SIMD ARCHITECTURES

Bertil Svensson

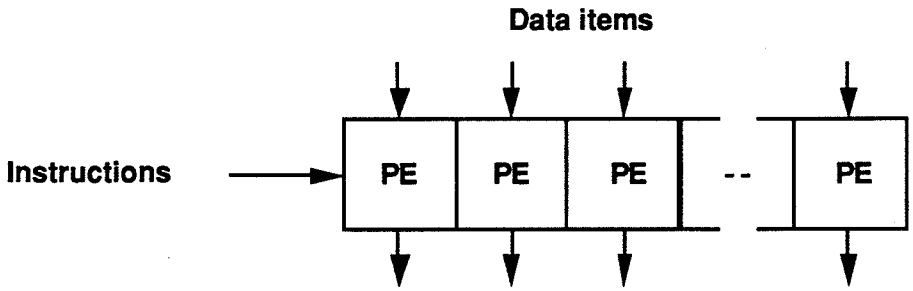
SIMD Processor Array Architectures

(This text is more or less identical to material found in
Lawson, H.W. (with contributions by B. Svensson and L. Wanhammar),
Parallel Processing in Industrial Real-Time Applications, Prentice-Hall, 1992)

SIMD Processor Array Architectures

1. Principles of SIMD

In Single Instruction stream, Multiple Data stream (SIMD) processors one instruction works on several data items simultaneously by using several Processing Elements (PEs), all of which carry out the same operation as illustrated in Figure SIMD-1.

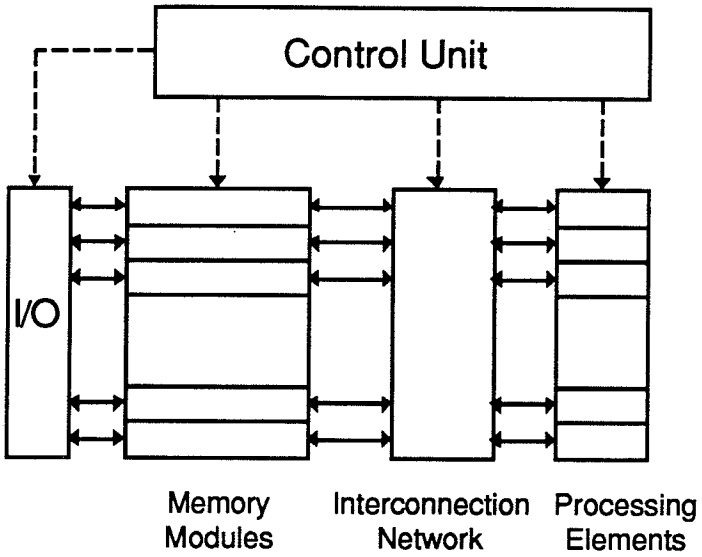


Principle of SIMD processor.

Figure SIMD-1.

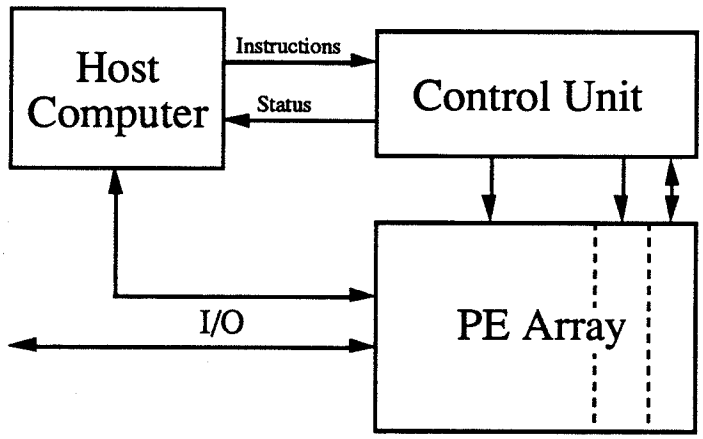
A SIMD Processor has a single Control Unit reading instructions pointed to by a single Program Counter, decoding them and sending control signals to the PEs. Data are to be supplied to, and derived from, the PEs by a memory with as many data paths as there are PEs. Figure SIMD-2a shows the resulting processing structure which is also known as a Processor Array. The Interconnection Network provides flexibility in choosing source and destination for data to and from the PEs, necessary in many algorithms. The I/O system plays the role of converting – typically at very high rates – input/output data between the format of the outside world and the internal format of the array. The design of the I/O system is highly application dependent.

In order to provide overall control of the Processor Array, as well as to execute sequential operations, it is common to use a conventional Host Computer which then views the Processor Array as an attached processor as illustrated in figure SIMD-2b.



Processor Array

Figure SIMD-2a



Host Computer - Processor Array Relationship

Figure SIMD-2b

2. *Array Topologies*

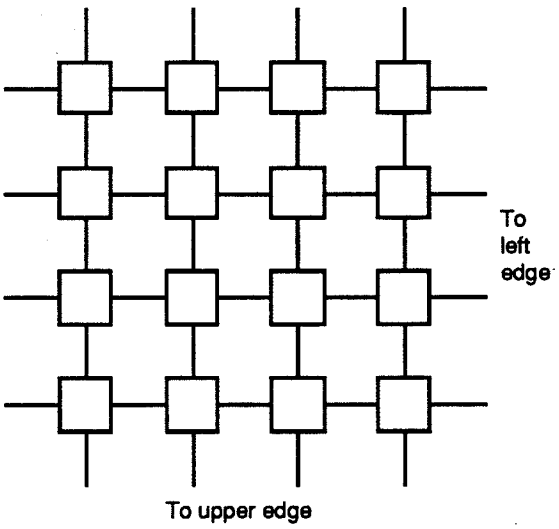
The topology of the processor array is defined by the structure of the interconnection network. The choice of topology is heavily influenced by the demands of the applications that the architecture is primarily designed for. Looking back at the short history of SIMD

processor arrays one may observe that many of the architectures have been designed for specific application domains. Typically, the designers have tried to combine the communication demands of the specific area with more general demands.

We will introduce the most common topologies and show the interconnection structures of several existing (or earlier existing) research machines and commercial machines. The list is by no means exhaustive, but chosen to demonstrate the variety of topologies.

2.1 Mesh connected arrays

The *twodimensional mesh* as illustrated in figure SIMD-3 is the most common topology of SIMD processor arrays. The main reason for this popularity is its obvious support for close local connections, which are exploited in several application areas. The main drawback of the 2D mesh is the relatively large maximum-distance value, *Dmax*. With wrap-around connections at the edges as in figure SIMD-3, *Dmax* for an N processor array is equal to the squareroot of N (e.g., for a 1024 processor array, *Dmax* is 32).



Twodimensional mesh with wrap-around connections
at the edges

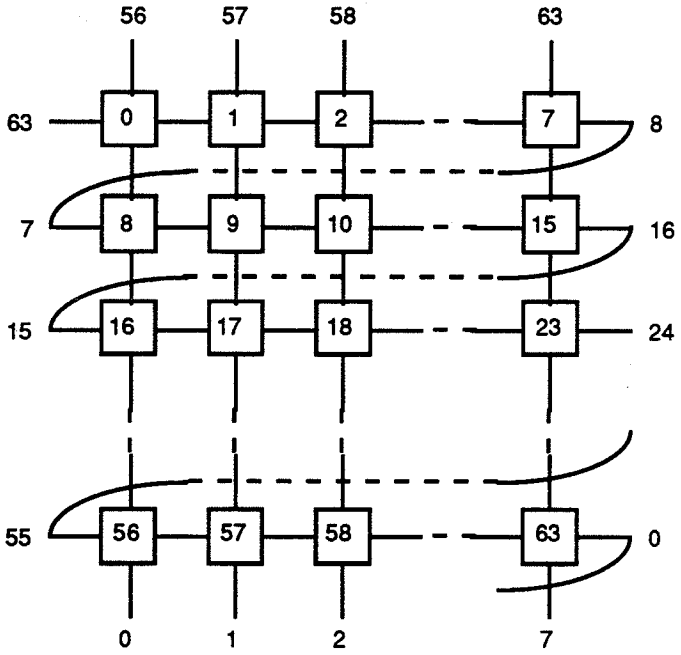
Figure SIMD-3

ILLIAC IV

The first processor array to be implemented was ILLIAC IV – "the first Supercomputer".

Designed at University of Illinois under the leadership of Daniel Slotnick and implemented by Burroughs Corporation, it was intended to have 256 powerful PEs, divided into four "quadrants", 64 PEs in each. However, only one quadrant was constructed. The topology is shown in figure SIMD-4. As can be seen it is a 2D-mesh with modified horizontal wrap-around connections compared to the one shown in the previous figure. Rather than being connected to PE number 8, PE number 15 is connected to the next PE in order, i.e. number 16. Thus, the interconnection structure of ILLIAC IV can be described as a one-dimensional nearest neighbor structure (a long row of PEs) with additional connections 8 steps forward and backward (the vertical connections in the figure).

ILLIAC IV was operational at NASA Ames Research Center from 1972 to 1981 and remained the world's most powerful computer for its entire lifetime. The original paper on the ILLIAC IV hardware is Barnes et al. (1968).



The topology of ILLIAC IV

Figure SIMD-4

DAP and MPP

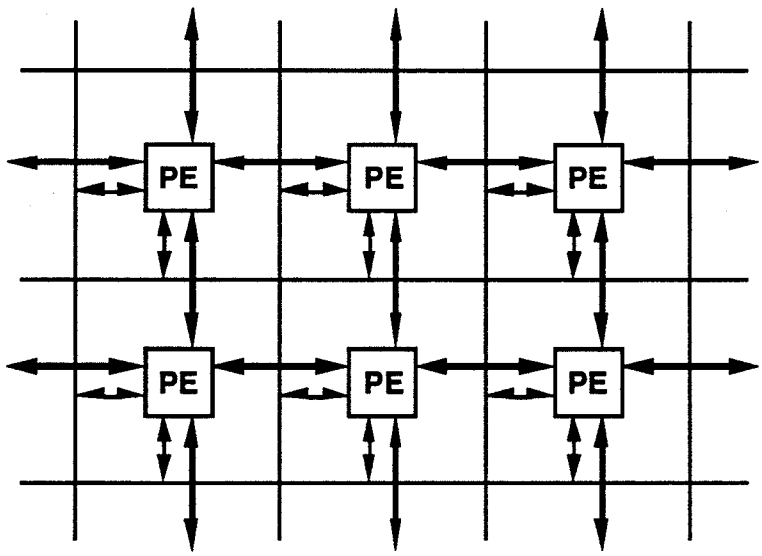
The *Distributed Array Processor – DAP* – produced by ICL (International Computers Limited) and AMT (Active Memory Technology Ltd) as well as the *Massively Parallel Processor – MPP* – from Goodyear Aerospace are the prime examples of large scale two-

dimensional processor arrays. Both use bit-serial PEs (which will be described later). DAP is manufactured in 1024 (32 by 32) and 4096 (64 by 64) PE arrays, while the only MPP machine that was constructed has a 16384 (128 by 128) PE array.

The DAP architecture provides an additional interconnection facility in that two sets of data paths – "highways" – pass along the rows of PEs and along the columns of PEs, respectively, as shown in figure SIMD-5. By these data paths a row (or column) of bits can be broadcast in one cycle so that each row (column) of PEs receives the same data pattern. Another use is in extracting data from the array, where the basic operation is to AND together all the rows or columns, respectively.

Thus it can be seen that provisions have been made to overcome the shortcoming of the ordinary 2D-array – its poor long-distance communication performance.

The chief designer of DAP was Stewart Reddaway of ICL. The main ideas behind his concept can be found in Reddaway (1979). A description of the current line of products from AMT is given in Hunt (1989). MPP was designed by Kenneth Batcher (1980). Potter (1985) is a good reference for further details.



Connectivity for a small section of the DAP.
Horizontal and vertical lines are Row Highways
and Column Highways, respectively.

