2. Parallel computer structures, a summary

This chapter gives a very short summary of basic parallel archirectures. The main goal is not to give a detailed explanation (you can find a detailed material developed in the framework of activity 4. of this Tempus cooperation), instead to mention the arcitectures that are important in the field of parallel digital signal processing.

2.1 Pipeline Structure

Pipeline processing is defined as that type of processing, where execution of a complete process is *decomposed* into many subprocesses. These subprocesses are performed by separate processing elements (PE) connected to each other in a chain as it is illustrated by figure 2.1.

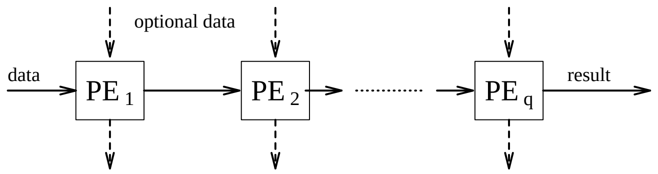

Figure 2.1: Generalized pipeline structure

The input of this structure is the input of the first PE, each PE is waiting for input to be generated by the preceeding PE and passes processed data to the next PE.

This method is well known from general computer instruction execution, where it is quite common to use a pipeline structure. In this case separate PE-s serve for instruction fetch, instruction decoding, operand fetch and instruction executing respectively. In some cases data not only accepted/passed from the previous PE but also through I/O port(s).

A signal processing example is the pipeline realizations of an FIR filter as it can be seen on figures 2.3 and 2.4. Figure 2.2 shows the block diagram of an FIR filter. It is clear that at each step $(q-1)$ multiplications have to be done, and these multiplications can be done in parallel at the same time. On the other hand the addition of the elements have to be done in "one step". To get a pipeline structure the additions can be done iteratively as

