Generation and Verification of Systolic Algorithms

Laura Ruff
Babes-Bolyai University Cluj
laura@cs.ubbcluj.ro

March 22, 2007

Contents

List of Figures

Chapter 1

Introduction

. . . to be written later.
Some ideas to write about:

- thinking sequentially vs. parallel programming

  Obvious advantages of parallel programming (speed, efficiency) **but** much more simple to write a sequential program.

  → **parallelisation** of sequential programs

- the systolic architecture

  Systolic architectures are considered to be a very suitable means of implementing parallel algorithms in several areas, like linear algebra, signal processing, pattern matching, dynamic programming, etc. [Pet93]. The term *systolic* was fist introduced by Kung and Leiserson [KL78] (see also [KL80],[Kun82]), designating the parallel computation where the data is systematically "pumped" from the external memory through an array of processors, just like the human blood is pumped by the heart through the vascular system.

  Systolic arrays combine pipelining and multiprocessing techniques, and are composed of a number of processing elements (**PE**), connected together by a *regular interconnection*. Each PE is performing - usually the same and very simple - computations on its input data and local memory variables, it stores data and is communicating with the neighbouring PE-s. The functioning of the PE-s is synchronised by a global clock, that results in a *regular data flow* through the system.

  The systolic model suits very good for construction of high performance special purpose computer devices.

  . . .

1.1 Overview

. . . To be written later (at the moment is just copy-paste from the beginning of the chapters)

Chapter 2 (Survey): In this chapter we provide a short survey about the state of the research done in the field of automatic generation and verification of systolic arrays, used as starting point to our research. We first present the systolic array design methods available in the literature and sum up the main ideas used. We also compare them in order to find out the advantages and weak points of the most frequently used techniques.

Concerning the available formal verification methods, they are classified according to several comparison principles and the formal framework used.

Finally we summarise the main characteristics that we find the most important with respect to our research.

Chapter 3 The Space-Time Transformation Methodology : The *space-time transformation method* is the most commonly used design technique, more exactly it is the common name for the main ideas used in several design methods. All of these techniques are based on the concept of transformations applied to the index space representing the computations. The purpose of the transformations is to determine *where* and *when* the computations will be performed. This is the reason for the term "space-time" in the name of the method.

In this chapter we give a more detailed description of this particular design technique for the sake of a more precise comparison with our novel method presented in Chap. 5.

An interesting case study is presented at the and of the chapter, that will be revisited in Sect. 5.6.4, thus it can reveal the advantages and drawbacks of the two methods. The design of the case study was actually inspired by the online array generated with our functional–based method.

Chapter 4 Mapping Systolic Arrays onto Fixed Size Architectures:

Various attempts have been made to overcome the drawbacks of the space-time transformation method.

We have chosen to present in this chapter an alternative method, that starts from an input of the same form as in the case of the space-time transformation methodology (recurrence equations), but the scheduling of the computations is obtained in a much simpler way.

This method is based on the ideas presented in [KRS95], [KRS96] that simplify the tedious task of finding an adequate timing function. The computations are represented by the nodes of a directed graph and the time function is given by the level of the nodes in the modified dependence graph

after some empty nodes were introduces according to an algorithm based on rewrite rules, presented in [KRS94].

However, the price for the simplicity is that the size of the problem hs to be fixed in advance, thus the method does not work for parametrised problems.

Again, the method is exemplified with a significative case study: a linear systolic array is generated for the problem of sequences alignment.

Chapter 5 Functional-Based Systolic Array Design:

We introduce a *functional view* (or inductive view) of systolic arrays: a systolic array is composed of a *head processor* and an identical *tail array* of a smaller size. By exploiting the similarity between the inductive structure of a systolic array and the inductive decomposition of the argument by a functional program, it is possible to develop an elegant and efficient method for the automatic synthesis of systolic arrays.

After a detailed presentation of the formal background used, we study the functioning of different systolic array-types.

By formal analysis, the structure of the functions which can be realised by arrays having certain properties is identified. Then, by equational rewriting, the expression of the list function which must be realised is transformed into an expression having the required structure. The resulting expression reveals the scalar function which must be implemented by each individual processor.

The utility and efficiency of the design method is demonstrated through examples and representative case studies.

Chapter 6 Conclusions and future work are summarised.

1.2 Acknowledgements

...

Chapter 2

Survey

In this chapter we provide a short survey about the state of the research done in the field of automatic generation and verification of systolic arrays, used as starting point for our research. We present at first the systolic array design methods available in the literature and sum up the main ideas used. We also compare them in order to find out the advantages and weak points of the most frequently used techniques.

Concerning the available formal verification methods, they are classified according to several comparison principles and the formal framework used.

Finally we summarise the main characteristics, that we find the most important with respect to our research.

2.1 Design Methodologies

The first examples of systolic arrays that appeared in the early eighties were designed intuitively, in an "ad-hoc" manner, requiring a great amount of creativity and inspiration of its inventors. Many researchers were fascinated by the simple and regular structure of systolic arrays, able to perform complex computations on a great amount of data in a very efficient way. Later on they proposed several systematic design methodologies to overcome the limitations of heuristics and designer intuition and to provide a formal framework for mapping algorithms to systolic architectures.

In this section we present a short survey about the research done in this field, highlighting the most important characteristics of the distinct design methods. We also point out the advantages and disadvantages of the several methods.
Among the first systematic design methodologies two seemingly distinct methods, called **data dependency method** and **parameter method** were proposed.

The essence of the *data dependency method* is the representation of the dependency structure of an algorithm in concise, matrix form. The technique for mapping algorithms into systolic arrays is based on the mathematical transformations of index sets and the so called *data dependence vectors*.
This method has been extensively studied by Fortes, Wah and Moldovan [Mol83], [MF86], [FW87], S.Y. Kung [Kun87] and others. The procedure based on this method and presented in [Mol83] was also implemented at the University of Southern California in a software package called ADVIS (automatic design of VLSI[1] systems).

A procedure for the derivation of optimal systolic design was first developed in the *parameter method* [LW85]. This methodology presents the optimal mapping of algorithms that are represented as *linear recurrences* onto systolic arrays. Three sets of parameters are used to characterise systolic arrays:

- velocities of data flow
- data distribution
- periods of computation

The relationship between these parameters are represented as constraint equations that control the correctness of the design, thus the design is formulated as an optimization problem.

[1] Very Large Scale of Integration

The two design methodologies, the parameter method and the data dependency method are compared by O'Keefe, Fortes and Wah [OFW91], describing the relationship between them. It is shown that the parameter method applies to a subclass of the algorithms that can be processed by the dependency method. Explicit mathematical relations are established between the parameters, equations and constraints of the two important methods for the design of systolic arrays.

Also remarkable is the research carried out at Irisa in this field ([Qui84], [Qui87], [QR90], [QD89], [MQRS90]) that led to the design of a declarative language for systolic array description called $ALPHA$ (see [VMQ91], [WS94]) and the corresponding program transformation environment, called Alpha du centaur ([GQMS88], [DVQS91]). Later another programming environment, called $MMALPHA$ was written in Mathematica and C for manipulating Alpha programs.

The mentioned works, same as others like that of Delosme and Ipsen [DI86], Nelis and Deprettere [ND88], Huang and Lengauer [HL87] contributed to the foundation respectively the evolution of a unifying approach to the design of VLSI algorithms usually referred to as the **space-time transformation methodology**. In a certain way, all the proposed methods by the above mentioned researchers vary very little, differing slightly in the degree of formalism and the way to approach the problem. In the ultimate instance, they all use the concept of transformation of dependencies. A review of the main ideas involved in these systolic algorithm synthesis methods is presented by Song in [Son94].