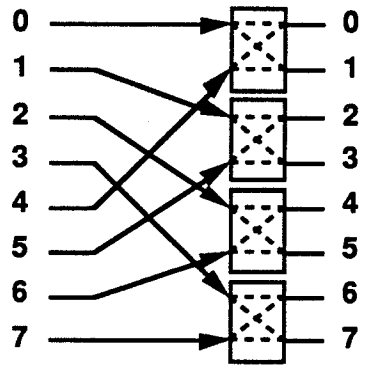
Figure SIMD-5

2.2 Shuffle-exchange networks, direct and indirect binary n-cubes

The *shuffle-exchange network*, see figure SIMD-6, consists of a shuffle function (the arrows) and an exchange function (the boxes). In an N PE array, any destination can be reached in logN iterations, i.e. *Dmax* = logN. For N=1024, *Dmax* is 10, whereas for the 2D mesh it was 32. Thus the maximum-distance value of the shuffle-exchange network is low compared to the 2D mesh. The cost of the network measured in number of connections per PE is also lower.

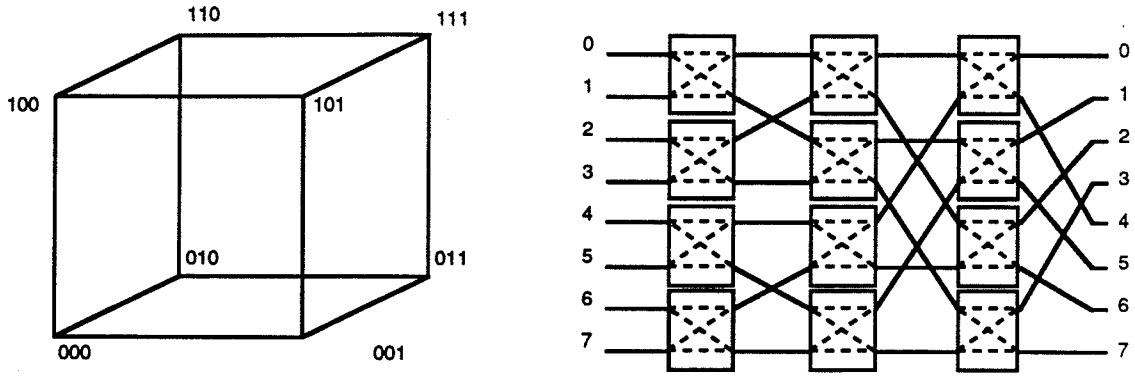
In a *direct binary n-cube network* the PEs are at the corners of an n-dimensional cube (where n=logN), and each PE has an n-way switch. See figure SIMD-7a for n=3. The maximum distance to any other PE is again logN.

The direct cube can be simulated by the *indirect binary n-cube network*, which is a multistage network with only two-way switches. Figure SIMD-7b shows the network for N=8 (n=3).



Shuffle-exchange network

Figure SIMD-6



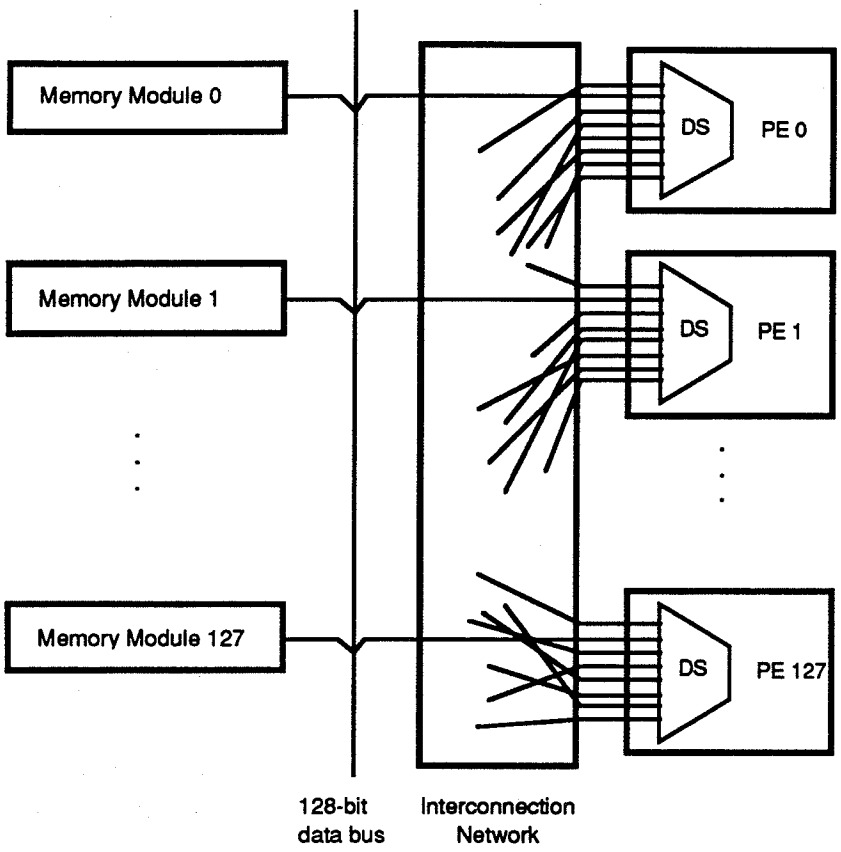
(a) (b)

Direct (left) and indirect (right) binary 3-cube

Figure SIMD-7

LUCAS

LUCAS (Lund University Content Addressable System) was designed and built as a research vehicle to study organization principles, PE design, programming, and application development on SIMD machines. Therefore the interconnection network was made reconfigurable so that different topologies could be chosen. Eight different sources can be wired to the input of each PE, as shown in figure SIMD-8. This allows several simple network structures to be implemented simultaneously. Application development and programming in areas like image, signal, and database processing has been made on a machine with both shuffle-exchange network and nearest-neighbor connections (in one dimension). More details of the interconnection network are given by Svensson (1983). Fernström, Kruzela and Svensson (1986) describes the entire LUCAS project.



Reconfigurable interconnection structure of LUCAS.
All Data Selectors, DS, are controlled by the same code.

Figure SIMD-8

STARAN

STARAN, a commercial product manufactured by Goodyear Aerospace Corporation in the seventies, used a "flip network" between memory modules and PEs. This network is functionally identical to the indirect binary n-cube. More important than the ability to reach any memory module from any PE was probably the many different access patterns that were made available through the network in combination with a clever addressing scheme. See also the historical perspective chapter in Part I of this book. Batcher (1974, 1976, 1977) provides additional details.

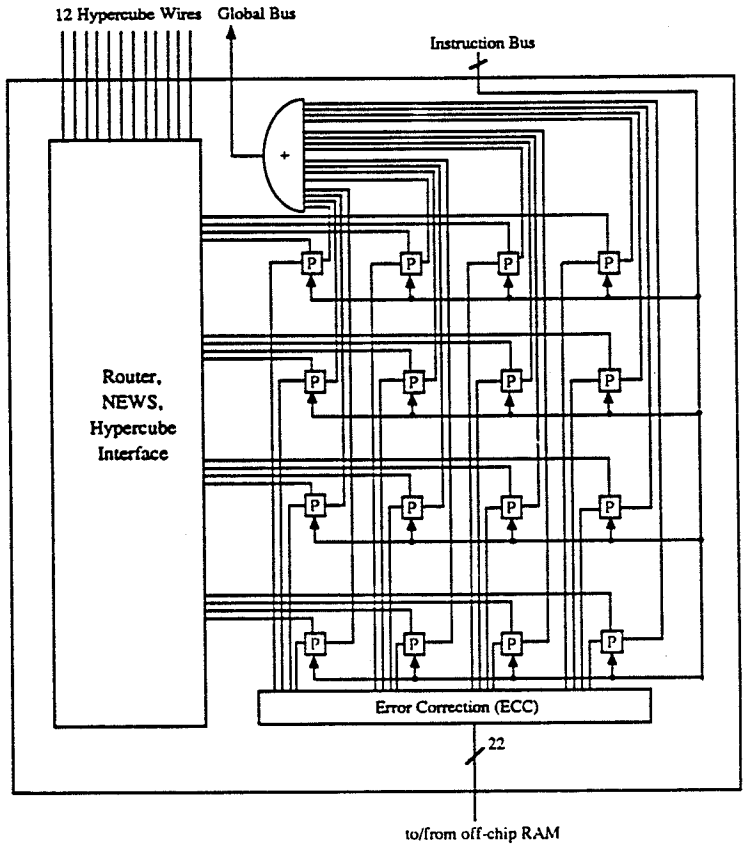
The Connection Machine

The *Connection Machine Model CM-2* from Thinking Machines Corporation uses a 12-dimensional direct binary cube to support general patterns of communication among 4096 nodes (in a fully configured system), each node incorporating 16 bit-serial processors.

The *routers* – one at each node – are capable of dealing with contention for hypercube wires by routing the message alternative (and longer!) ways.

While the router network supports completely general patterns of communication by message passing, additional special hardware supports certain commonly used regular patterns of communication. This is called the *NEWS network,* because one of its uses is to implement a two-dimensional grid (North, East, West, and South connections). However, grids of any dimension up to 31 are supported. For example, possible grid configurations for 64K processors include 256 x 256, 1024 x 64, 65536 x 1, 64 x 32 x 32, and 16 x 16 x 16 x16. The structured NEWS communication is significantly faster than general routing.

Figure SIMD-9 shows a node of 16 PEs including Router, NEWS, and Hypercube interface.



CM-2 Processor Chip (from (Thinking Machines Corporation, 1989))

Figure SIMD-9

When the parallel data structure has more elements than there are physical processors the system operates in *virtual processor* mode. This means that each hardware processor simulates (by sequential execution) a number of virtual processors, each with a correspondingly smaller memory. The virtual processor mode is supported by microcode.

As an example of the way in which a data structure and a communication demand may be mapped onto the architecture, imagine a set of 2^{22} virtual processors (about 4 million) that we want to organize as a 2048 x 2048 square grid. We assume a 64K Connection Machine system.

First, a 2-dimensional grid of shape, say, 64 x 64 is embedded in the boolean 12-cube. This can always be done using Gray-coding of the grid coordinates and implies selection of a subset of the wires that form the 12-cube. Each of the 64 x 64 nodes has 16 physical processors arranged as a 4 x 4 grid. Within each physical processor we now need 64 virtual processors, which we imagine are arranged as an 8 x 8 grid.

In order for each virtual processor to send a value to its east neighbor, three different types of communication need to take place. Within each group of 64 virtual processors (implemented in a single physical processor), 56 of them have their east neighbor within the same physical processor, i.e. the work is done by having each physical processor rearrange data within its own memory. The remaining eight values are to be sent to the physical processor to the east. In some cases this is within the same 16 PE node, in some cases it is not.

Within each group of 16 physical processors, 12 of them must send data to another physical processor that is within the same chip. This transfer is thus independent of the entire router/hypercube-wire mechanism.

Remaining to be sent over the wires is only 1/32 of the total communication. Each node has to send 32 messages to its east neighbor along one hypercube wire.

All three types of communication are supported by specialized hardware on the CM-2, allowing the communication to take place without use of the slower general routing mechanism. This is true for grid communication patterns of any dimension and is handled by microcode.

In conclusion, a very general network, the boolean cube, has been selected as the basic communication structure for the CM-2, but special hardware support has been added for the most common regular structures – the multidimensional grids.

2.3 Linear arrays

Several linear arrays of processors have been built, mainly for use in low level image processing. The earlier mentioned general purpose systems STARAN and LUCAS have a linear array structure in addition to shuffle and flip networks respectively. Examples of pure linear arrays, i.e. with only nearest neighbor interconnections in one dimension, are LAPP (Forchheimer and Ödmark, 1983), SLAP (Fisher and Highnam, 1985), and AIS-5000 (Schmitt and Wilson, 1988). LAPP integrates a row of photodiodes and processing elements into a single chip and is designed for high speed, low cost industrial quality measurement and inspection, which will be demonstrated in an application study in Part V of this book.

AIS-5000

The AIS-5000 (Schmitt and Wilson, 1988) is a commercially available system from Applied Intelligent Systems Inc. It has up to 1024 processing elements arranged in a one-dimensional chain that, for computer vision applications, can be as wide as the image itself. The PEs are bit-serial and each PE has its own bit-wide RAM of (currently) 32k bits. The system integrates 128 processors on each board, thus the whole processing array is a small desktop system. Its main use is in industrial vision systems where the speed requirements are far greater than what can be met by ordinary, sequential processors. Processing of images is done line by line.