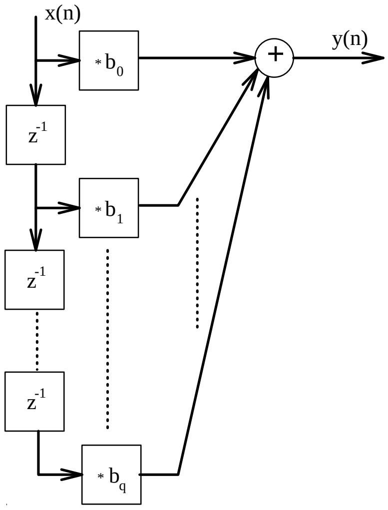

Figure 2.2: The blokkdiagramm of an FIR filter

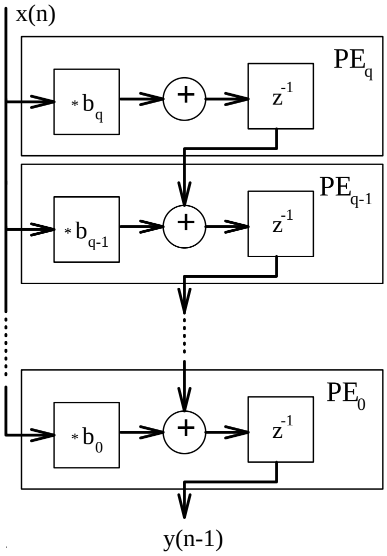

Figure 2.3: Pipeline realization of an FIR filter with global communication

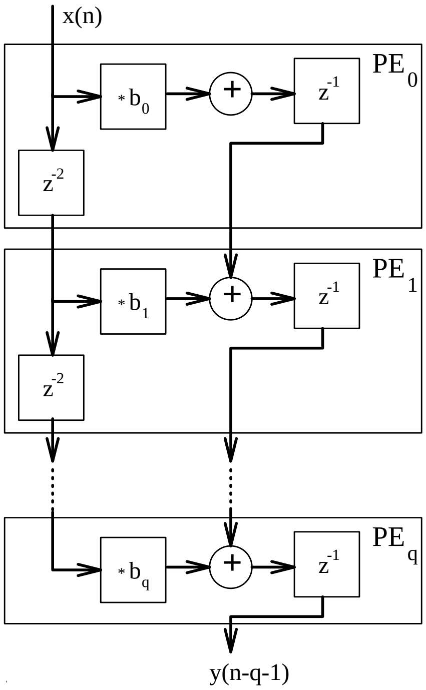

Figure 2.4: Pipeline realization of an FIR filter with local communication

figure 2.3 shows. In this structure there is a *global communication line* which transfers the actual input $x(n)$ to each PE. In most of the cases realization of global communication line is problematic. Using the structure shown on figure 2.4 the global communication line is eliminated and there are only *local connections* between the PE-s. As it can be seen in this case there is a $(q+1)$ time shift between the input $x(n)$ and the resulted $y(n)$ data streams, and more memory elements are needed to realize this structure.

Another example can be the matrix-vector multiplication. A possible realization is shown on figure 2.5 [Kung88]. This realization needs q input ports. This fact represents very well the usual problem of parallel processing: efficient parallel processing requires in most of the cases extreamly intensive I/O activities. Similar solution can be used here as used on figure 2.4 to localize the communication, so a_{ij} matrix elements can be used as a sequential extra input for PE_1 and than propagate them through PE-s. It will need much more time (increasing with q^2), extra memory and extra management of data propagation.

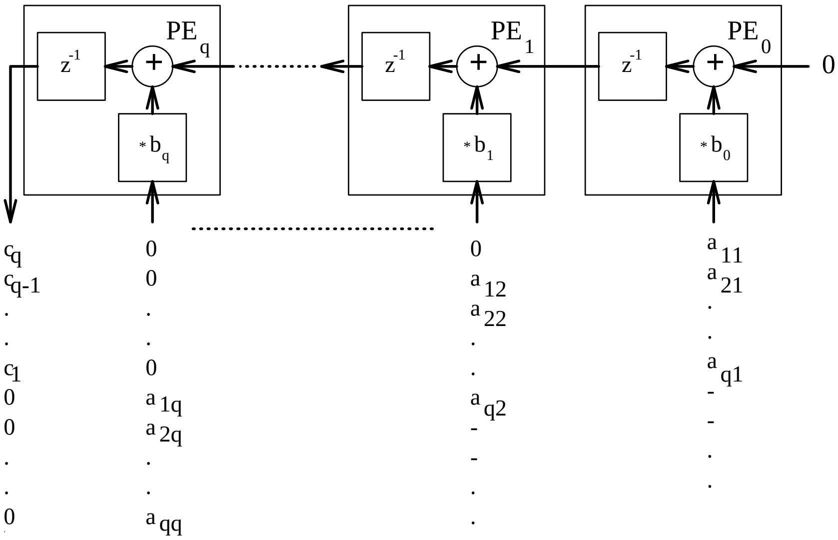

Figure 2.5: Pipeline realization of matrix-vector multiplication

2.2 Single Istruction Multiple Data Stream Structure (SIMD)

Figure 2.6 shows the SIMD architecture [Kung88]. Each PE performs the instruction received from the Control Unit (CU) at the same time. PE-s use their own local memory and a memory shared between each other. Various implementation of memory sharing are possible, but the main idea is always the same:

- The PE-s communicate with each other using this sheared memory.

- The CU controls the I/O of the whole system to/from sheared memory.

This processor structure is used mainly in big supercomputers, and is not suited very well to signal and image processing applications due to the big overheads.