The task is the transformation of a sequential algorithm, expressed in the form of up to three nested loops, or in the form of *uniform recurrence equations* (see [KMW76]), into a systolic algorithm applying a set of mathematical transformations on the *dependence vectors*.
The computations involved in an algorithm, as well as the dependences between computations, are transformed by a *time function* and a *space allocation function*.

- The *time function* maps each computation of the algorithm to a positive integer that represents the time at which the computation is executed.

- The *allocation function*, on the other hand, obtains the position of the processing element to perform the computation involved.

Such a space-time mapping must satisfy several constraints to be valid. Basically two conditions has to hold:

- if a statement depends on the result of another statement, it should be executed after the result itself is available,

- two statements allocated to the same processor should not be scheduled at the same instant of time

If we restrict ourselves to the case when the transformations applied are linear functions, it appears that the above conditions can be equivalently expressed as a linear programming problem, when one considers uniform recurrences. Linear transformations have proven to be very effective in order to design locally connected architectures such as systolic arrays. These are the main reasons why research in this domain has mainly focused on uniform recurrences, and linear transformations.
A more detailed presentation of the space-time transformation method is provided in **Chap. 3**.

In most cases, the initial specification is expressed as a recurrence equation. However, the majority of the above mentioned works have worked with uniform recurrence equations (URE), a subclass of linear recurrence equations (LRE).

The *uniformisation problem*, that is to say the problem of transforming linear recurrence equations into uniform recurrence equations is tackled by Quinton and Dongen in [QD89], Fortes and Moldovan [FM84] and others. Such a transformation is mandatory when the target architecture is locally connected, as for example, a systolic array.
A more general review about the various transformations that can be applied to a system of recurrence equations is presented by Lavenier, Quinton and Rajopadhye in [LQR99].
The main formal manipulation techniques of *SRE*s are summarised and exemplified:

- Serialization of Reductions
- Alignment
- Localization
- Change of basis

Many algorithms were already parallelised using the efficient technique of time-space transformations. However, this methodology also has some drawbacks. The constraint, that a location in space is assigned to each index value from the very beginning causes that the data will be introduced in a regular order. This is well suited for the design of regular architectures, but it also can lead to the exclusion of other possible solutions of the problem.
Time and space boundary conditions often constitute another problem in the design of systolic arrays.

The problem should usually be specified in the form of *uniform recurrence equations* (URE). The class of problems that can be expressed in this form is restricted and expressing most problems using UREs is not always an easy task.

Various attempts have been made to overcome the drawbacks of the space-time transformation method. A remarkable alternate approach represents the methods based on viewing systolic design as program design.
Gribomont and Dongen presents another methodology for the mapping of systolic arrays based on the concept of **generic systolic array** introduced and illustrated in [GD92].
This approach represents a program-oriented methodology, being more closed to the program design. The technique takes in account two points, namely that:

- the architecture is often chosen before the real beginning of the development
- the basic operations to be executed by the individual cells are partially known at the beginning

Thus the development starts from a *generic systolic array*, whose parameters have to be instantiated. A more detailed description of the method is given in [GD92].

As already mentioned, this method is more closed to the program design, thus cannot be fully automated. It still needs some creativity and intuition from the designer.

Another disadvantage of the space-time transformation method is that it heavily depends on finding an affine timing function. The problem with finding an affine timing function is that one needs to solve a system of linear recurrence equations, which is generally difficult, and also possible only for systems having certain properties. The method proposed by Kazerouni, Rajan and Shyamasundar in [KRS95], [KRS96] is not restricted to affine timing functions.
They propose a general method for mapping a system of linear recurrence equations (SLRE) onto specific systolic architectures. Their method generates solutions suitable for existing target architectures rather than designing new ones. This means that the design method becomes easier, one does not have to solve a system of linear equations, but the solution is of fixed size, no parameters are taken into consideration.

The method essentially consist of mapping *normalised linear recurrence equations*, a subclass of linear recurrence equations, - which properly includes the class of uniform recurrence equations - onto a generic architecture called *basic systolic architecture* and then applying correctness preserving transformations to adopt this intermediate solution onto specific target architectures (also presented in [KRS94]).

For additional details one may also see the short survey about the systolic array design methods available in the literature [Sza02a],[Sza02b].

2.2 Partitioning

The synthesis method for systolic arrays based upon space and time transformations can be extended to generate systolic implementations on a fixed number of processors. The main idea of all these extensions is to merge many cells into a single processor, so as to compress the array. This step is called *partitioning*. Algorithm partitioning is essential when the size of the computational problem is larger than the size of the systolic array intended for that problem.
It can take two different forms:

- The locally parallel globally sequential (**LPGS**) form, presented by Moldovan in [MF86].

- The locally sequential globally parallel (**LSGP**) form, studied by Darte, Delosme [DD90], [Dar91], Bu, Deprettere and Dewilde [BDD90].

The LPGS approach means that the array is first partitioned into blocks, the size of which is the number of available processors (p), and then each block one after the other is computed. The different computation points in the current block are allocated and scheduled in the p processors, in accordance to the dependence constraints. This method requires small local memories, but a large external buffer is needed to store the data used by the next block. In the LSGP approach the array obtained by the mapping procedure is partitioned into p blocks of virtual processors, each block being allocated to one physical processor.

Of course, the different points allocated to the same processor have to be computed at different times in the array in such a way that they can be sequentially executed by the physical processor. This method permits to synthesise systolic arrays with a fixed number of cells and, as a particular case, permits to improve the efficiency of the cells in a systolic array obtained by the projection method.
The two methods are illustrated in Fig. 2.1.

a) In the LPGS scheme, the block size is chosen to match the array size, i.e. one block can be mapped to one array. All nodes within one block are processed concurrently, i.e. locally parallel. One block after another block of node data is loaded into the array and processed in a sequential manner, i.e. globally sequential.

b) In the LSGP scheme, one block is mapped to one processing element (PE). Each PE sequentially processes the nodes of the corresponding block. The number of blocks is equal to the number of PEs in the array.

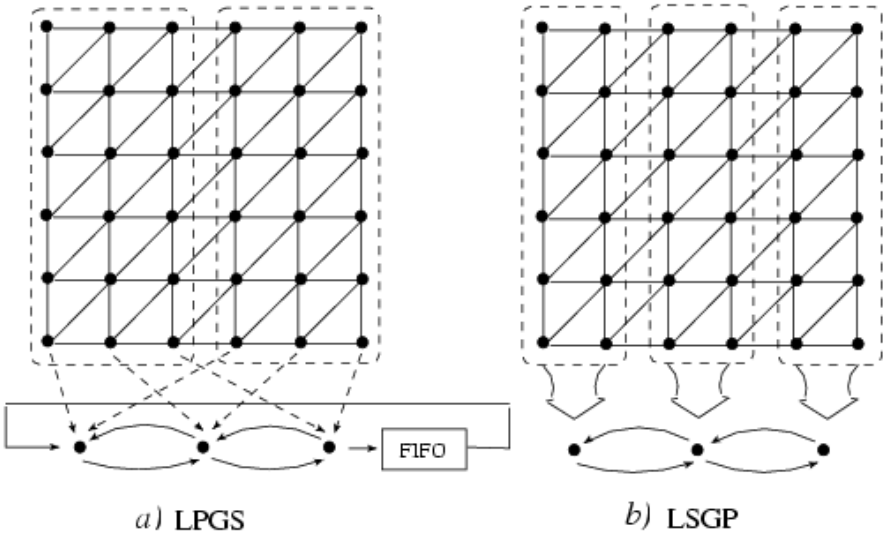


Figure 2.1: LPGS and LSGP method [CK98].