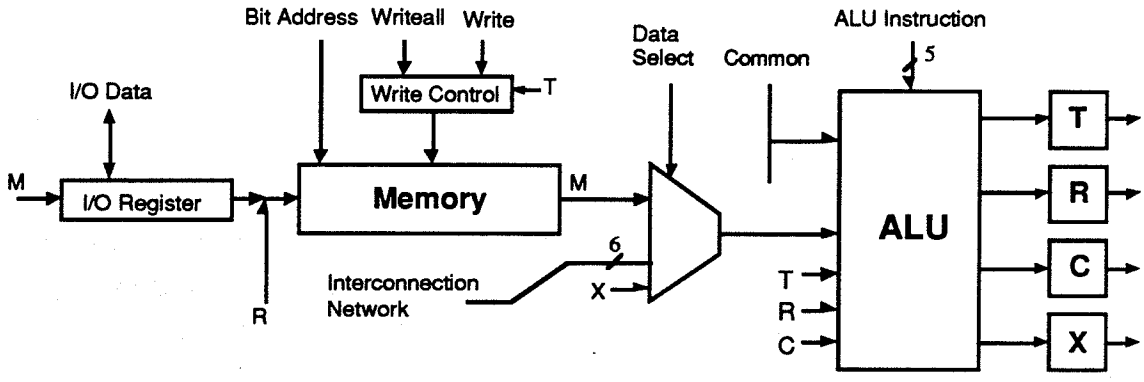
3. Processing Element (PE) Design

In the design or choice of a processor array there is a traditional trade-off problem, namely between the power of the Processing Elements and the size of the array. One extreme is based on "massive parallelism" and very simple, bit-serial processors. This is the philosophy represented by e.g. DAP, MPP, Connection Machine and AIS-5000. At the other end of the scale we find designs like PICAP3 (described below) with a moderate number of specialized floating point processors. Surprisingly enough, the two extremes may be combined in the same design: The Connection Machine represents both schools in that it, in addition to its tens of thousands of bit-serial processors, also has thousands of 32-bit floating point processors. The two sets of processors share the same communication structure and memory.

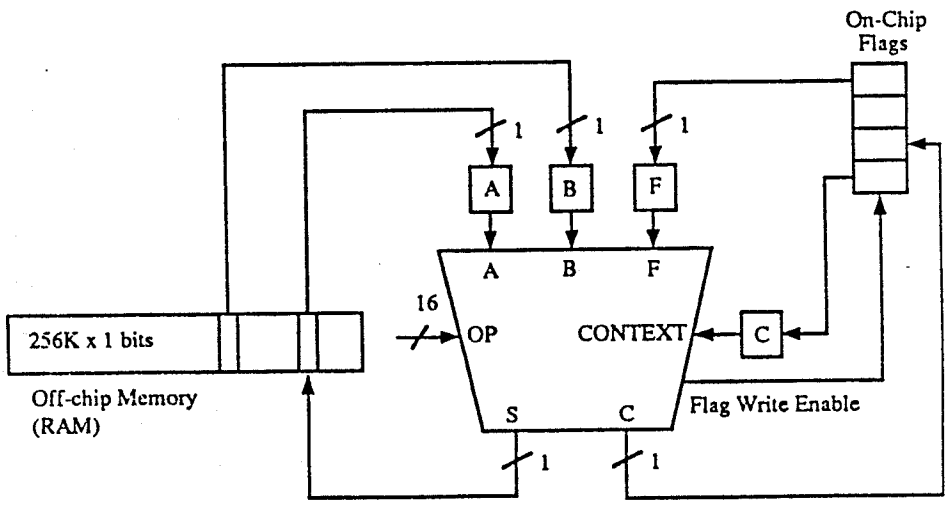
There is a connection also between the power of the PEs and the interconnection structure. Multi-bit PEs may require multibit data paths to match their processing bandwidth. This in turn restricts the degree of connectivity of the array.

3.1 Simple bit-serial processors

The machines that show the highest degree of parallelism also show the simplest processor designs. The DAP and the Connection Machine both have bit-serial PEs that can perform a full adder operation and various other logical operations on one, two, or three bits. They also have a small set of one-bit registers to store intermediate results, and an activation flip-flop. The PEs of LUCAS and AIS-5000 also belong to this category. As an illustration of the degree of complexity found in these PEs, figures SIMD-10a and 10b show the bit-serial PEs of LUCAS and the Connection Machine CM-2, respectively.



(a)



(b)

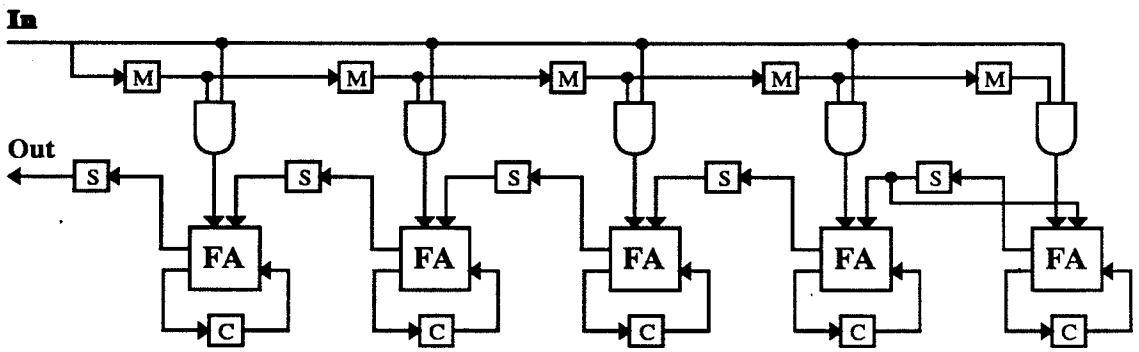
Examples of bit-serial PEs:
a) LUCAS, b) Connection Machine

Figure SIMD-10

3.2 Enhanced bit-serial processors

Multiplication is a common operation in many of the applications in which processor arrays are used, e.g. in signal and image processing. A serious drawback of simple bit-serial processors is that multiplication time grows quadratically with the data length. However, there is a method of doing bit-serial multiplication that requires no more time than what is needed to read the operands (bit by bit, of course) and store the result (also bit by bit). The method, based on a carry-save adder technique, requires as many full adders as the data length. Figure SIMD-11 shows the design for multiplication of two 2's complement integers. It is operated by first shifting in the multiplicand, most significant bit first, into the array of M flip-flops. The bits of the multiplier are then successively applied to the input, least significant bit first, and the product bits appear at the output, also least significant bit first.

The design, which was proposed but not implemented in the LUCAS project, will now be used in the "Hampus" system, an implementation of a configurable, modular processor array building upon the experiences of LUCAS (Svensson, 1989). A similar multiplier design has been proposed for the "Centipede", a further development of the AIS-5000 concept (Wilson, 1988).

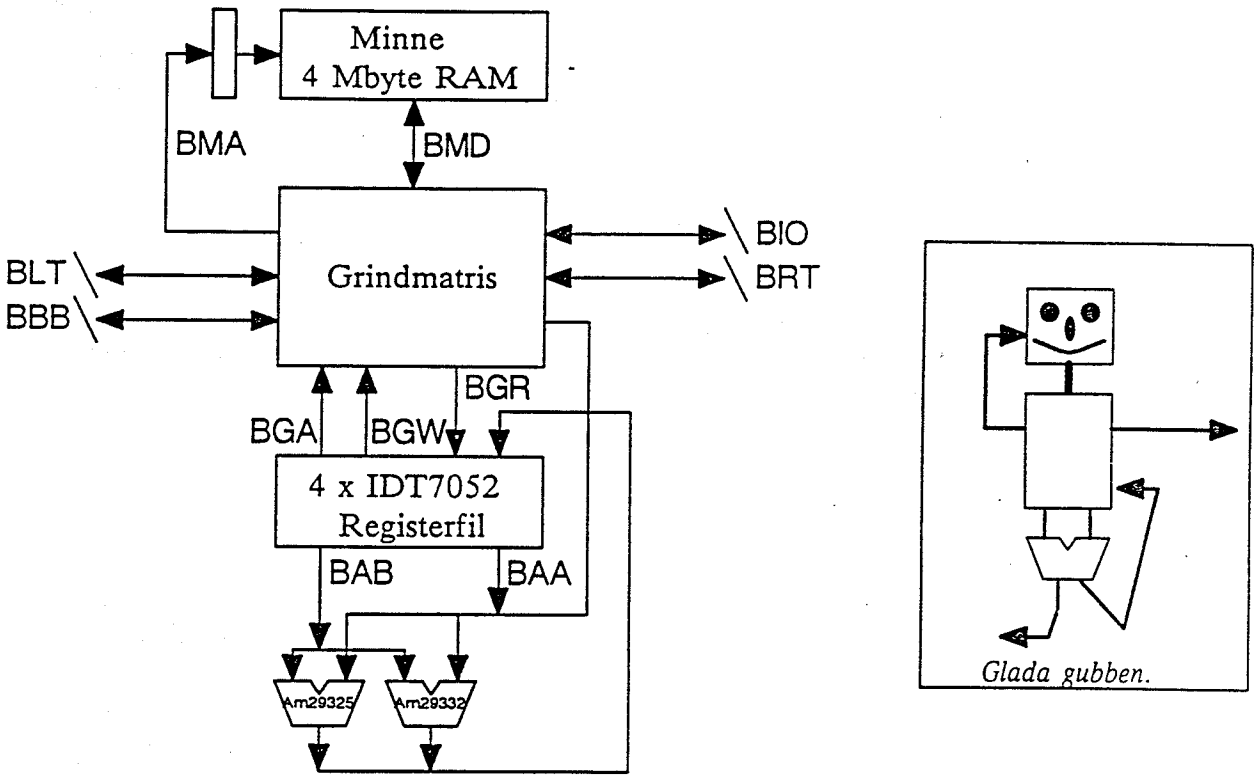


Principle of bit-serial multiplier used in the PEs of Hampus-1

Figure SIMD-11

3.3 Bit-parallel, floating point PEs

While still conforming to the SIMD principle,the proposed and partially implemented PICAP3 design (Lindskog 1988, Segerström 1990) shows processing elements that are considerably more complex than those that we have considered so far. See figure SIMD-12. Each PE comprises 32 bit integer and floating point arithmetic (Am29332 and Am29325), a four-port register file, and a set of complicated gate arrays to implement functions like local addressing, microword decoding, memory control, and parity check. No more than four PEs with associated memories (4 Mbyte each) can be implemented on one circuit board. A PICAP3 system is anticipated to have up to 64 PEs.



A processing element ,"Glada gubben" (Merry man), in PICAP3.32

Figure SIMD-12

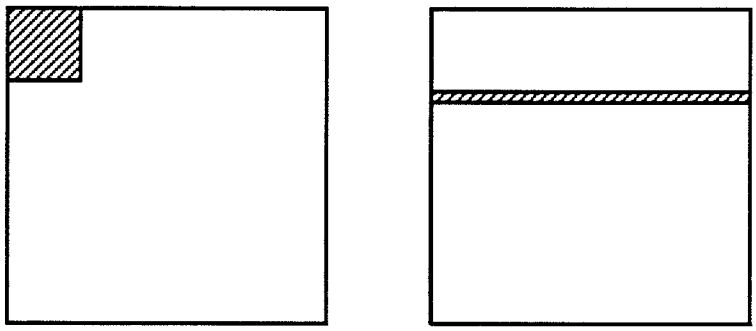
4. Problem Parallelism vs Machine Parallelism

To achieve highest efficiency, a SIMD processor array requires the problem to expose at least the same degree of parallelism as the array. In nearly all applications that call for large amounts of computing power this is the case. However, it is not always required

that the machine is as parallel as the problem. In addition, it may imply difficulties, concerning e.g. communications and input/output, to try to utilize the entire amount of parallelism. The purpose of this section is to show trade-off situations and demonstrate the various consequences of the decisions made.