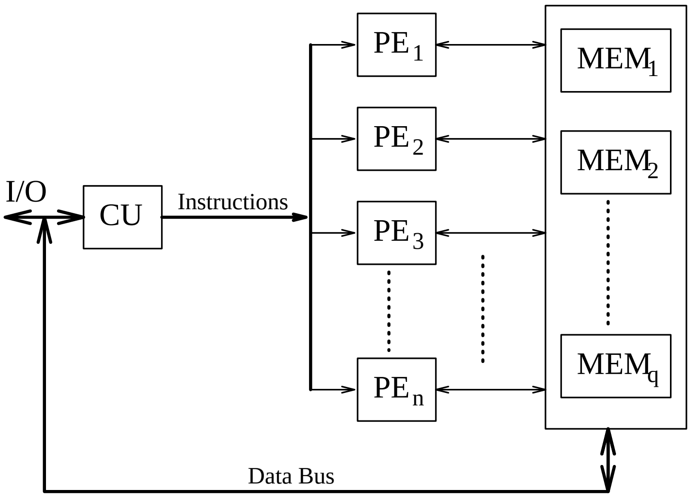

Figure 2.6: The SIMD structure

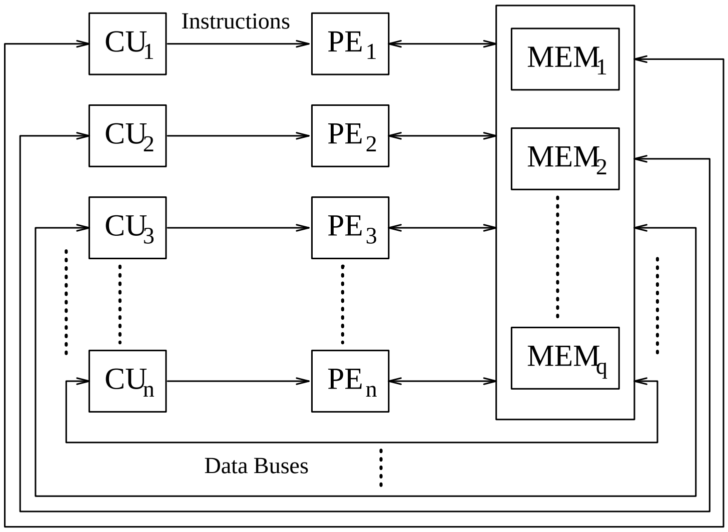

Figure 2.7: The MIMD structure

2.3 Multiple Istruction Multiple Data Stream Structure (MIMD)

Figure 2.7 shows the MIMD architecture [Kung88]. The Main idea is that every PE has it's own instruction flow got from it's own Control Unit (CU). PE-s use their own local memory and a memory shared between each other. The shared memory has the same function as at the SIMD architecture, namely to serve as communication platform between PE-CU pairs and system I/O.

This processor structure is also used mainly in big supercomputers, and is not suited very well to signal and image processing applications.

2.4 Vector Processors

The basic idea of a vector processor is to combine two vectors, element by element, to produce the output vector, e.g. to form the addition of two vectors. A very simple way to implement this operation is to use a pipeline arithmetic unit. Two streams of data supplied to the arithmetic unit carry the streams for the two input vectors. These data streams come from a memory which supplies one element to each input streams on every clock cycle. The arithmetic unit produces one output value during each clock cycle. The difficulty is the design of the memory system to sustain a continuous flow of data from memory to the arithmetic unit and the return flow of data from the arithmetic unit to the memory. The majority of the architectural tricks used in vector processors are devoted to sustaining that flow of data and to scheduling sequences of operations to reduce the flow requirements. In our vector addition problem we assume a basic one-cycle rate for the delivery of operands, production of results, and restoring the result data into memory. This calls for a memory system that can read two operands and write one operand in a single cycle. Conventional memory systems can perform at most one read or one write operation per cycle, so the memory system we require for our task has at least three time the bandwidth of a conventional memory system. To meet the requirements two basic approaches have emerged in commercial vector machines: (*) build the necessary bandwidth in main memory by using several independent memory modules to support concurrent access to independent data, (**) build an intermediate high-speed memory with the necessary bandwidth and provide a means for high-speed transfer between high-speed memory and main memory. The first approach acknowledges that if one memory module can access at most one datum per access cycle, than to access N independent data in one access cycle requires N independent memory modules. The second approach produces higher bandwidth by shortening the access cucle in a small memory, but the small memory is loaded from a large memory, and the large memory can still be the bottleneck in the system in spite of the high bandwidth of the small memory. To make best use of a small high-speed memory, we should make multiple use of operands transferred to this memory. In this way the net demand by the processor on the large memory is reduced and the bandwidth of the large memory need not be as large as the peak bandwidth required by the processor.