In the LPGS method, a general rule is to select a (global) scheduling that respects the data dependencies.
As to the scheduling scheme for the LSGP, after processor allocation, an acceptable (that is sufficiently slow) schedule is chosen so that at any instant there is at most one active PE in each block.
Note that the LPGS design has the advantage that blocks can be executed one after another in a natural order. However, this simple ordering is valid only when there is no reverse data dependence for the chosen blocks.

A unified partitioning and scheduling scheme is proposed for LPGS and LSGP in [HH95]. The main contribution includes a unified partitioning model and a systematic two level scheduling scheme. The unified partitioning model can support LPGS and LSGP design in the same manner. The systematic two level scheduling scheme can specify the intra-processor schedule and interprocessor schedule independently. Thus, more interprocessor parallelism can be effectively explored. The design methodology presented in [CK98] also proposes such a unified approach to the partitioning problem.

2.3 Verification of Systolic Arrays

An overview about the formal verification methods used for validating systolic arrays is presented in this section.

The traditional way of validating hardware systems was simulating and testing. This method is limited not only because of its expensiveness, but it is also hard to achieve total fault coverage. For this reason formal verification methods are considered more and more often as an alternative approach to ensure the quality and correctness of hardware designs [KG99].

Also the formal hardware verification methods have their limitation and the most significant one is their complexity. Even so, the regularity of the systolic systems and the simplicity of the processors make automatic reasoning and analysis possible.

2.3.1 General Aspects Concerning Formal Verification Methods

Formal verification is like a mathematical proof. The correctness of the hardware system is determined regardless of its input values, by considering its function rather than its behaviour, and by verifying logical properties.

Generally, when a formal verification has to be performed, two descriptions of the problem are given: a *specification* of the system and an *implementation*. Implementation and specification are in fact representations of the circuit at different levels of abstraction.

- the *specification* of the system describes how we expect the output data of the system to be related to the input data.
- the *implementation* describes how the system is built (behaviour of PE-s and how they are connected to each other in the circuit).

The task of formal verification is to check – using rigorous mathematical reasoning (instead of an experimental approach) – whether the given implementation really meets all parts of its specification.

A survey about formal hardware verification in general can be found in [Gup91] or [KG99].

Hereinafter we describe some important aspects, comparison principles to be taken into consideration when talking about formal methods of hardware verification, specially systolic array verification.

The first step towards developing an automatic verification method is to find an adequate formal framework that should aim to be simple, complete, precise, coherent, automatable, efficient and reliable. There have been many

proposals for specification of systolic systems. They differ in expressiveness, treatment of semantics, range of applicability, etc.
The specification and implementation models used are: *logic* [Gor87] - propositional logic, first-order predicate logic, higher-order logic, temporal logic, or *automata/language theory* [Mil91] - finite state automata, trace structure, process algebra. An important aspect is the degree of automation offered by a verification technique, i.e. is it enough to describe the specification and the implementation in a proper way or we have to provide additional information during the verification process.
In the literature we find techniques ranging from completely automated methods (e.g. [LB99]) to interactive theorem proving using logical calculi (where the user should employ tactics or suggest appropriate lemmas during the verification process, e.g. Boyer-Moore theorem prover system [Gri88], [BM79], HOL system (due to M. Gordon) [Gor87], NURPL proof development system (due to N. Constable) [DLT89].

Another possibility to compare verification methods is to consider the class of systolic arrays that can be verified with a method. The size of the class of problems that can be verified with a certain method is usually in inverse proportion to the complexity of the method. This means that completely automated verification methods usually can be applied to a restricted class of systolic algorithms, while methods, which can solve a more general class of algorithms often require "extra information" (e.g. help lemmas, induction hypotheses) to succeed in performing the proof.

A good communication between designers and implementers is also a requirement, therefore the *ease of use* of a method could be an advantage compared to very general and powerful methods, that are restricted mostly to verification experts.

Another question concerning verification methods is how it could be applied. That means, whether it only gives a certification that the given implementation of the array is correct in respect to the problem it should solve, or we can also learn more about the array when performing the verification task. In the literature one can find formal frameworks and methods that are suitable not only for verification purposes but also for the design of systolic arrays, simulation, fault diagnoses (see [DQ94], [LB99]).

A hardware module can be abstracted at various *architectural levels*. Some examples of abstraction levels could be:

- *behavioural modeling* (also referred to as functional or system-level) – the hardware module is described - using a high level language - by its logical behaviour rather than its physical implementation

- *register-transfer level* – where the register values are the basic operands

with combinational logic blocks between them based on standard logic elements (multiplexer, shifter, adder)

- *gate-level* – description is based on standard logic gates like *and*, *or*, *xor*, *not*, etc.

Verification can be performed on different levels of abstraction or even in the course of successive steps on different levels of abstraction in a top-down or bottom-up manner. The implementation at a certain level i becomes the specification of a more detailed level $i-1$ and the specification at level i is the implementation of a more abstract level $i+1$.

The notion of *abstraction* permits unnecessary detail to be hidden from the high-level model.

Melham [Mel88] describes four different *types of abstraction* widespread in hardware verification.

- *Structural abstraction* hides the details about the implementation's internal structure in the specification. The specification gives a 'black-box' view of the design, that describes only the system's behaviour observable from outside, without entering into details about the internal design.
- *Behavioural abstraction* suppresses details about what the component does under operating conditions that should never occur. Behavioural abstraction may also include "don't care" conditions.
- *Data abstraction* relates signals in the implementation to signals in the specification when they have different representations. Data abstraction requires a mapping that determines how the states or signals of the implementation are to be interpreted in the specification's semantic domain.
- *Temporal abstraction* relates time steps of the implementation to time steps of the specification.

Ling an Bayoumi [LB89] specifies also another type of abstraction, specific for systolic arrays:

- *Abstraction of Systolic features*: Systolic arrays have some particular properties at the array level. These are synchrony, regularity, repeatability, modularity, spatial and temporal locality, pipelinability and parallel processing ability [Kun87].

Abstraction of systolic features means to use a formalism (eventually introduce a new one) that exploits these properties.

2.3.2 Classification of the Formal Frameworks Used

Research in the domain of the formal verification of systolic arrays uses ideas, frameworks and techniques from various research fields including first or higher-order predicate logics, temporal logics, fixed-point induction, automated theorem proving, automata-theoretic techniques, language containment, process algebra, etc.

According to [KG99] from a conceptual point of view there are two main approaches to the specification and to the corresponding verification of hardware, referred to as: *property verification* respectively *implementation verification.*

Property Verification

This approach is concerned with specifying desired *properties* for the design. Formal verification in this case is generally concerned with properties of a temporal nature. That means, it applies to the characteristics of the system's behaviour on execution rather than to static attributes of the system.

Temporal logics [Eme90] are a unifying framework for expressing such properties. Temporal logic – is a generalisation of predicate logic to enclose the temporal domain for effective description of dynamic environments. It substantially advances traditional logic because it can capture time and dynamic behaviours - essential features in hardware descriptions - with concise clear notation. It avoids the introduction of explicit time functions and time variables. [LB99]

The verification task consists in showing that all of the system's possible behaviours satisfy the temporal properties of its specification.