4.1 Image processing approaches.

In MPP, with its 16384 PEs arranged as a 128 x 128 grid, one PE is typically assigned to each pixel. Thus, images are divided into square parts, sized 128 x 128, when processed (see figure SIMD-13a). This method implies a quite complicated I/O system to reorder data. The "staging memory" that accomplishes this in the MPP is a large part of the machine (a part that was actually added late in the design phase). The method also implies neighborhood access problems at the edges of the subimages, problems which must be taken care of at the lowest level of programming.



(a) (b)

Processing part of an image
a) subimage by subimage b) line by line

Figure SIMD-13

With LUCAS and AIS-5000, organized as linear arrays, one PE takes care of one line of the image, see figure SIMD-13b. This simplifies I/O significantly, at least in the case when the image is input and output in line-scan format. Normally, the PEs have enough memory to allow the whole image to be stored in the array. The memory of each PE then stores a whole pixel column. Vertical neighbor pixels are in the same PE, horizontal ones are in neighboring PEs. Long distance communication in the vertical direction is achieved at no cost (within the same PE).

The linear array approach to image processing has a limit (by definition): the number of PEs cannot exceed the number of pixels of a line. However, duplication of arrays can be

made to enhance performance if needed. This can be arranged as parallel working arrays or arrays working in pipeline fashion.

To make this collection of mapping methods more complete we should also mention that each PE in a 2D array can process a small, coherent part of the image. Reordering of data at input/output is needed also in this case, although it can be done one line at a time, thus requiring less of staging memory. The neighborhood access patterns are different for different pixels within the subimage, but the instantaneous pattern is the same for the whole set of physical PEs.

The three examples (subimage-on-whole-array, line-by-line, and subimage-on-PE) demonstrate that virtual processors can be mapped to physical processors in several ways, giving different consequences for input/output and computations. Choosing organization and mapping carefully is as important as using as many processors as possible in the solution of a problem. For example, it might be beneficial to use fewer, but more powerful, processors and a simple and efficient mapping and input/output scheme.

4.2 Neural net approaches.

Processor arrays are candidates for performing the computations of neural network models efficiently. The computations involved are uniform and arithmetically simple. This suggests that simple processing elements are sufficient and that the SIMD type of architecture is appropriate.

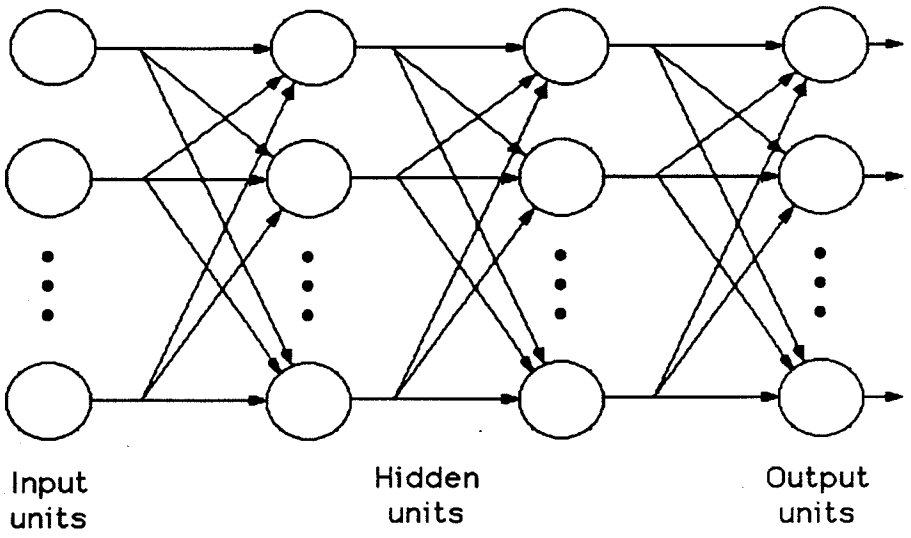
The number of interconnections in even an artificial neural network is often orders of magnitude greater than the number of available processing units. In addition, extensive training sets are normally used, in some models they can be run independently of each other. Thus the problem parallelism exceeds the machine parallelism with orders of magnitude, leaving the system designer with a multidimensional design space.

A popular neural net model is the multilayer feedforward network with error back-propagation, shown in figure SIMD-14. In the first phase of computation the input to the network is provided and values propagate through the network to compute the output vector. In these computations each neuron first computes a weighted sum of all its inputs. Then it applies an activation function to the sum, resulting in an activation value - or output - of the neuron. Usually a sigmoid function with a smooth threshold-like curve is used as activation function.

The output vector of the network is then compared with a target vector, which is provided by a teacher, resulting in an error vector. In the second phase the values of the error vector

are first propagated back through the network. The error signals for hidden units are thereby determined recursively: Error values for layer *l* are determined from a weighted sum of the values of the next layer, *l*+1, again using the connection weights - now "backwards". The weighted sum is multiplied by the derivative of the activation function to give the error value.

Now, finally, appropriate changes of weights and thresholds can be made. The weight change in the connection to unit i in layer *l* from unit j in layer *l*-1 is proportional to the product of the output value and the error value.



Neural network of the multilayer feedforward type

Figure SIMD-14

Mapping of neural network computations on SIMD arrays have been studied by, among others, Brown et al. (1988), Svensson and Nordström (1990), Nordström (1990), and Singer (1990). Nordström identifies six different ways of achieving parallelism: (i) each node in a layer can be mapped to one PE, (ii) each interconnection can be mapped to a PE, (iii) the training examples can be mapped to different PEs, (iv) different training sessions (e.g. different starting weights) can be started on different PEs, (v) the different layers can be pipelined, and (vi) the forward and backward passes can be run simultaneously on different patterns.

Svensson and Nordström use method (i). Brown et al. use method (ii) since they run a rather small network on a fairly large array. Singer runs several training sessions simultaneously, i.e. method (iii), which turns out to be the most efficient method on the

Connection Machine (minimizes the need for communication).

5. Input/Output for SIMD Arrays

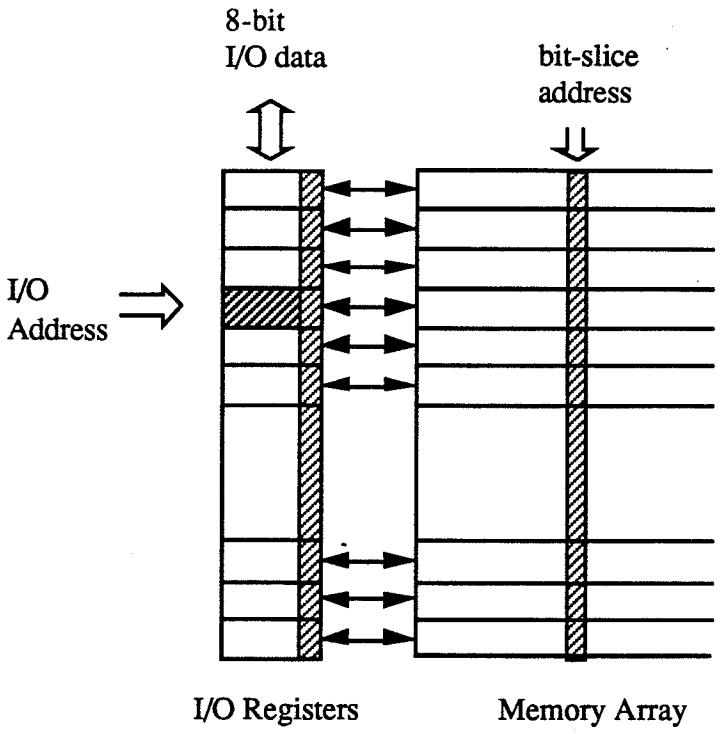
In many applications, especially in real time situations, high data rates into or out of the processing array are required. To obtain a well balanced architecture, the design of the I/O system is as important as the design of processing elements and array topology.

Input/output can be discussed in terms of different classes of data formats:

(1) *Conventional format*: Word-sequential, bit-parallel format, i.e. the traditional data format used by conventional computers.
(2) *Processing format*:
(2a) Bit-slice or bit-plane format, which is the processing data format of arrays with bit-serial PEs, or
(2b) Word-slice or word-plane format, which is the processing data format of arrays with bit-parallel PEs.
(3) *Application format*, e.g. the format of an image or a data base (often this is a multi-dimensional array).

The input/output system shall serve as an interface between the different formats. In real time applications, with processor arrays directly connected to input and output devices, the interface between the application data format (3) and the processing format (2) is the most important, but transfer between the word-at-a-time format of the front-end processor and the processing array must also be efficient.

We illustrate the transfer between the different formats on the I/O system of LUCAS. See figure SIMD-15 and figure SIMD-10a. A set of 8-bit I/O Registers (shift registers) is connected to the Memory Array, one register per memory word. The I/O Registers can be read or written from the Front-End Processor or dedicated I/O Processor in the conventional word-at-a-time format. A data input process can be divided into two phases: one to fill the I/O Registers from the Front-End or I/O Processor, one to shift the contents out of the I/O Registers and into any field of the Memory Array. The first phase needs one write cycle of the Front-End or I/O Processor to transfer 8 bits, the second phase requires 8 Memory Array write cycles, in each of which an entire bit-slice is transferred.



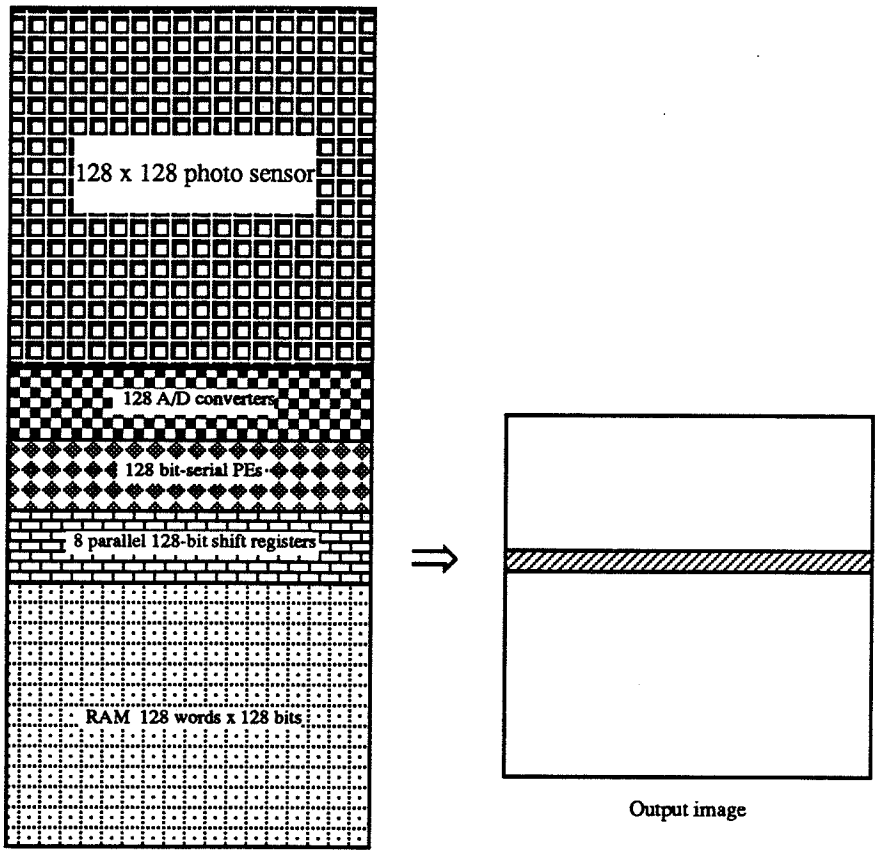
Input/Output structure of LUCAS

Figure SIMD-15

A variation of this input/output scheme that is used in some designs does not allow for addressing of the I/O registers. Instead, data is shifted in and out of the register set. Such I/O register systems may be used also for communication between the PEs.

PASIC (Chen et al. 1990), see figure SIMD-16, is an example of a design with such a bit-parallel shift register along the row of PEs. This can be used for image output as well as for inter-PE communication. However, for image input there is a direct, word-parallel, bit-serial interface between the photosensor array (128 by 128 pixels in present version) and the linear array of PEs. Thus, PASIC is an example of a design which has special facilities to interface between the data format of the application and the processing data format, resulting in a significantly higher input speed. PASIC is primarily intended for low-level vision processing where high speed image I/O is required.