2.5 Array Processors

The term *Array Processors* were coming with the appearance of VLSI and Wafer Scale Integration technonogy, when extremly large number of processing elements are integrated into a single unit.

With the advantage of today's technology it is possible to match hardware design of the underlying processor, and algorithm design in a feedback way, that is the design of one influences the design of the other matching to each other.

There are mainly two type of array processors the *systolic* and the *wavefront* arrays. The basic difference between them lies in the organisation of communications.

2.5.1 Systolic Array Processors

A systolic array processor system is a network of processors which *rhytmically compute and pass data through the system.* Every processor transfers the input and output data *regularly at the same time*, than makes a short computation. These activities follow each other rhytmically like the periodes of the human heart [Kung88]. This rithmical processing is timed by a *global clock.*

The array consists of *modular processing elements with homogenous interconnections.* It means that the actual size of the array designed for a given algorithm depends on the actual number of parameters/datas to be used in that particular realization. Moreover, the computing network may be theoretically extended indefinitly.

The array manifests a locally-communicative interconnection network, i.e. the structure is *partial local.* Each PE has connected to some of it's neighbour.

For every PE-s an output data computed from a given input can be transfered *at least one time unit delayed* after the last input data arrival.

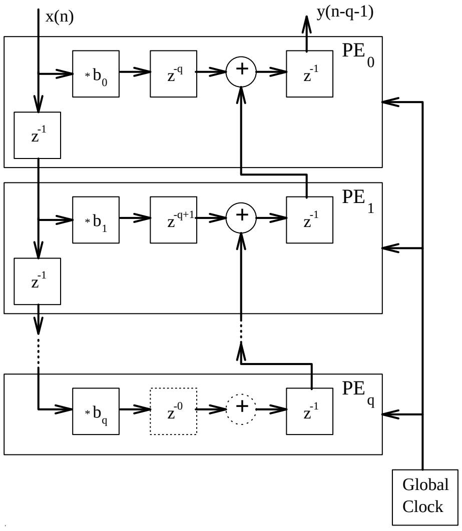

Figure 2.8: FIR filter realization with systolic array

To demonstrate a simple systolic array the figure 2.4 can be recalled which is a simple linear systolic array. Although this simple pipeline structure discussed in section 2.1 suppose unidirection data flow, in systolic arrays the connections between two PE-s allows bidirectional data flow, with the restrictions of time delay mentioned above. To demonstrate this bidirectional data flow the FIR filter structure on figure 2.4 (page 3) can be modified as it is shown on figure 2.8. To emphasize the existence of the global clock of the systolic array it is explicitly drawn on the figure 2.8. Unfortunately this structure is not more effective than the previous one due to the requirements of related data.

Another example is the bouble sorter algorithm showed on figure 2.9. There are true bidirectional data movements between the processor elements. Each PE executes the same sequential program with 2 steps, one step for each clock pulse. The neighbouring PE-s execute the program with a phase shift so while the i-th PE executes the MIN search, the $(i+1)$-th the MAX search. The initial values are the values to be sorted using '$-\infty$', and '$+\infty$' (the smallest and the largest numbers) at the left and the right sides respectively to close the sequence. After n (number of elements to sort) steps the result is in the memory of the PE-s sorted increasing order from left to right.