In [LB89] Ling and Bayoumi introduces a novel formalism based on *interval temporal logic* (ITL), developed for systolic array reasoning, called *Systolic Temporal Arithmetic* (STA). The advantages of the developed formalism are its ease of use (STA does not require users to find out any assertions or loop invariants for verifications), the fact that the STA exploits systolic array properties to produce notation and reasoning methods suitable for systolic arrays to make specifications and verifications more simple and effective.

Implementation Verification

This second approach is based on specification in terms of a high-level model of the system. In this framework, verification requires reasoning about the relationship between the high-level model, also referred to as *specification* and a lower-level model, that is the *implementation*.

In order to be able to reason about the implementation and the specification, formal interpretation must be given for their description. There are three common approaches to this formalisation:

1. Specification of the system's behaviour in terms of functions and predicates of *standard first or higher-order predicate logic*.

   Theorem proving systems based on first or higher-order logic have been used to verify systolic circuits: the Boyer-Moore theorem proving system, based on first order logic, used to mechanise the verification method presented by Purushothaman and Subrahmanyam [PS89]; higher order logic systems like HOL, described in [Gor87] or NURPL in [DLT89].

2. Description of the system behaviour by a *state transition system*, whose definition can be expressed in languages such as CSP[2].

   Here we mention that due to the regularity of the systolic arrays recursivity is very often used in the description of such systems. Recurrence equations, specially *uniform recurrence equations* are most suitable for this purpose.

   Gribomont [Gri88] uses a small language derived from Hoare's CSP to describe the processing units of the systolic array. Chen [Che83] uses systems of recursive equations to describe systolic circuits.

3. Description in terms of infinite languages recognised by a finite state automata.

   In the paper of Margaria [Mar96] the internal structure of the basic cell is specified as a Finite State Machine and implemented in hardware at the gate-level.

[2]CSP– Communicating Sequential Processes, a language for describing patterns of interaction between processes, introduced by C. A. R. Hoare [Hoa85]

Top-Down Synthesis – Button-Up Abstraction – Meet in the Middle Comparison

Another approach to systolic array verification is the combination of the proof process with that of synthesis. In [DQ94] an interesting verification method is presented, where the proof process is a combination of top-down synthesis and bottom-up abstraction until a common middle-point is reached (the idea was proposed by Hans Eveking[Eve87]). This method takes into consideration also the available architectural synthesis methods for systolic arrays.

The formal representation of both, the specification and the implementation of an array is given in ALPHA, a functional/equational language used for synthesis of regular synchronous architectures [DQ94].

A circuit can be represented in the ALPHA language at different levels of abstraction: structural description as well as functional or temporal abstraction level.

The verification is performed by doing program transformations based on the semantics of ALPHA. The proof process is semi-automatic in the sense that the designer has to select transformations whose application is automatic. The proof methodology deals with parametrised circuits.

- The *specification* of the array consists of the recurrence equations describing its operations.
- The *implementation* is the actual model of the array at register transfer level.

By *synthesis*, the recurrence equations are transformed in an explicit architecture.

By *abstraction*, the implementation is simplified by eliminating all initialization mechanisms, and some optimizations which were applied when designing the array. The implementation is abstracted up to the same representation level by means of induction.

The *verification* is completed by matching both descriptions.

2.4 Synthesis and Verification – Conclusions

We provided in the former section a short survey about the systolic array verification methods available in the literature, pointing out their main characteristics, and we also gave some principles of comparison of these methods.

Temporal logics, specially interval temporal logic (ITL) turns out to be a very concise and powerful framework to reason about systolic arrays. In this case we are talking about property verification, as the formal verification is concerned with properties of a temporal nature, i. e. the characteristics of the system's behaviour on execution.

Due to the regularity of systolic array structure, recursivity is frequently used in their formal description. Therefore fixed-point induction is a commonly used technique in systolic array verification, though it is not always simple to find the fix-point of a recursive system. Theorem proving systems, based on first or higher-order logic are also used to automatically perform the proof or at least a part of it, but beside the advantage of the computational power of such a system there is usually also a disadvantage, namely that additional information (in terms of lemmas) has to be given to the system in order to succeed. This makes their use rather difficult, available only for experts.

It is hard, actually impossible, to find a "perfect" verification method, suitable for a large set of problems, completely automatic and also easy to use, not only for experts. Yet, we would like to point out some characteristics, we find the most important in connection with our work. A very important aspect is the appropriate choice of the modeling language, preferable to be closer to the modeling practice of hardware designers. Another important feature that we would expect from a verification method is not only to certificate the correctness of a given hardware but also to make it possible to gain more information about its design.

Systolic array synthesis and verification are two research fields strongly related to each other. In this context, we find the idea of combining the synthesis and the verification process very useful.

Chapter 3

The Space-Time Transformation Methodology

The *space-time transformation method* is the most commonly used design technique, more exactly it is the common name for the main ideas used in several design methods. All of these techniques are based on the concept of transformations applied to the index space representing the computations. The purpose of the transformations is to determine *where* and *when* the computations will be performed. This is the reason for the term "space-time" in the name of the method.

In this chapter we give a more detailed description of this particular design technique for the sake of a more precise comparison with our novel method presented in Chap. 5.

An interesting case study is presented at the and of the chapter, that will be revisited in Sect. 5.6.4, thus it can reveal the advantages and drawbacks of the two methods. The design of the case study was actually inspired by the online array generated with our functional–based method.

3.1 Introduction

The **space-time transformation methodology** is a unifying approach to the automatic design of systolic algorithms based on the concept of transformation of dependencies. In the sequel we will present the basic notions and commonly used ideas characteristic to this method, pointing out also the difficulties encountered.

A common way of describing an algorithm is that of using abstract specifications such as *recurrence equations*.

The design methods based on the space-time transformation use as starting point the representation of the sequential algorithm in terms of a system of *Uniform Recurrence Equations* (URE), a subset of Linear Recurrence Equations. The reason for using URE systems is that only such systems are suitable for being *directly* mapped onto systolic architectures, as they require local data and local interconnections between the processing elements.

In order to apply the method to a larger set of problems - e. g. Linear Recurrence Equations - the problem of uniformisation should be studied (also discussed in [QD89]).

Definition 3.1. Uniform Recurrence Equation System
A *system of uniform recurrence equations* ($SURE$) is a collection of $s \in N$ equations of the form (3.1) and input equations of the form (3.2):

$$V_i(z) = f_i(V_1(z-\theta_{i_1}), \ldots, V_k(z-\theta_{i_k})) \quad (3.1)$$
$$V_i(z_i^j) = v_i^j, j \in \{1, ..., l_i\} \quad (3.2)$$

where

- $V_i : D \rightarrow R$. $V_i, i \in \{1, \ldots, s\}$ are variable names belonging to a finite set V. Each variable is indexed with an integral index, whose dimension, n (called the *index dimension*), is constant for a given SURE (in practice this is usually 2 or 3).

- $z \in D$, where $D \subseteq \mathcal{Z}^n$ is the domain of the SURE. (In the following we consider limited convex polyhedral domains. See [Wil93]).

- v_i^j is a scalar constant (input), $z_i^j \in D_{inp}$, where $D_{inp} \subseteq \mathcal{Z}^n$ is the domain of the inputs.

- $\theta_{i_1}, \ldots, \theta_{i_k}$ are vectors of Z^n and are called *dependence vectors* of the $SURE$.