PASIC is a further development of the LAPP concept which is treated in the *Application case studies* in Part V of this book.

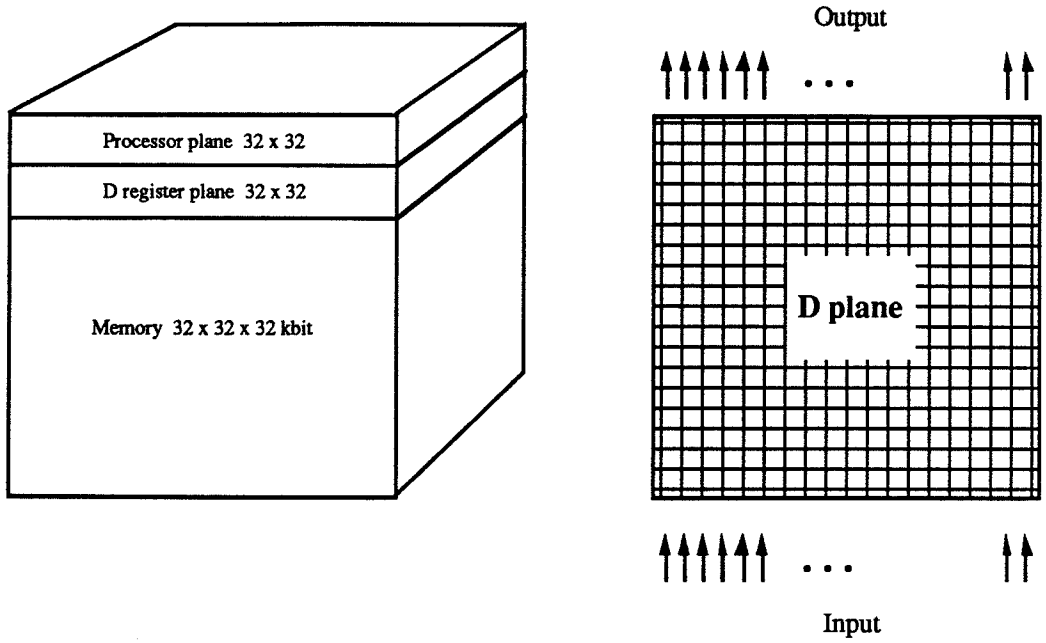


PASIC overview

Figure SIMD-16

The high-speed I/O facility of the two-dimensional AMT DAP processor array (Hunt, 1989) is depicted in figure SIMD-17. One of the one-bit registers in the PEs, the D register, may be loaded from the memory or written back to the memory, but does not otherwise take part in PE operations. The D plane may be shifted towards the North edge of the array, so that successive rows of the plane are output at that edge; at the same time successive data words may be presented as input at the Southern edge. The shifting is done independently of the normal array instruction stream and may be done at a faster clock rate. It is controlled by one of a number of input/output couplers rather than by the array control unit, and may be thought of as a DMA facility.

After the shifting of a plane is finished, it is stored to – or the next plane is loaded from – the array. This is done by a request to the array control unit, thus delaying the normal instruction stream for one clock cycle. Reordering of the data is needed in several cases; to this end couplers with double buffer arrangements are used.



AMT DAP 500

Figure SIMD-17

6. Programming SIMD Arrays

6.1 Instruction level

The "instruction level" of a processor array shows powerful instructions, performing tasks that need procedures with loops on sequential computers. Examples of instructions are:

Maximum value of a field (vector, matrix etc.). On a bit-serial array this is found by traversing the bit-slices from most to least significant bit and successively discarding values with a 'zero' if there are others still active with a 'one'.

Exact match, or *Closest match* between a constant and the values of a field.

Pairwise multiply between the elements of two vectors, matrices etc.

Typical for the instructions is also that the effect of the operations can be limited to a certain subset of the processors. The subset is specified by an array of zeros and ones, a *selector*. Selectors are often determined through an associative process, i.e. processors are selected on the basis of the contents of their respective memories.

Based on the type of operands and type of result six basic types of instructions to manipulate data in the memory array can be identified:

Instruction type	*Example*
field --> field	Increment field, permute field
field --> selector	Maximum/minimum value of a field
field,field --> field	Multiply fields, pairwise max of vector elements
field,field --> selector	Pairwise equality between vector elements
constant,field --> field	Multiply by constant, AND with constant
constant,field --> selector	Exact match, closest match, greater than

6.2 High-level languages.

High-level languages for SIMD processor arrays are often referred to as *data parallel languages*. Typically, they are very similar to conventional languages but allow the programmer to organize data so that operations may be applied to many elements of data simultaneously. This is accomplished by adding new data types and extending the meaning of existing program syntax when applied to parallel data. Extension of the control structure is often also done.

When programming in data parallel style, emphasis is put on the use of large, uniform data structures, such as arrays, whose elements can be processed all at once. For example, the statement **A=B+C** indicates many simultaneous addition operations if **A**, **B**, and **C** are declared to be arrays. Each array element is in the memory of a different processor, or at least in the memory of a different virtual processor.

Thus, much of the parallel code looks just like sequential code. The compiler has to examine the declarations to determine whether **B+C** will require a single addition operation (in the front-end processor) or thousands (in the PE array).

Scalar and parallel values may be mixed in a program, for example when multiplying every element of an array by a constant. In that case the scalar value is broadcast to all processors at once. Also, an operation on parallel data may yield a scalar result, for example when finding the sum or maximum of all the elements of an array. In this case a reduction operation is performed, which can be supported in various ways by the hardware (e.g. by the ability to form a binary tree).

Conditionals are implemented in SIMD arrays by limiting the impact of operations to a certain subset of processors. This is often achieved through the use of a parallel data type

called *selector*.

Pascal/L and Parallel Pascal

Pascal/L (Fernström, 1982) is an extension of Pascal for parallel processing, developed in the LUCAS project . In Pascal/L the parallelism of the architecture has a correspondence in the syntax of the language. Thus, constructs in the language are directly implementable as elementary operations of a SIMD processor array.

There are two kinds of parallel variables: **selectors** and **parallel arrays**. A selector defines a boolean vector distributed over the Memory Modules and is intended to control the parallelism of operations. (At execution time, the Activity Registers are set in those PEs where the corresponding selector element has the value TRUE). A parallel array consists of a fixed number of components which are all of the same type and which are located in the Memory Modules.

For example,

```
var   SEL       : selector [0..999] := (0..399 -> TRUE);
      WEIGHTS   : parallel array [0..999,0..999] of integer (12);
```

declares a selector with elements in the first 400 MMs selected and a 1000 by 1000 matrix of 12-bit integers located in the first 1000 MMs, 1000 components in each MM.

An indexing scheme allows simultaneous access to a column or a subset of the column components of a two-dimensional array. For example, WEIGHTS[*,5] selects column 5 of WEIGHTS, and WEIGHTS[SEL,5] selects a subset of column 5 of WEIGHTS. A parallel array may be used without any index at all (and no brackets), in which case all components of the array are referenced. Parallel variables are allowed in expressions and assignments, e.g. 4+WEIGHTS[*,5] adds 4 to all components of column 5.

New control structure concepts are included in Pascal/L to allow control of selection and repetition along the parallel dimension:

The **where do elsewhere** construct defines different actions to take place in different Memory Modules depending on the contents of a selector:

```
where SEL do WEIGHTS[*,10] := 2*WEIGHTS[*,10]
          elsewhere  WEIGHTS[*,10] := 0;
```

The **case where** statement and the **while and where do** statement are the two other extensions defined.

In expressions and assignments where the corresponding components of the parallel variables are located in different Memory Words, the variables must be aligned. The kind of alignment needed is defined by the programmer in terms of standard alignment functions which correspond to the data movements over the interconnection network.

For example, if the perfect shuffle connection is included:

```
M[*,0] := shuffle(M[*,1])
```

To support associative processing a number of standard functions and procedures are defined. The **first** function is used to find the first component of a selector with the value TRUE. It returns a new selector with only this element true. The **next** procedure assigns the value FALSE to the first TRUE element of the selector. This is useful when processing selected elements sequentially. The **some** function, finally, returns the value TRUE if there is at least one TRUE element of the selector, otherwise it returns the value FALSE.

Parallel Pascal (Reeves, 1984) was designed with the MPP as the initial target architecture and was the first high level programming language to be implemented on the MPP. The extensions are similar to those of Pascal/L. They include also a set of standard functions that implement reduction operations: **sum, product, all, any, max, and min**, all operating on arrays.

For example, given the definition

```
var  a:  array [1..100,1..5] of integer;
     b:  array[1..100] of integer;
     c:  integer;

b := sum(a,2)
```

computes the sum of the rows of a, and

```
c := sum(a,1,2)
```

computes the sum of all elements of the array a.

Fortran-8x, DAP Fortran, CM Fortran

Extensions to Fortran 77 for parallel processing have been proposed in a draft ANSI standard, *Fortran 8x*. Many of the features in the proposal emerge from *DAP Fortran* which was developed for the DAP. Fortran for the Connection Machine, *CM Fortran*, which will be our example, implements the array features of Fortran 8x.

The most important difference between Fortran 77 and Fortran 8x is that expressions in Fortran 8x can treat entire arrays as atomic objects. For example, in the statement A=B+C, A, B, and C may be scalars, vectors, matrices, or multidimensional arrays.

Arrays are stored in the Connection Machine with one element per virtual processor. The arrays map directly onto the multidimensional communications grid of the Connection Machine system.

CM Fortran also includes functions that inquire about array attributes, perform data reduction, or perform other complicated array operations. Examples of reduction oprations are SUM, PRODUCT, MAXVAL, MINVAL, ANY, ALL, and COUNT. Examples of high-level operations are DOTPRODUCT (vector dot product) and MATMUL (matrix multiplication).

A reduction operation may take a MASK argument, so that only selected processors participate in the operation. For example, the expression SUM(A, MASK=A.GT.0) sums only the positive elements of the array A.

Like the Pascal based languages discussed above, CM Fortran has an extended control structure. For example, in the code

```
WHERE (B .NE. 0)
  C = A / B
ELSEWHERE
  C = 1.0E30
END WHERE
```

where A, B, and C are all conforming arrays, the result of A/B is assigned to C in each processor containing a non-zero element of B, and 1.0E30 is assigned to C in all other processors. As another example, the following code clears the part of the matrix H that is below the diagonal:

```
FORALL (I = 1:N, J = 1:N, I .GT.J)  H(I,J) = 0.0
```

Note the use of a mask expression in addition to the index variables I and J.

Our description of CM Fortran (as well as C* and *Lisp below) is based mainly on the Connection Machine Technical Summary (Thinking Machines Corporation, 1989). More details can be found in (Thinking Machines Corporation, 1990) and the CM programming manuals. Hockney and Jesshope (1988) have a rather detailed treatment of data parallel versions of Fortran and Lisp.

*C**

*C** is a parallel dialect of C that was developed at Thinking Machines Corporation but which has also been implemented on other machines than the Connection Machine.

Two new storage classes are added, that describe where the data reside. The keywords used are **mono** and **poly** . Scalar (**mono**) data reside in the memory of the front-end, and parallel (**poly**) data reside in the memory of the processor array. Existing operators are extended to operate on parallel data. Two rules are added to the usual rules of C evaluation: The *Replication Rule* states that a scalar value is automatically replicated where necessary to form a parallel variable. The *As-If-Serial rule* states that a parallel operator is executed for all active processors as if in some serial order. The latter is a simple way of stating the guarantee that, from the programmer's point of view, the processors do not interfere with each other (while still permitting a parallel implementation).

The C* compiler for the Connection Machine computer system is implemented as a translator to ordinary C code that is then compiled by an ordinary C compiler for the front-end computer. The C* compiler parses the C* source code, performs type and data flow analyses, and then translates parallel code into a series of function calls that invoke operations on the machine instruction level.

**Lisp*

The **Lisp* language is an extension of Common Lisp for programming the Connection Machine in a data parallel style. It supports primitives that correspond directly to the operation of the hardware. Therefore it is possible to write code that executes very efficiently.

As was the case with the extensions to other languages treated above, a new, parallel data type has been added: A **pvar** (parallel variable) is a Common Lisp data object that has a value in each processor, virtual or physical, of the Connection Machine.