Figure 2.10 shows a more representative example, the band matrix mul-

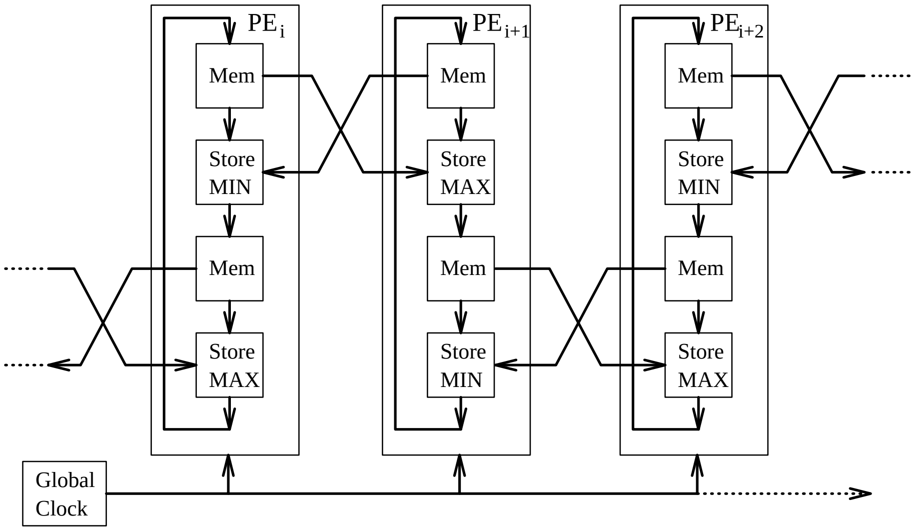

Figure 2.9: Bouble sorting realization with systolic array

tiplication algorithm [Kung88]:

$$\underline{\underline{C}} = \underline{\underline{AB}} \quad \text{with } '\text{bandwidth}' \ 4 \tag{2.1}$$

This structure is a real two dimensional solution of the problem. The detailed description of this algorithm will be discussed later.

2.5.2 Wavefront Array Processors (WAP)

Wavefront arrays are *data driven self timed* arrays of processors. The instructions of each processor is data driven, the *triggering* of instructions depends upon the *availability of operands and resources* required. Unrelated instructions can be executed concurrently without interference.

The major difference (and can be said - the only one) distinguishing a wavefront array from systolic array is the *data driven property*, so there is no strict timing requirement and global clock. In wavefront arrays the requirement of correct timing is replaced by *correct sequencing.*

Regularity, modularity and local interconnections are also major requirements of wavefront arrays. As it is described in section 2.5.1, each PE can be connected only to some of it's neighbours, the PE-s are making similar short programs. The computing network can be scaled according to the problem size (theoretically indefinitely).