- $V_i(z)$ does not appear on the right-hand side of the equation

- $f_i : R^s \rightarrow R$

Note: We gave a simplified form for the equations of a $SURE$ in (3.1), respectively (3.2). Generally a computation may consist of several cases. Also the domains associated to the several computations may be different. A more general form for the input equations would be the following:

$$V_i(z) = \begin{cases} f_{i,1}(\ldots) & , z \in D_{i,1} \\ \ldots & \\ f_{i,c}(\ldots) & , z \in D_{i,c} \end{cases} \tag{3.3}$$

where $D_{i,n} \bigcap D_{i,m} = \varnothing$ for $n \neq m$. $D_i = \bigcup_{j=1,c} D_{i,j}$. The domain D of the $SURE$ is the convex hull of all D_i domains. In the same way, the input equations may also consist of more cases.

Unless it is uniform, a system of recurrence equations cannot be mapped directly on a systolic array. Indeed, if the dependence vectors are not constants, the PEs may have to communicate with an arbitrary large number of other PEs (also called *broadcasting*), and this is not desirable for a systolic array, where the PEs should only be connected with their neighbours.

Definition 3.2. (**Dependence Vectors**)
Let $\Theta = \{\theta_0, \theta_1, \ldots, \theta_l\}$ the set of vectors θ_i of a $SURE$ of the form (3.1)-(3.2). The θ_i vectors are called *dependence vectors* of a system.

Consider points $z, z' \in D$. We say that z is dependent on z' by θ_i, if $\exists\ \theta_i \in \Theta$, such that

$$z = z' + \theta_i.$$

Given a $SURE$ of the form given in definition 3.1, we say that the variable instance $V_i(z)$ *depends on* $V_j(z')$ if there is an equation of the form (3.1), where $V_i(z)$ appears on the lhs. of the equation and $V_j(z')$ is on the rhs. We denote the dependence by $V_i(z) \leftarrow V_j(z')$.

The dependences can be represented by a directed graph called *dependence graph* of the $SURE$. It abstracts the dependency relations among the variables in the $SURE$.

Definition 3.3. (**Dependence Graph – DG**)
The dependence graph of a $SURE$ is a directed graph $G(V, E)$, (also denoted by (D, Θ)), where
the set of nodes $V = D$, i.e. the nodes are points of D, and
the set of edges $E = \{\varepsilon_{zz'} | \exists \theta_i \in \Theta, z = z' + \theta_i\}$, i.e. if z is dependent on z' by θ_i, there exists an edge from node z' to node z.

Informally, a $SURE$ (as well as the associated dependence graph) can be seen as a multidimensional systolic array, where the points of the domain D (respectively the nodes of the DG) are the PEs of the array and the communication channels are determined by the dependencies (the edges of the DG). In this context a transformation applied to the system that preserves the number of domain points and the dependencies leads to a computationally equivalent system. The goal of such a transformation is to obtain a system where one of the indices can be interpreted as the *time index* and the others as *space-indices*.

Linear (and affine) transformations are most commonly used because they preserve the dependencies between the computations. Moreover, if the transformation is *unimodular*[1], then it has the advantage that it preserves the number of points in the domain, and in addition it admits an integral inverse. However it is not mandatory to use unimodular transformations.

In the following sections we show how an adequate timing function respectively allocation function can be found.

3.2 Timing Function

Given a URE system, we want to obtain a timing function t that schedules the computations associated to the points of domain D. For a given $z \in D$ we assume that the computations $V_i(z)$ are performed in parallel, and take a unit time.

A natural requirement is that in order to perform the computations of $V_i(z)$, its arguments should have been computed before. The time function t associates to each point $z \in D$ the time instant when it is computed, such that the dependencies are respected. If such a function exists, then we say that the $SURE$ is *computable*.

Definition 3.4. (**Timing Function**)
For each variable V_i of the $SURE$ of the form given in definition 3.1, we call the *timing function of the variable* V_i the positive function $t_{V_i} : D \to \mathcal{N}$, that gives for any point z of the domain D the time instant when $V_i(z)$ will be computed, such that for each dependence $V_i(z) \leftarrow V_j(z')$ of an equation of the form (3.1) should hold:

$$t_{V_i}(z) > t_{V_j}(z') \ .$$

[1] A transformation $x' = Ax$ is unimodular if the determinant of the matrix A satisfies $det(A) = \pm 1$.

For each variable V_i of the equations of the form (3.1) of a $SURE$, we are looking for an affine timing functions $t_{V_i} : D \rightarrow \mathcal{N}$ having the same linear part:

$$t_{V_i}(z) = T * z + \delta_{V_i} \ .$$

Note that T is an $1 \times n$ vector $T = (t_1, \ldots, t_n)$. The fact that the timing function has the same linear part for each variable assures that after performing a transformation on the $SURE$ according to the timing functions, the uniformity of the $SURE$ will be preserved.

Constraints

The timing function has to satisfy the following two conditions:

1. (positivity constraint) $\forall V$ variable of the $SURE$, $\forall z \in D$,

$$t_V(z) \geq 0 \tag{3.4}$$

2. (dependency constraint) For each dependence $V_i(z) \leftarrow V_j(z')$ of an equation of the form (3.1) should hold:

$$t_{V_i}(z) > t_{V_j}(z') \tag{3.5}$$

The first condition is a matter of convenience, since t is interpreted as time. The second condition makes scheduling of dependent computations possible.

We might also want to minimise the computation time of the system. This would mean to add the corresponding restriction on T, namely that the sum $t_1 + \ldots + t_n + \sum_{V_i} \delta_{V_i}$ should be minimised.

3.2.1 Finding an Adequate Affine Timing Function

The previously mentioned constraints build a system of inequalities. Any of its solutions - if there exists any (that means, the system is computable) - gives an adequate timing function. The problem of the computability of a $SURE$ in its full generality is undecidable [SQ93]. However there exists necessary and sufficient conditions for the restricted class of *affine timing functions*, which is of practical interest.

The theorem 3.1 of Quinton and Roberts gives a method to automatically determine, whether there exists an affine timing function for a parametrised system of equations. The method based on it is also referred to as vertex method. Its details are presented in [Qui87] and [QD89].

Theorem 3.1. (Quinton and Roberts [QR90])

For a given $SURE$, with dependence graph $(D;\Theta)$ and set V_D of vertices of D, the parameters $\lambda_0, \lambda_1, \ldots, \lambda_{n-1}$ and γ define a timing function t iff

1. $\forall v_i \in V_D, \lambda^T v_i + \gamma \geq 0$ and
2. $\forall \theta_i \in \Theta, \lambda^T \theta_i > 0$, or $\lambda^T \theta_i \geq 1$.

See [QR90] for the proof.

Note that if z is dependent on z' by θ_i, or $z = z' + \theta_i$, then

$$\lambda^T \theta_i = \lambda^T (z - z') = t(z) - t(z').$$

Thus $\lambda^T \theta_i$ expresses the *delay* between the computation of z and the computation of z'. Theorem 3.1 requires this delay to be greater than 0 (at least 1).

Theorem 3.1. gives conditions for the obtainment of a time function. Another question is that of optimality of the time function. Several criteria for optimality can be adopted. Consider a $SURE$ with dependence graph $(D;\Theta)$. One criterium is to minimise the delay between dependent computations according to a dependence $\theta_k \in \Theta$. Thus we want to minimise $\lambda^T \theta_k$, subject to

$$\lambda^T \theta_i \geq 1, \forall \theta_i \in \Theta.$$

An alternative method for the compression of the constraints into a finite set is the so called *Farkas method* [Fea92], that uses the following version of the Farkas Lemma:

Theorem 3.2. (affine form of Farkas lemma)

Let D be a polyhedron defined by the inequalities $A * z + b \geq 0$, where A is a $k \times n$ matrix. Then the affine form ϕ is non negative on D if and only if there exist $\lambda_0, \Lambda = (\lambda_1, \ldots, \lambda_k) \geq 0$ such that $\phi = \Lambda(\ A \ \ b\ ) + \lambda_0$.