There are two ways of viewing a pvar. In one model, each processor is simultaneously running the same Common Lisp program, and the pvar represents a variable that exists in all processors and gets operated upon simultaneously in all processors. In the other model, the pvar represents an array whose size is the same as the number of processors. The elements of the array are located in consecutive processors.

As described earlier in this chapter, the Connection Machine provides very flexible communication between processors by packet switching. *Lisp provides functions for using this communications system. The **pref!!** function allows each active processor to simultaneously read the value of a pvar in any processor. Even if two or more processors attempt to read the data of a single processor, they all receive the same correct data. ***pset** allows each active processor to simultaneaously write the value of a pvar to any other processor. If two or more values are destined for the same place, the user can specify how they are to be combined (for instance by adding the values together).

*Lisp also includes reduction functions, e.g. ***min**, ***sum**, and ***logior** (bitwise logical OR). For example, (***all** (***sum** (!! **1**))) will sum together the quantity 1 in all processors in the Connection Machine. The result will be the number of virtual processors in use at that moment.

6.3 Conclusion

Data parallel programming is astonishingly simple and may be applied successfully to many more problems than was originally conceived. It has been a long lasting conception that efficiency could be achieved only in certain very regular calculations, and that it in general is very difficult to map a problem onto a given machine structure. Several years of use of data parallel machines have changed this view. Hillis and Steele (1986), after having presented several examples of data parallel algorithms, state:

> "Our current view of the applicability of data parallelism is somewhat broader. That is, we are beginning to suspect that this is an appropriate style wherever the amount of data to be operated upon is very large.
> ... One potentially productive line of research in this area is searching for counterexamples to this rule: that is, computations involving arbitrarily large data sets that can be more efficiently implemented in terms of control parallelism involving multiple streams of control. Several of the examples presented in this article first caught our attention as proposed counterexamples.
> ... Having one processor per data element changes the way one thinks. We found that our serial intuitions did not always serve us well in parallel contexts. For example, when sorting is fast enough, the order in which things are stored is often unimportant. Then again, if searching is fast, then sorting may

be unimportant. In a more general sense, it seems that the selection of one data representation over another is less critical on a highly parallel machine than on a conventional machine since converting all the memory from one representation to another does not take a large amount of time."

Not only that data parallel algorithms are easier to find than was expected, they are also easier than expected to program in high-level languages. Very small changes to the languages we are used to are needed, and the data parallel style of thinking and of expressing things leaves us with programs which are easy to read and effective to execute. In general, there is also no problem with efficiency on various sizes of the data sets, since the concept of virtual processors takes care of the potential problems in an elegant way.

7. Application Examples

7.1 Image processing examples

With a suitable architecture that utilizes a high degree of parallelism, image processing tasks with real-time requirements may be solved even with rather small-sized systems. We will demonstrate this by two examples:

A PASIC example

An edge detection task typically used as a preprocessing step in pattern recognition is described as a demonstration of the capabilities of PASIC by Chen et al. (1990).

The prototype PASIC system, as described above, works with an image size of 128 x 128 pixels, each with eight bit resolution. Recall that the processors are arranged as a linear array, thus processing one line of the image at a time.

The edge detection algorithm is as follows:

1. *Median filtering and smoothing*
Median filtering is done in two steps, first 1 x 3 vertical neighborhood, then 3 x 1 horizontal neighborhood on the result of the previous. Smoothing is performed by adding two horizontal neighbors followed by adding two vertical neighbors of the result.

2. *Gradient calculation*
The gradients in the x- and y-directions are computed using the Sobel operator (see figure SIMD-18a). The gradient magnitude is approximated by the sum of the abso-

lute values of the two gradients.

3. *Thresholding and thinning*
A binary picture is created by thresholding. This picture now shows the edges. By an iterative thinning process in four steps the widths of the edges are reduced to one pixel (if originally not wider than five pixels).

1	0	-1
2	0	-2
1	0	-1

1	2	1
0	0	0
-1	-2	-1

(a)

0. Input	8
1. Median and smoothing	318
2. Gradient computation	221
3. Thresholding and thinning	201
4. Output	17
TOTAL:	765 cycles

One cycle is 50 ns.
765 cycles per row.
128 rows.
Execution time: 4.78 ms

(b)

(a) Sobel operator (b) Execution times for the different steps of edge detection

Figure SIMD-18

The execution times of the different steps, assuming a clock rate of 20 MHz, are shown in figure SIMD-18b. The resulting total execution time of less than 5 ms allows for a frame rate of more than 200 frames per second on this one-chip system! If the chip (and image size) is scaled up, the processing time grows linearly, so that for a 512 PE system working on 512 x 512 images, the maximum frame rate would be 50 frames per second.

It should be noted that, since the PEs are bit-serial, the execution time for the major part of this algorithm grows linearly also with the number of bits per pixel. In many applications, fewer bits than eight may be used with good result.

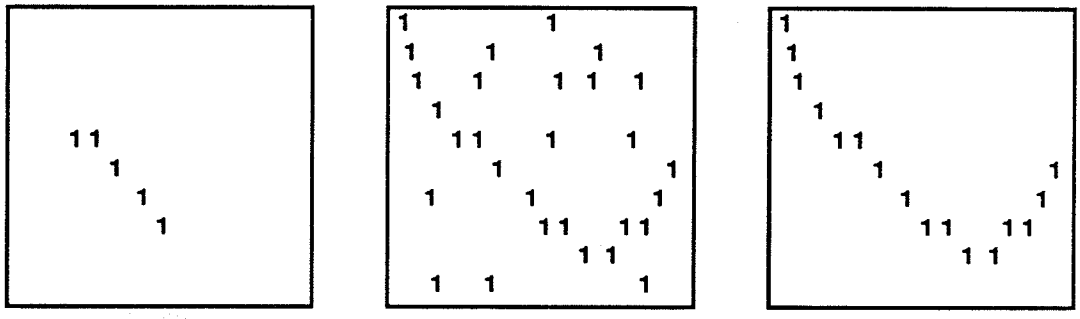
A LUCAS example

An edge detection task run on LUCAS, using "tracking" to get true edges, was described by Svensson (1983). Like the above algorithm, it may be performed in real-time. In this particular algorithm no smoothing of the image is done before the edge detection. The gra-

dient image, derived in a way similar to the one described above, is thresholded at two different levels. The low threshold gives *all potential* edge points, while the high threshold gives only the strongest points, i.e. the *safe* edge points. Now the safe points are propagated iteratively to connected pixels marked in the image of potential points. The procedure is illustrated in figure SIMD-19 and an example of a run is given in figure SIMD-20.

The computation time on a 128 by 128 pixel image with 8-bit grey-scale is 1.4 ms using a 20 MHz clock. The final tracking phase is extremely efficient on this kind of architecture and represents only 14 percent of the total time.

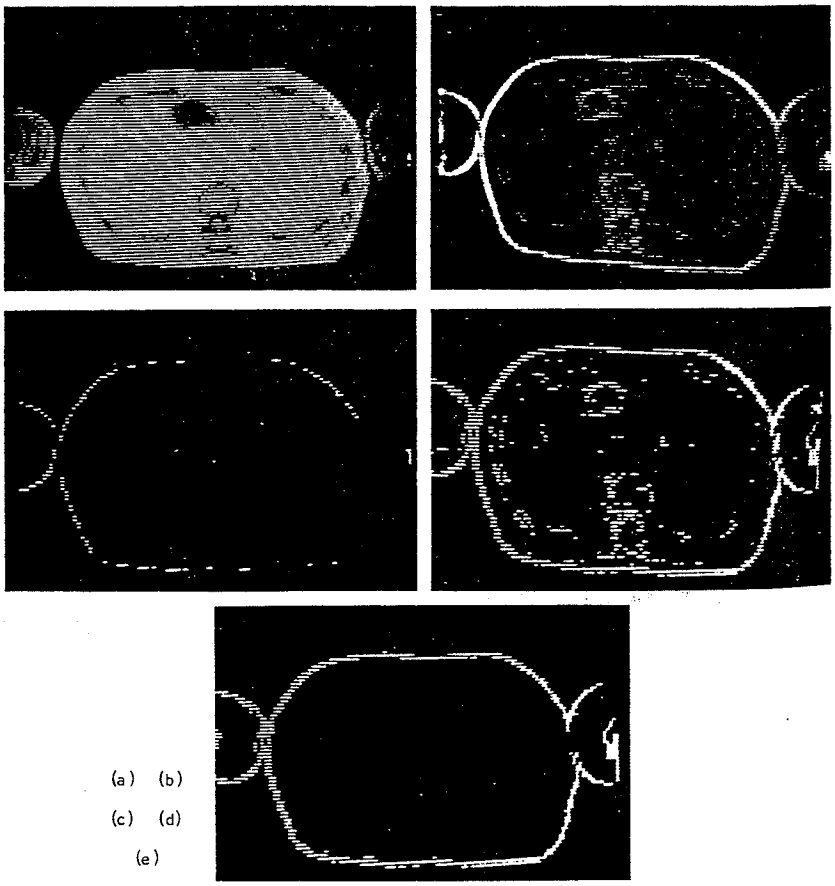
As in the PASIC case, scaling up the system and the images will result in linearly increasing computation times.



(a) (b) (c)

Tracking process. Result of thresholding at high level (a) and at low level (b). Result of tracking the '1's of (a) in (b) is shown in (c)

Figure SIMD-19



Edge detection of a magnetic resonance image by tracking.
From top left to bottom: (a) original image,
(b) gradient image derived by Roberts' cross difference operator,
(c) result of thresholding image (b) at level 160,
(d) result of thresholding image (b) at level 64,
(e) result of tracking the points in image (c) along the points in image (d).

Figure SIMD-20

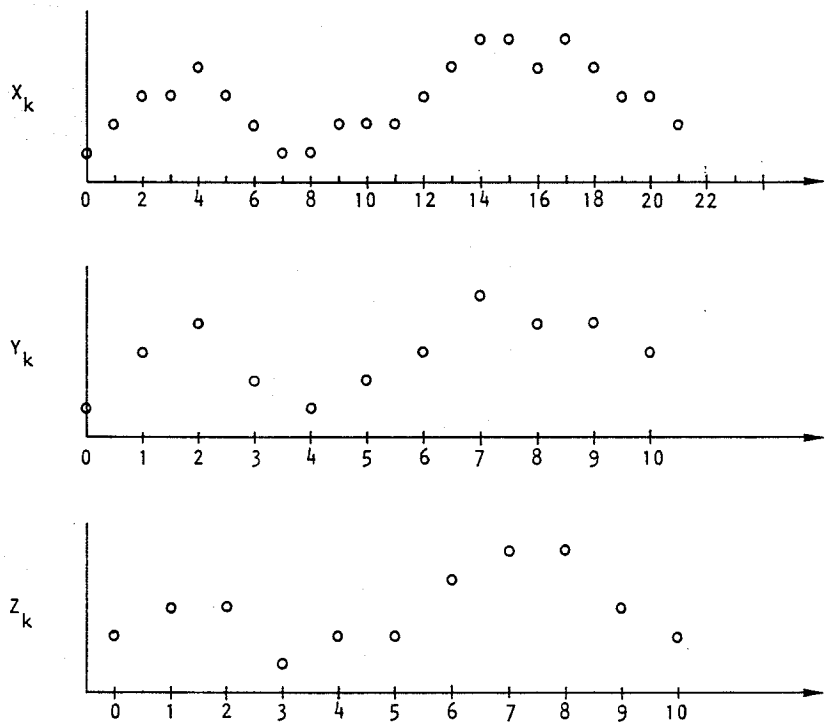
7.2 Signal processing examples

The Fast Fourier Transform (FFT) algorithm is the basis for most signal processing applications. FFT is a method for efficiently computing the Discrete Fourier Transform (DFT) of a time series (discrete data samples). A straightforward calculation of the DFT on a sequential computer takes $O(N^2)$ time, where N is the number of samples, whereas only $O(N\log_2 N)$ time is needed when the FFT method is used. The algorithm is well suited for parallel computation. Using N processing elements, the processing time can be reduced to $O(\log_2 N)$.

