-time	14	12	10/10	2/3	10	21	2.1
3×3 *Matrix Addition*	18	9	10/10	9/10	3	27	9

4: Used processors splitted up as above
5: $t1$:= Complete serial computation time
6: $t2$:= Complete parallel computation time
7: Speed-up factor := $t1/t2$

7. Conclusions

In this article we have proposed the methodology of parallel computation of those algorithms, which can be scheduled deterministically. Many of the algorithms in real-time processing, especially digital signal processing, image processing and pattern recognition are of such a form. The solution of the problems of parallelization, optimization as well as scheduling have been pointed out. It has been shown that the constraints of computation time can be achieved by use of several processing elements, which can lead to a speed-up factor of up to 10. The various hardware system components, e.g. data- and I/O-processors as well as memories are connected by a crossbar switch matrix, which is able to path data between the different components within a single time period. However, with increasing number of processing elements the hardware complexity increases especially for this interconnection network. This can be avoided, when using other interconnection structures [1, 6, 7, 8, 10].

Acknowledgement

The authors would like to thank Dr. N. Roethe and H. Roth for many helpful discussions.

References

[1] G.A. Anderson and E.D. Jensen, "Computer interconnection structures: taxonomy, characteristics and examples", *ACM Comput. Surveys*, Vol. 7, No. 4, Dec. 1975, pp. 197–213.
[2] J. Backus, "Can Programming be liberated from the von Neumann Style? A functional style and its algebra of programs", *Comm. of the ACM*, Vol. 21, No. 8, August 1978, pp. 613–641.
[3] J.L. Baer, "A Survey of some theoretical aspects of Multiprocessing", *ACM Comput. Surveys*, Vol. 5, No. 1, Mar. 1973, pp. 31–80.
[4] R. P. Brent, "The parallel evaluation of general arithmetic expressions", *Journal of ACM*, Vol. 21, No. 2, April 1974, pp. 201–206.
[5] K. Ecker, "Organisation von parallelen Prozessen", *Reihe Informatik/23*, BI Mannheim, 1977.
[6] Ph.E. Enslov Jr., "Multiprocessor organization—A survey", *ACM Comput. Surveys*, Vol. 9, No. 1, Mar. 1977, pp. 103–129.
[7] T. Feng, "A survey of interconnection networks", *Computer*, Dec. 1981, pp. 12–27.
[8] M.J. Flynn, "Some computer organizations and their effectiveness", *IEEE Trans. on Computers*, Vol. C-21, No. 1, 1972, pp. 948–960.
[9] R. Gemballa, "Untersuchung von Paralleltransformationen an Teilprogrammgraphen fuer Multiprozessor-systeme und Implementierung eines entsprechenden Transformationssystems", Diplomarbeit, TH Darmstadt, FB Informatik, Nov. 1980.
[10] L.S. Haynes, R.L. Lau, D.P. Seiworek and D.W. Mizell, "A survey of highly parallel computing", *Computer*, Vol. 15, No. 1, Jan. 1982, pp. 9–26.
[11] D.J. Kuck, "A survey of parallel machine organization and programming", *ACM Comput. Surveys*, Vol. 9, Mar. 1977, pp. 29–59.
[12] J. Lenzer, "Formale Beschreibung von Rechnerarchitekturen und Programmsprachen: Implementierung und Anwendung in einem Erzeugenden System fuer Codegeneratoren", Dissertation TH Darmstadt, FB Informatik, June 1981.
[13] J. Lenzer and N. Roethe, "A high level language signal processing system", in: *Proc. EURASIP*, 1980, pp. 369–375.
[14] J. Lenzer and G. Wieber, "On design strategies for parallel algorithms in signal processing using graph models", in: *Proc. ICASSP*, 1980, pp. 939–942.
[15] H. Roth and R. Steinmetz, "Analyse von Algorithmen durch Datenflussgraphen und deren Abbildung auf Multiprozessorsysteme zum Einsatz in der Echtzeitsignalverarbeitung", Studienarbeit, TH Darmstadt, FB Elektrische Nachrichtentechnik, 1982.