See [Sch86] for the proof.

A linear constraint can always be written in the form that a linear function is positive on a domain, thus one can make use of Theorem 3.2.

According to the method presented in [Fea92], the conditions (3.4) - (3.5) that a timing function t must satisfy can be replaced in the following way using Theorem 3.2:

- **positivity constraint:**
  if a variable V is defined on the domain $D_V = \{z|a_i z + b_i \geq 0, 1 \leq i \leq k\}$, the timing function $t_V(z) = T * z + \gamma_V$, $(T = (\tau_1, \ldots, \tau_n))$ is positive on D_V iff $\exists \lambda_0, \Lambda \geq 0$

$$\begin{cases} T * z + \gamma - \sum_{i=1}^{k} \lambda_i (a_i * z + b_i) - \lambda_0 = 0 \\ \lambda_i \geq 0, \quad 0 \leq i \leq k \end{cases} \tag{3.6}$$

  From 3.6 we have $n + 1$ equations:

$$\begin{cases} \tau_j - \sum_{i=1}^{k} \lambda_i * a_{ij} = 0 & 1 \leq j \leq n \\ \gamma - \sum_{i=1}^{k} \lambda_i * b_i - \lambda_0 = 0 & \\ \lambda_i \geq 0 & , 0 \leq i \leq k \end{cases} \tag{3.7}$$

- **dependency constraint:**
  If a variable $V(z)$ depends on $W(z')$, by $\theta = z - z'$ on the domain $D_{VW} = \{z|A * z + b \geq 0\}$, where A is an $k \times n$ matrix, then the timing functions $t_V(z) = T_V + \gamma_V$ and $t_W(z) = T_W + \gamma_W$ $(T_V = (\tau_1^V, \ldots, \tau_n^V)$, $T_W = (\tau_1^W, \ldots, \tau_n^W))$ respect the dependency constraints, that is $t_V(z) - t_W(z') - \delta \geq 0$ (δ is a delay between the computation of W and V) iff $\exists \mu_0, (\mu_1, \ldots, \mu_k) \geq 0$

$$\begin{cases} T_V * z + \gamma_V - T_W * z' - \gamma_W - \delta - \sum_{i=1}^{k} \mu_i (a_i * z + b_i) - \mu_0 = 0 \\ \mu_i \geq 0, \quad 0 \leq i \leq k \end{cases} \tag{3.8}$$

  Again, from (3.8) we have $n + 1$ equations:

$$\begin{cases} \tau_j^V - \tau_j^W - \sum_{i=1}^{k} \mu_i * a_{ij} = 0 & 1 \leq j \leq n \\ \gamma_V - \gamma_W - \delta - T_W * \theta - \sum_{i=1}^{k} \mu_i * b_i - \mu_0 = 0 & \\ \mu_i \geq 0 & , 0 \leq i \leq k \end{cases} \tag{3.9}$$

- **objective function constraints:**
  Using the Farkas method, one can find many schedules that satisfy the positivity and dependency constraints. This additional type of constraints depends on our choice related to the optimization of the problem from a certain point of view (this could be for example the minimization of the overall execution time).

3.3 Allocation Function

Given a suitable timing function, the next step is to find an adequate allocation function.

Definition 3.5. (**Allocation Function**)
For each variable V_i of the $SURE$ of the form given in definition 3.1, we call the *allocation function of the variable* V_i the function $p_{V_i} : D \rightarrow D'$, such that $D' \subseteq Z^{n-1}$ and for any point $z \in D$, $p_{V_i}(z)$ is the index of the PE where the computation of $V_i(z)$ will take place.

Given an affine timing function with the same linear part for every variable V_i, we are looking for an adequate allocation functions having the same property. That is $\forall V_i$ of the equations of the form (3.1) of the $SURE$ we should find an allocation function $p_{V_i} : D \rightarrow D'$ of the form

$$p_{V_i}(z) = P * z + \gamma_{V_i},$$

where $D' \subseteq \mathcal{Z}^{n-1}$.

3.3.1 Obtaining an Adequate Allocation Function

We describe in this section how a set of constraints can be set up for the obtainment of suitable allocation function(s) for a given timing function.

Constraints

The following set of constraints can be defined:
General constraint:
In order to get an adequate allocation function for a given timing function, the condition that should hold can be intuitively expressed in the following way: two different computations performed at the same time-step should not be mapped onto the same PE. This means that the linear part P of the allocation function should not be parallel to the direction T corresponding to the timing function.
Formally:

$$P \neq c * T, \tag{3.10}$$

where $c \in \mathcal{Z}$ is a constant.

Weak conditions:
For every dependence $V_i(z) \leftarrow V_j(z')$ of an equation of the form (3.1) such that $V_i \neq V_j$, the following condition should hold:

$$\|p_{V_i}(z) - p_{V_j}(z')\| \leq t_{V_i}(z) - t_{V_j}(z') \tag{3.11}$$

(that is the value $V_j(z')$ should be computed "close enough" to the place where $V_i(z)$ will be computed, so that it can arrive from PE with index $p_{V_j}(z')$ to PE with index $p_{V_i}(z)$ in $t_{V_i}(z) - t_{V_j}(z')$ time steps).

Strong conditions:
We define the data-flow direction for each variable V_i.

Definition 3.6. (**Data-Flow Direction**)
In the case of the variable V_i, if in the equation of the form (3.1) that computes $V_i(z)$, there is one and only one occurrence of variable V_i on the rhs of the equation, such that $V_i(z)$ depends on $V_i(z-\theta_{V_i})$, then we obtain the *data-flow direction of variable* V_i, denoted Dir_{V_i} by applying the linear time-space transformation (T, P) onto the dependence θ_{V_i}:

$$Dir_{V_i} = (T * \theta_{V_i}, P * \theta_{V_i}),$$

where the component $T*\theta_{V_i}$ indicates the "velocity" of V_i and the component $P * \theta_{V_i}$ shows the "direction" along which V_i will be transmitted.

For each dependence $V_i(z) \leftarrow V_j(z')$ of an equation of the form (3.1) such that $V_i \neq V_j$, the following should hold:

$$\|p_{V_i}(z) - p_{V_j}(z')\| = \left\lfloor \frac{t_{V_i}(z) - t_{V_j}(z')}{T * \theta_{V_j}} \right\rfloor P * \theta_{V_j} \tag{3.12}$$

3.4 Performing the Time-Space Transformation

Given the timing function t_{V_i} and the allocation function p_{V_i} for each variable of the $SURE$, we apply these functions onto the system, performing a so called "time-space transformation" (that means a transformation performed on the index space of the system). As a result we get an equivalent system in which the first index of the domain can be interpreted as the *time index* (representing the time instant when a certain computation takes place) and the other indices are the *space indices*, denoting the number of the PE where a certain computation takes place.

A computation $V_i(z), z \in D$ will take place at the time instant $t = t_{V_i}(z)$ at the PE with number $p = p_{V_i}(z)$. We denote the computation in the transformed $SURE$ corresponding to $V_i(z)$ with $\overline{V_i}(t, p)$. The time-space transformation corresponding to variable V_i is $Tr_{V_i} : D \rightarrow \overline{D}$, such that $\forall z \in D,\ Tr_{V_i}(z) = (t_{V_i}(z), p_{V_i}(z))$.