The FFT is a clever computational technique to compute the DFT coefficients. The DFT of a time series is here obtained as a weighted combination of the DFTs of two shorter

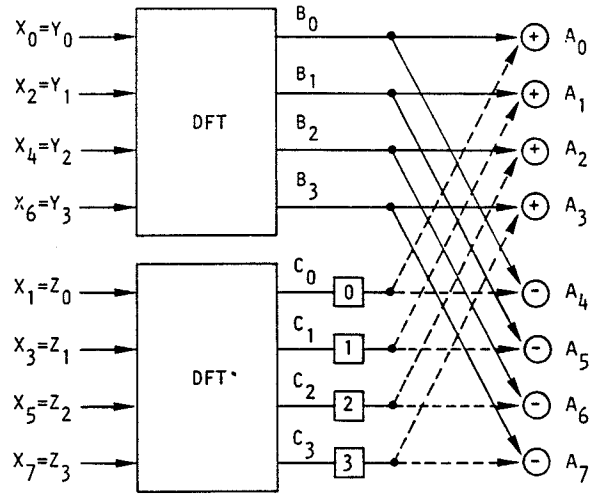
time series. These, in turn, are computed in the same way, until the DFT of a single point is needed. This is the sample point value itself.

Figure SIMD-21 shows the decomposition of a time series and figure SIMD-22 illustates the calculation. Further decomposition yields the computational flow graph of figure SIMD-23. This graph may be arranged as in figure SIMD-24, showing that the perfect shuffle-exchange pattern is ideally suited for the computation (cf. figure SIMD-6). A binary cube pattern also solves all communication problems.



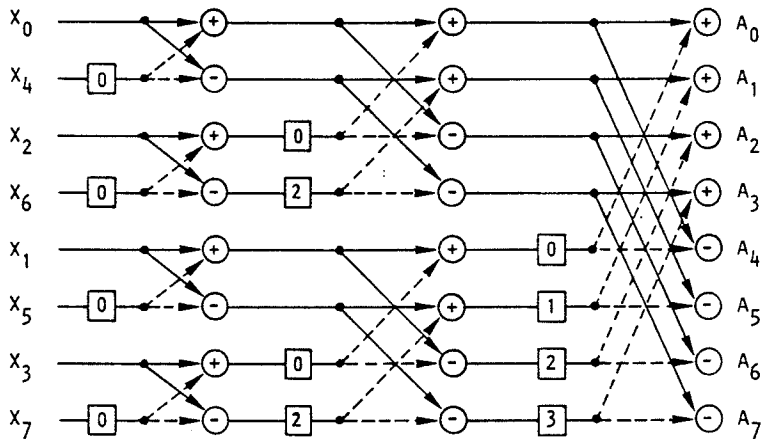
Decomposition of a time series into two half as long series

Figure SIMD-21



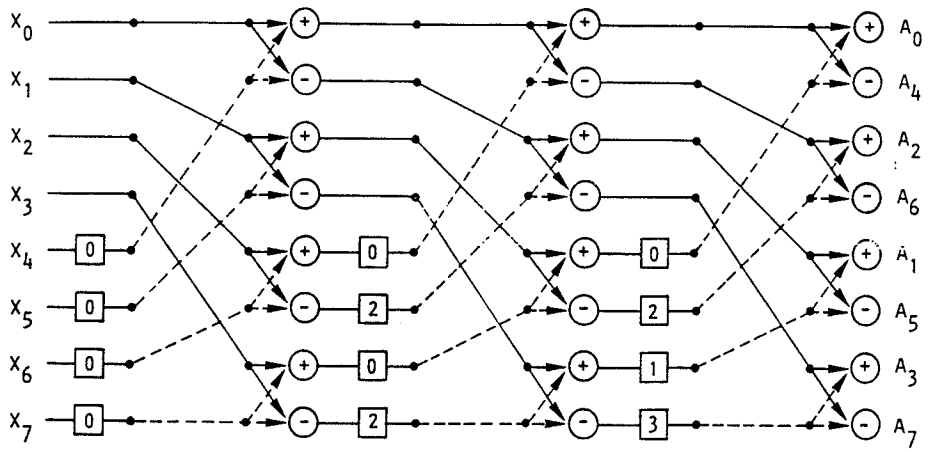
Signal flow graph illustrating how calculation of an 8-point DFT can be reduced to the calculation of two 4-point DFTs. A number within a square represents multiplication by $e^{-2\pi j/N}$ raised to the number. In the lower half, the value arriving by the dotted line is subtracted from the value arriving by the solid line. In the upper half the two values are added.

Figure SIMD-22



Calculation of an 8-point DFT using the FFT algorithm

Figure SIMD-23



Adaptation of the FFT algorithm to the perfect shuffle-exchange interconnection structure

Figure SIMD-24

FFT on a perfect shuffle connected computer (LUCAS) is described in (Fernström et al., 1986). Assuming a 10 MHz clock, the execution time for a 256-point FFT with 16-bit data on a 128 PE array is 0.33 ms. Bit-serial multipliers are utilized.

By performing first row and then column FFTs independently on a matrix of input data, we obtain a two-dimensional FFT. A transform of a 256 x 256 image requires 512 one-dimensional transforms (256 in each direction), thus takes 170 ms. Half of the transforms are computed entirely within the PEs, all at the same time.

Fast Fourier Transforms may be computed also with other array topologies. On the linear array organized PICAP3 a transposition of the array has to be made between the transformations in the x- and y-directions, allowing all 1D transformations to be computed within the PEs. Lindskog (1989) reports a calculated execution time of 93.5 ms using a 32 PE array on a 512 x 512 complex data image. On the coarse-grained PICAP3 the transposition phase takes about 20 percent of the total time. As described above, PICAP3 was designed to have very powerful floating point units as PEs.

7.3 A learning network example

The error back-propagation scheme for training a multilayer neural network was briefly described earlier in this chapter. The computations involve mainly matrix-by-vector multi-

plications, where the matrices contain the connection weights and the vectors contain activation values or error values. Such a multiplication contains N^2 scalar multiplications and N computations of sums of N numbers.

The fastest possible way to compute this is to perform all N^2 multiplications in parallel, which requires N^2 PEs and unit time, and then form the sums by using trees of adders. The addition requires N(N-1) adders and O(logN) time. This is, however, an unrealistic method depending on both the number of PEs required and the communication problems caused. Instead, it is practical to take the approach of having as many PEs as neurons in a layer, N, and storing the connection weights in matrices, sized N by N, one for each layer. The PE with index j has access to row j of the matrix by accessing its own memory word.

The computations performed on an array of bit-serial processors are described in (Svensson and Nordström, 1990). The calculation times for different network sizes are calculated. Bit-serial multipliers of the type presented in figure SIMD-11 are used. Some results from the study are presented in table SIMD-1. In the table, *b* is the data length and *N* is the number of neurons per layer.

		N		
		256	1024	4096
b	8	2.4	9.9	40.6
	12	3.6	14.4	58.6
	16	4.7	18.9	76.6

(a)

		N		
		256	1024	4096
b	8	1.0	4.2	17.6
	12	1.4	5.8	24.2
	16	1.8	7.5	30.7

(b)

a)Training time per layer (ms). b) Recall time per layer (ms).
10 MHz clock frequency is assumed.

Table 1

A common measure of the performance of neural net hardware is the number of "CPS" (Connections per second). In a net with N neurons per layer, N^2 connections are used and/or updated in each layer. The MegaCPS figures for bit-serial array processors of different sizes are given in Table 2.

		N		
		256	1024	4096
b	8	27	106	413
	12	18	73	286
	16	14	55	219

Training

		N		
		256	1024	4096
b	8	66	250	953
	12	47	180	694
	16	36	140	546

Recall

Number of MegaCPS (Million Connections Per Second) for different network sizes and data precisions.

Table 2

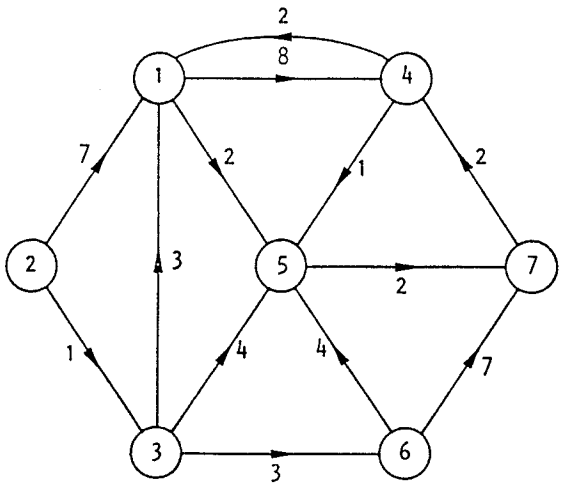
If external RAM is used, 64 PEs of the complexity we discuss can easily be integrated on one VLSI chip. A 1024 PE array will have 16 such chips, each with approximately 100 pins. Memory can be implemented using chips with 64k x 4 bits, giving a memory chip count of 256. With appropriate mounting technology such a network may be implemented on one board. It would e.g. run a four-layered feedforward network with 1024 neurons per layer at the speed of 29 training examples or 81 recall examples per second.

7.4 A graph problem.

Problems that can be identified as graph theoretic show up in diverse areas, e.g. traffic planning and network analysis. A common task is to find the shortest path between any two vertices of a graph. The connection between two vertices may be uni-directional or bi-directional. In the first case the graph is called a *directed graph.* Also, a cost (or path length) may be associated with each path. Such graphs are called *weighted.*

Solutions of problems of this kind often take the form of searching large trees or updating matrices. Opportunities to exploit data parallelism are rich. As an example we will consider the problem of finding *the shortest path between all pairs of vertices in a weighted, directed graph.*

In a weighted, directed graph the paths between the vertices are uni-directional and there is a length associated with each path. Figure SIMD-25 shows an example of such a graph. The graph may also be described in the form of a matrix. The absence of a direct path between a pair of vertices is marked in the matrix with "infinite" (if).



	1	2	3	4	5	6	7
1	0	if	if	8	2	if	if
2	7	0	1	if	if	if	if
3	3	if	0	if	4	3	if
4	2	if	if	0	1	if	if
5	if	if	if	if	0	if	2
6	if	if	if	if	4	0	7
7	if	if	if	2	if	if	0

A weighted, directed graph and its distance matrix

Figure SIMD-25

To solve the problem on a SIMD array we will parallelize an algorithm due to Floyd (1962) which is considered as one of the two most efficient algorithm for sequential computers. It is well suited for parallel implementation. On sequential computers a computation time proportional to n^3 is required, where n is the number of vertices. On a parallel computer with n PEs it should be possible to perform the algorithm in a time proportional to n^2.

The algorithm works as follows. Starting with the original n by n matrix D of direct distances, n different matrices D_1, D_2,...,D_n are constructed sequentially. Matrix D_k is obtained from matrix D_{k-1} by inserting vertex k in a path wherever this results in a shorter path.

On a parallel computer with n PEs an entire column of the matrix may be updated simultaneously. In the k:th iteration, column p of D_k is obtained in the following way (using Pascal/L notation for matrix elements):

$$D_k(,p) := \min [\, D_{k-1}(,p) \,,\, D_{k-1}(,k) + D_{k-1}(k,p)]$$

A Pascal/L program for the entire algorithm reads as follows:

```
Program FLOYD;
const noofvertices = 128;
var   Dmatrix:   parallel array [1..noofvertices,1..noofvertices] of integer(8);
      k, p :     integer;
begin
   for k:=1 to noofvertices do
   begin
      for p:=1 to noofvertices do
      begin
         where (Dmatrix[*,k]+Dmatrix[k,p]) < Dmatrix[*,p] do
         Dmatrix[*,p]:= Dmatrix[*,k]+Dmatrix[k,p];
      end;
   end;
end.
```

It is easily seen that the execution time of the program is proportional to n^2. The task that is performed n^2 times is an 'add fields' instruction followed by a 'mark field greater than field' instruction and a selector masked 'move field'. These are all performed in constant time.

The algorithm requires a representation for an infinite value. A number that is a little smaller than half the greatest number that is possible to represent in the given field length may be chosen. In the worst case, two such numbers are added, which can be done without giving overflow.

We have shown how a good sequential algorithm may be ported in a straightforward way to a parallel environment. For other graph theoretical problems, however, new algorithms have to be invented. Svensson (1983) gives a couple of examples.