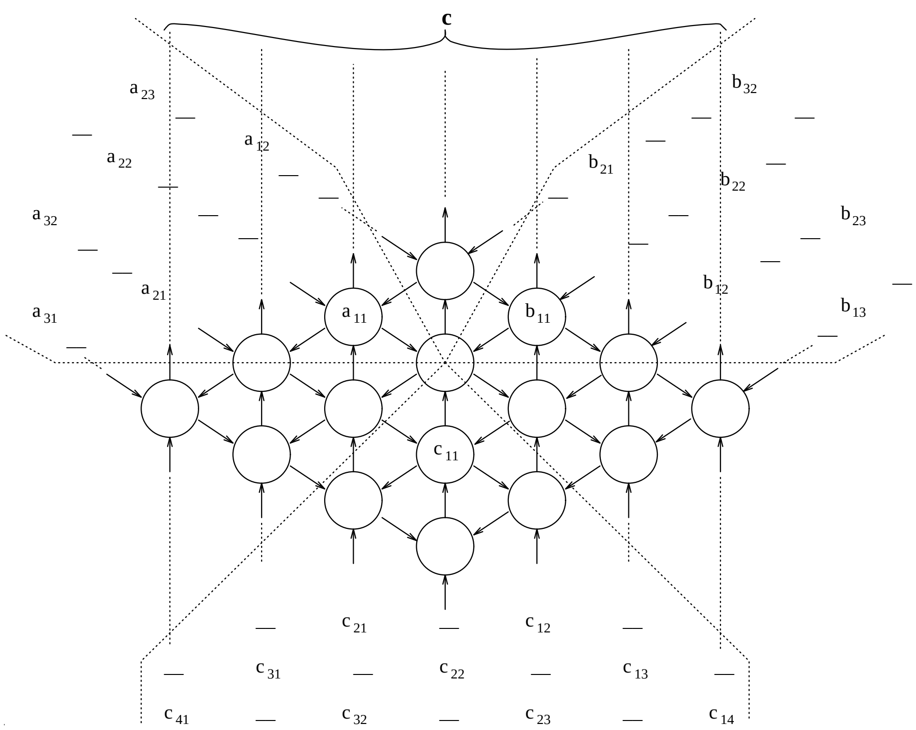

Figure 2.10: Bound matrix multiplication with systolic array

Wavefront arrays are implemented in most of the cases as a *programmable processor* array. The programmability is given by wavefront languages or using Data Flow Graphs (DFG) (see later). The program must include the following specifications:

- type of computation, what to do with the input data stream
- input data link
  - which data
  - from where
- output data link
  - which result
  - to where

Solving the localized FIR filtering problem with systolic array on figure 2.8 there were z^{-i} elements in each PE to ensure the synchronization. If the time delay between $x(n)$ input elements are not the same as the general system clock of the systolic array, a more complicated synchronization is needed. Using wavefront array it is not necessary to give an explicit delay and moreover the input rate is an unimportant question (until an upper limit).

2.5.3 Designing Considerations

There are several common considerations through the designing procedure of systolic and wavefron arrays.

From the point of view of array processing an algorithm is a set of rules for solving a problem in finite number of steps by a multiple number of interconnected processors. So the array algorithm depends on the characteristics of the PE-s as well as the interconnection strategies. In array algorithms the later has major interest.

Concurrency is the main idea for achieving high throughput. Concurrency means *decomposition* of the problem into

- *independent* subtasks can be executed by independent parallel processing element(s),
- *sequentially dependent* subtasks executable with processor elements in pipeline structure.

In the case of array processors the main goal is always to *localize* the communication with the help of *recursive* description of the algorithm to avoid the global interconnections between PE-s. So a special class of algorithms, namely recursive and locally dependent algorithms are suited well to array processing.

To measure the effectiveness of an array algorithm the well known measures of sequental programming cannot be used. In array Algorithms overheads are not uniformly distributed, fewer computation doesn't mean quicker throughput. In array algorithms the communication is expensive. First of all the global communication see for example figure 2.3, where data have to be broadcasted to every PE-s. It can be stated that nearly no way to realize global communications in array processors. They have to be transformed to sequences of local communications as in the example of FIR filter realization on figure 2.4.

To conclude, some designing aspects can be given:

- *Maximum parallelism.* The performance of an array algorithm is mainly not measured by number of computational steps. In these algorithms processing elements are relatively less expensive than communication. An algorithm will be favoured if it expresses a higher parallelism which is exploitable by computing arrays even if the algorithm needs more computation.

- *Maximum pipelinability.* The throughput rate can be linearly increased by using pipeline structure. The pipeline structure needs well structured algorithms, first of all recursive algorithms, with predictable and not very intensive data movements. Algorithms with dynamic branching dependent on data produced during the process are not well suited to pipelining.

- *Balance among computation, communication and memory.* It is critical to the effectiveness of array algorithms to have a sound balance among this three areas. It is very important to avoid *data drainings or bottlenecks*, and having a homogenous bandwidth everywhere in the array. First of all the I/O bandwidth is critical in array algorithms, and can be a hard limit for the implementation. So algorithms with sequental inputs and outputs are preferred instead of big amount of parallel input at the beginning and output at the last steps.

- *Tradeoff between computation and communication.* In some cases it is better to use an array algorithm with higher average computational needs if it offers local and data independent communication instead of global and/or data dependent interconnections.

To demonstrate these statements it is worth comparing the DFT and the FFT algorithms. The computational complexity of DFT is $\mathbf{O}(N^2)$ while the FFT is $\mathbf{O}(NlogN)$. On the other hand the FFT needs global interconnection because the 'size' of butterflies change from stage to stage while DFT enjoys simple communication needs it belongs to the 'locally recursive' class of algorithms.