There exists the inverse function $Tr_{V_i}^{-1} : \overline{D} \rightarrow D$, such that $\forall (t,p) \in \overline{D}$, the computation $\overline{V_i}(t,p)$ of the transformed $SURE$ corresponds to the computation $V_i(Tr^{-1}(t,p))$ from the original system.

The time-space transformation is performed in the following way:

- In all equations of the form (3.1):
  - we replace $V_i(z)$ on the *lhs* of the equation with $\overline{V_i}(t,p)$, and
  - for each dependence $V_i(z) \leftarrow V_j(z-\theta)$:
    we replace $V_j(z-\theta)$ by $\overline{V_j}(t_{V_j}(Tr_{V_i}^{-1}(t,p)-\theta), p_{V_j}(Tr_{V_i}^{-1}(t,p)-\theta))$.

  On the following scheme the arrows depict how the correspondencies between computations of the original, respectively the transformed $SURE$ are found:

$$\begin{array}{lccc} \text{n.:} & \overline{V_i}(t,p) & \text{dep. on} & \overline{V_j}(t_{V_j}(Tr_{V_i}^{-1}(t,p)-\theta), p_{V_j}(Tr_{V_i}^{-1}(t,p)-\theta)) \\ & \downarrow & & \uparrow \\ \text{o.:} & V_i(Tr_{V_i}^{-1}(t,p)) & \text{dep. on} & V_j(Tr_{V_i}^{-1}(t,p)-\theta) \end{array}$$

where "o." stands for the original and "n." for the new dependency.

3.5 Polynomial Multiplication - Case Study

In this section we present a detailed case study which describes the design process of a systolic array for polynomial multiplication. The problem is a relatively simple one, however the array that we obtain is not trivial. The choice for this particular problem becomes even more interesting because it is revisited is Sect. 5.6.4 where an online array is generated with our functional–based method. Thus the problem presented reveals the differences, advantages and disadvantages of the two design methods.

Specification:

Let A and B two univariate polynomials of degree $n-1$ and $m-1$ respectively:

$$A = a_0 + a_1 * x + a_2 * x^2 + \ldots + a_{n-1} * x^{n-1}$$
$$B = b_0 + b_1 * x + b_2 * x^2 + \ldots + b_{m-1} * x^{m-1}$$

We denote the product of A and B with C (polynomial of degree $n+m-1$),

$$C = A * B = c_0 + c_1 * x + c_2 * x^2 + \ldots + c_{m+n-2} * x^{n+m-2},$$

where

$$c_k = \sum_{i+j=k} a_i * b_j, \quad \forall k,\ 0 \le k \le m+n-2; 0 \le i \le n-1, 0 \le j \le m-1 \tag{3.13}$$

We would prefer a recursive description of the coefficients of C rather than that given in equation (3.13). One possibility is to use the following notation (where A_i stands for a_i if $0 \le i \le n-1$, otherwise it is 0. Similarly $B_j = b_j$ if $0 \le j \le m-1$, and 0 otherwise.):

$$A*B = \underbrace{A_0 * B_0}_{C_{0,0}} + (\underbrace{A_0 * B_1 + A_1 * B_0}_{C_{0,1}})*x + (\underbrace{A_0 * B_2 + \overbrace{A_1 * B_1}^{C_{1,1}} + A_2 * B_0}_{C_{0,2}})*x^2 +$$

$$+ (\underbrace{A_0 * B_3 + \overbrace{A_1 * B_2 + A_2 * B_1}^{C_{1,2}} + A_3 * B_0}_{C_{0,3}}) * x^3 + \ldots$$

Generally:
$\forall i, j :\ 0 \le i \le j;\ i + j \le m + n - 2$

$$C_{i,j} \quad = \quad \begin{cases} A_i * B_i & , i = j \\ A_i * B_j + A_j * B_i & , j = i + 1 \\ A_i * B_j + A_j * B_i + C_{i+1,j-1} & , j > i + 1 \end{cases} \tag{3.14}$$

The result: $c_k = C_{0,k}, \forall k,\ 0 \le k \le m + n - 2$.

3.5.1 Uniformisation of the Recurrence Equation

In equation (3.14) A_i is needed in the computation of $C_{i,j}$ for all values of j, $i \le j \le m + n - 2 - i$ this means a broadcast of A_i (to $\ldots$). Similarly A_j is needed in the computation of $C_{i,j}$, $\forall i, 0 \le i \le m + n - 2 - j$. A common method to eliminate broadcast is to pipeline the given value through the nodes where it is needed (see [QD89]). Thus we replace A_i with a new variable $A1_{i,i}$, and pipeline it in the direction $(i, j) \rightarrow (i, j + 1)$. A_j will be replaced by the variable $A2_{0,j}$ and pipelined through the direction $(i, j) \rightarrow (i + 1, j)$. B_i and B_j will be replaced in the same way with $B1$ and $B2$ respectively. We obtain the following uniform recurrence equation:

$$\forall i, j :\ 0 \le i \le j;\ i + j \le m + n - 2$$

$$C_{i,j} = \begin{cases} A2_{i,j} * B2_{i,j} & , j = i \quad (3.15) \\ A1_{i,j} * B2_{i,j} + A2_{i,j} * B1_{i,j} & , j = i+1 \quad (3.16) \\ A1_{i,j} * B2_{i,j} + A2_{i,j} * B1_{i,j} + C_{i+1,j-1} & , j > i+1 \quad (3.17) \end{cases}$$

$$A1_{i,j} = \begin{cases} A_i & , j = i \quad (3.18) \\ A1_{i,j-1} & , j > i \quad (3.19) \end{cases}$$

$$B1_{i,j} = \begin{cases} B_i & , j = i \quad (3.20) \\ B1_{i,j-1} & , j > i \quad (3.21) \end{cases}$$

$$A2_{i,j} = \begin{cases} A_j & , i = 0 \quad (3.22) \\ A2_{i-1,j} & , i > 0 \quad (3.23) \end{cases}$$

$$B2_{i,j} = \begin{cases} B_j & , i = 0 \quad (3.24) \\ B2_{i-1,j} & , i > 0 \quad (3.25) \end{cases}$$

Note that equations (3.18), (3.20), (3.22), (3.24) are input equations of the form (3.2).

Now the input A_i appears in input equation (3.18) and (3.22), too. B_i also appears in two input equation. This would mean that we have to input the coefficients of the polynomials A and B twice.

This can be avoided by changing input equation (3.18) with

$$A1_{i,j} = A2_{i,j} \quad , j = i \tag{3.26}$$

In the same way we replace (3.20) by:

$$B1_{i,j} = B2_{i,j} \quad , j = i \tag{3.27}$$

Table 3.1. shows the dependencies of the $SURE$.

Note that the dependencies for $A1$ and $B1$ respectively $A2$ and $B2$ are the same, so in the following we will only reason about $A1$ and $A2$, $B1$ respectively $B2$ can be handled in the same way.

3.5.2 Finding an Adequate Timing Function

According to the method presented in [Fea92] we are looking for affine timing functions with the same linear part for each variable V of the $SURE$ (3.15)-(3.25) of the form $t_V = x * i + y * j + z_V$.