8. Massively Parallel SIMD Arrays in Embedded Systems, MIMSIMD.

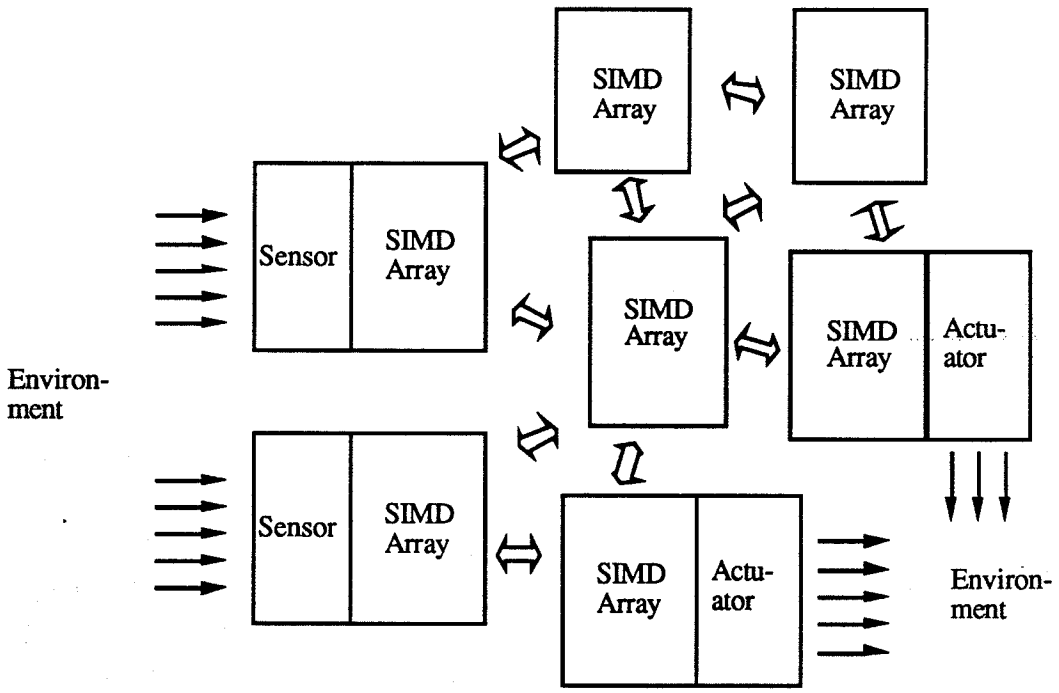
In the same way as we have seen conventional computers evolve from big machines residing in a few computer centers, via minicomputers and desktop computers, to embedded, yet powerful systems without degrading the computing power, we will see highly parallel computers used in workstations or integrated into industrial systems solving real-time tasks.

An example of an effort to implement a miniaturized massively parallel computer is the *BLITZEN* project in the Research Triangle of North Carolina (Blevins et al. 1988). A highly integrated chip has been designed comprising as many as 128 bit-serial processing elements, each with 1kbit of on-chip memory. A board of some tens, maybe up to a

hundred, of those chips could easily be plugged into the backplane of a workstation or used in industrial environments where small size and large computing power is needed.

A likely development is that such massively parallel systems will be integrated with sensors and actuators. Such integrations are necessary in order to cope with the high input-output data rates. We have already demonstrated a couple of such examples, the LAPP and the PASIC. There will certainly be more.

Along that line of development we will see systems of many cooperating SIMD arrays, each integrated with some sensory or motor function, or serving some other specialized task in the system, see figure SIMD-26. The different arrays are controlled by their own streams of instructions, thus the total system may be characterized as MIMSIMD (Multiple Instruction streams, Multiple SIMD). The cooperation between the modules will typically require very high bandwidth inter-array communication.



A MIMSIMD system integrated with sensors and actuators

Figure SIMD-26

These ideas conform with Arbib's (1989) "sixth generation computers" concept:

> "The study of animal and human brain suggests overall architectural principles for "sixth generation computers." Each such machine will comprise a network of more specialized devices, with many of these devices structured as highly parallel arrays of interactive, neuron-like, possibly adaptive, components."

According to Arbib, those sixth generation systems will be characterized by:

> (a) *Cooperative computation:* the computer will be a heterogeneous network of special-purpose and general-purpose subsystems. Some of the sybsystems (such as the front ends for perceptual processors (see (b)), and devices for matrix manipulation) will be highly parallel;
> (b) *Perceptual robotics:* increasingly, computers will have intelligent perceptual and motor interfaces with the surrounding world; and
> (c) *Learning:* Many of the subsystems will be implemented as adaptive "neural style" networks.

9. Implementation Considerations

The development of Very Large Scale Integration technology (VLSI) has had an important influence on the design and implementation of processor arrays, and has on the whole been a necessary prerequisite for producing arrays of general usefulness.

As more and more processors are implemented on the same chip, the number of connections to other chips (at least in nearest-neighbor connected systems) also increases, even if at a slower rate. This may cause practical connection problems between chips. Even worse problems, however, arise in the connections between processors and memory if the memory is put in separate chips. Having local addressing, which we have found e.g. in PICAP3, is totally impossible if several processors are implemented on the same chip. Thus, we will probably see a development towards integrated memory and processor chips. The combination of the two problems points to the solution of implementing a whole array on a wafer, i.e. Wafer Scale Integration (WSI).

WSI, however, raises another problem. It is acceptable for wafers that are intended for VLSI fabrication to incorporate defects. They lead to the failure of single dies only, and if they are not too many they can be tolerated. However, if a whole, continuous array is on the same wafer, we require all processors in the array to function correctly. Defects can not be tolerated at all.

To cope with this problem, techniques are being developed to switch out faulty processors and make a continuous array of the correct ones, or to include redundancy in the array so that the chance of getting a functioning array increases.

Without getting into any details at all, we conclude this section by noting that much interest is directed towards possible future *optical* realizations of processor array functions, especially solutions to the communications problem. Beams of light may cross each other without interfering, making it possible to communicate between millions of processors at once. The first practical systems utilizing optical technology will no doubt be hybrid electronic/optical systems.

10. References

Arbib, M.A. (1989). Schemas and Neural Networks for Sixth Generation Computing, *Journal of Parallel and Distributed Computing*, vol. 6, pp. 185-216, 1989.

Barnes, G.H., R.M. Brown, M. Kato, D.J. Kuck, D.L. Slotnick, and R.A. Stokes (1968). The ILLIAC IV Computer, *IEEE Transactions on Computers*, vol C-17, no.8, pp. 746-757.

Batcher, K.E. (1974). STARAN Parallel Processor System Hardware, *Proc. AFIPS NCC*, 1974, pp. 405-410.

Batcher, K.E. (1976). The FLIP Network in STARAN, *Proc. 1976 Int. Conf. on Parallel Processing*, Waldenwoods, Mich.

Batcher, K.E. (1977). The Multidimensional Access Memory in STARAN, *IEEE Trans. on Computers*, vol. C-26, no. 2, February 1977, pp. 174-177.

Batcher, K.E. (1980). Design of a Massively Parallel Processor, *IEEE Transactions on Computers*, vol C-29, pp. 836-840.

Blevins, D.W., E.W. Davis, R.A. Heaton, and J.H. Reif (1988). BLITZEN: A Highly Integrated Massively Parallel Machine, *Proc. Frontiers of Massively Parallel Computing*, Fairfax, Virginia, Oct. 1988.

Brown, J.R., Garber, M.M. and Vanable, S.F. (1988). Artificial Neural Network on a SIMD Architecture, *Proceedings of the 2nd Symposium on the Frontiers of Massively Parallel Computation*, pp. 43-47, Fairfax, Virginia, 1988.

Chen, K., A. Åström, T. Ahl, and P.E. Danielsson (1990). PASIC: A Smart Sensor for Computer Vision, *Proceedings of* 10*th International Conference on Pattern Recognition*, Atlantic City NJ, June 16-21, 1990.

Fernström, C. (1982). Programming Techniques on the LUCAS Associative Array Computer", *Proceedings of the* 1982 *International Conference on Parallel Processing*, Bellaire, Michigan, Aug. 1982.

Fernström, C., I. Kruzela, and B. Svensson (1986). *LUCAS Associative Array Processor – Design, Programming and Application Studies*, Lecture Notes in Computer Science, vol. 216, Springer Verlag, Heidelberg.

Fisher, A.L. and P.T. Highnam (1985). Real-time Image Processing on Scan Line Array Processors, *IEEE Computer Society Workshop on Computer Architecture for Pattern Analysis and Image Database Management*, Miami Beach, Florida, 1985, pp.484-489.

Floyd, R.W. (1962). Algorithm 97: shortest path, *Communications of the ACM*, vol. 5, p. 345.

Forchheimer, R. and A. Ödmark (1983). A Single Chip Linear Array Picture Processor, *Proc. of 3rd Scandinavian Conference on Image Analysis*, Copenhagen, July 1983, pp. 320-325.

Hillis, W.D. and G.L. Steele (1986). Data Parallel Algorithms, *Communications of the ACM*, vol. 29, no. 12, December 1986, pp. 1170-1183.

Hockney, R.W. and C.R. Jesshope (1988). *Parallel Computers 2: architecture, programming and algorithms*, Adam Hilger, Bristol, 1988.

Hunt, D.J. (1989). AMT DAP – a Processor Array in a Workstation Environment, *Computer Systems Science and Engineering*, vol.4, no.2, April 1989, pp.107-114.

Lindskog, B. (1988). PICAP3 – An SIMD Architecture for Multi-Dimensional Signal Processing, PhD Thesis, Linköping Studies in Science and Technology, No. 207, Linköping University, Sweden.

Lindskog, B. (1989). PICAP3. A Linear SIMD Array with Floating-Point Arithmetic, Report LiTH-ISY-I-0971, Linköping University, Sweden.

Nordström, T. (1990). On the Simulation of Neural Networks on SIMD Arrays, *Proceedings of 1st Nordic Workshop on Parallel Algorithms and Architectures in Vision and Image Processing*, Linköping, Sweden, March 7-9, 1990 (extended abstract).

Potter, J.L. (editor)(1985).*The Massively Parallel Processor*, The MIT Press, Cambridge, Massachusetts.

Reeves, A.P. (1984). Parallel Pascal: An Extended Pascal for Parallel Computers, *Journal of Parallel and Distributed Computing*, vol. 1, 1984, pp. 64-80.

Reddaway, S. (1979). The DAP Approach. *Infotech State of the Art Report on Supercomputers*, Infotech Intl. Ltd., Maidenhead, Berks., UK.

Schmitt, L.A. and S.S.Wilson (1988). The AIS-5000 Parallel Processor, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 10, no.3, pp. 320-330.

Segerström, J. (1990). Konstruktion av Processormodul för PICAP3 – Problem och Möjligheter, Licentiate Thesis, Linköping Studies in Science and Technology, Thesis No. 207, Linköping University, Sweden. (In Swedish).

Singer, A. (1990). Implementations of Artificial Neural Networks on the Connection Machine, Technical Report RL90-2, Thinking Machines Corporation, Cambridge, Massachusetts, January 1990.

Svensson, B. (1983). LUCAS Processor Array: Design and Applications, PhD thesis, Department of Computer Engineering, University of Lund, Sweden, April 1983.

Svensson B. (1989) Implementation and application of a software configurable massively parallel computer, *Second Swedish Workshop on Computer Systems Architecture,* Bålsta, Sweden, August 1989. Available as Research Report CDv-8903 from Halmstad University College.

Svensson, B. and Nordström, T. (1990). Execution of Neural Network Algorithms on an Array of Bit-Serial Processors, *Proceedings of 10th International Conference on Pattern Recognition*, Atlantic City NJ, June 16-21, 1990.

Thinking Machines Corporation (1989). *Connection Machine Model CM-2 Technical Summary*. TMC, Cambridge, Massachusetts.

Thinking Machines Corporation (1990).*Getting Started in CM Fortran*, TMC, Cambridge, Massachusetts.

Wilson S.S. (1988) One dimensional SIMD architectures – the AIS-5000, in *Multicomputer Vision*, S.Levialdi, Ed., Academic Press, London, pp.131-149.