For each dependence of Table 3.1 of the form $V_i(z) \leftarrow V_j(z')$ we are writing the dependency constraint of the form (3.5). We get the following inequalities:

Equation	lhs	rhs	Dependence vector
(3.17)	$C_{i,j}$	$C_{i+1,j-1}$	$(-1,1)$
(3.19)	$A1_{i,j}$	$A1_{i,j-1}$	$(0,1)$
(3.21)	$B1_{i,j}$	$B1_{i,j-1}$	$(0,1)$
(3.23)	$A2_{i,j}$	$A2_{i-1,j}$	$(1,0)$
(3.25)	$B2_{i,j}$	$B2_{i-1,j}$	$(1,0)$
(3.17)	$C_{i,j}$	$A1_{i,j}, A2_{i,j}, B1_{i,j}, B2_{i,j}$	$(0,0)$
(3.26)	$A1_{i,j}$	$A2_{i,j}$	$(0,0)$
(3.27)	$B1_{i,j}$	$B2_{i,j}$	$(0,0)$

Table 3.1: Dependence vectors

$$\begin{array}{lcll} C_{i,j} & \leftarrow & A1_{i,j} & \Rightarrow \ t_C(i,j) > t_{A1}(i,j) \\ C_{i,j} & \leftarrow & A2_{i,j} & \Rightarrow \ t_C(i,j) > t_{A2}(i,j) \\ C_{i,j} & \leftarrow & C_{i+1,j-1} & \Rightarrow \ t_C(i,j) > t_C(i+1,j-1) \\ A1_{i,j} & \leftarrow & A1_{i,j-1} & \Rightarrow \ t_{A1}(i,j) > t_{A1}(i,j-1) \\ A2_{i,j} & \leftarrow & A2_{i-1,j} & \Rightarrow \ t_{A2}(i,j) > t_{A2}(i-1,j) \\ A1_{i,j} & \leftarrow & A2_{i,j} & \Rightarrow \ t_{A1}(i,j) > t_{A2}(i,j) \end{array} \tag{3.28}$$

From the conditions marked with (3.28) and the computation time minimisation condition we get the following system of inequalities:

$$\begin{cases} z_c > z_{A1} \\ z_c > z_{A2} \\ y - x > 0 \\ y > 0 \\ x > 0 \\ z_{A1} > z_{A2} \\ x + y + z_C + z_{A1} + z_{A2} \rightarrow \text{minimal} \end{cases} \tag{3.29}$$

We also need the constraint that the time function is positive on the domain. Then from (3.29) we get the solution:

$$\begin{cases} x = 1 \\ y = 2 \\ z_{A2} = 0 \\ z_{A1} = 1 \\ z_C = 2 \end{cases}$$

The time functions are the following:

$$\begin{array}{lcl} t_C(i,j) & = & i+2j+2 \\ t_{A1}(i,j) & = & t_{B1}(i,j) = i+2j+1 \\ t_{A2}(i,j) & = & t_{B2}(i,j) = i+2j \end{array} \tag{3.30}$$

The common linear part of the time functions is $T = (1, 2)$.

3.5.3 Possible Allocation Functions

Given the timing functions found in section 3.5.2, we are looking for affine allocation functions with the same linear part for each variable V of the $SURE$ (3.15)-(3.25) of the form $p_V = \alpha * i + \beta * j + \gamma_V$. The common linear part of the allocation functions is $P = (\alpha, \beta)$.

The *general constraint* of the form (3.10) is in our case $(\alpha, \beta) \neq c * (1, 2)$, that is

$$\frac{\alpha}{\beta} \neq \frac{1}{2} \tag{3.31}$$

In Table 3.1. one can look for the dependence vector corresponding to a certain variable. That is $(-1, 1), (0, 1)$ and $(1, 0)$ for variables C, $A1$ and $A2$ respectively.

For a variable V and a corresponding dependence vector θ_V, according to definition (3.6) the dataflow-direction is

$$(T * \theta_V, P * \theta_V) = ((1, 2) * \theta_V, (\alpha, \beta) * \theta_V) \ .$$

The *weak conditions* (only for dependencies of the form $V(z) \leftarrow V(z')$) are:

$$\begin{array}{lcllcl} C_{i,j} & \leftarrow & C_{i+1,j-1} & \Rightarrow & |p_C(i,j) > p_C(i+1,j-1)| & \leq & t_C(i,j) - t_C(i+1,j-1) \\ A1_{i,j} & \leftarrow & A1_{i,j-1} & \Rightarrow & |p_{A1}(i,j) - p_{A1}(i,j-1)| & \leq & t_{A1}(i,j) - t_{A1}(i,j-1) \\ A2_{i,j} & \leftarrow & A2_{i-1,j} & \Rightarrow & |p_{A2}(i,j) - p_{A2}(i-1,j)| & \leq & t_{A2}(i,j) - t_{A2}(i-1,j) \end{array} \tag{3.32}$$

The *strong conditions* are:

$$\begin{array}{lcllcl} C_{i,j} & \leftarrow & A1_{i,j} & \Rightarrow & p_C(i,j) - p_{A1}(i,j) & = & \left\lfloor \frac{1}{2}(t_C(i,j) - t_{A1}(i,j)) \right\rfloor \beta \\ C_{i,j} & \leftarrow & A2_{i,j} & \Rightarrow & p_C(i,j) - p_{A2}(i,j) & = & (t_C(i,j) - t_{A2}(i,j))\alpha \\ A1_{i,j} & \leftarrow & A2_{i,j} & \Rightarrow & p_{A1}(i,j) - p_{A2}(i,j) & = & (t_{A1}(i,j) - t_{A2}(i,j))\alpha \end{array} \tag{3.33}$$

From (3.32) we get:

$$\left\{ \begin{array}{lcl} |\beta - \alpha| & \leq & 1 \\ |\beta| & \leq & 2 \\ |\alpha| & \leq & 1 \end{array} \right. \tag{3.34}$$

From (3.33) we get:

$$\begin{cases} \gamma_C - \gamma_{A1} &= 0 \\ \gamma_C - \gamma_{A2} &= \alpha \\ \gamma_{A1} - \gamma_{A2} &= \alpha \end{cases} \tag{3.35}$$

From the conditions (3.31) and (3.34) we get the set of solutions for α and β:

$$(\alpha, \beta) \in \{(-1,-1), (-1,0), (0,-1), (0,0), (0,1), (1,0), (1,1)\} \tag{3.36}$$

In (3.36) the first and the last three solutions are symmetric and the solution $(\alpha, \beta) = (0,0)$ can be excluded because the transformation matrix $\begin{pmatrix} T \\ P \end{pmatrix}$ would be then singular (that means that it would transform some points of D lying on a line into a single point, which is not admitted). Thus we have only three different results:

$$P \in \{(0,1), (1,0), (1,1)\} \tag{3.37}$$

From (3.37) and (3.35) we get three different solutions for adequate allocation functions corresponding to the given timing functions:

$$p_C(i,j) = p_{A1}(i,j) = p_{B1}(i,j) = p_{A2}(i,j) = p_{B2}(i,j) = j \tag{3.38}$$

$$\begin{cases} p_C(i,j) = p_{A1}(i,j) = p_{B1}(i,j) = i \\ p_{A2}(i,j) = p_{B2}(i,j) = i - 1 \end{cases} \tag{3.39}$$

$$\begin{cases} p_C(i,j) = p_{A1}(i,j) = p_{B1}(i,j) = i + j \\ p_{A2}(i,j) = p_{B2}(i,j) = i + j - 1 \end{cases} \tag{3.40}$$

3.5.4 Mappings to Different Systolic Arrays

We apply the time-space transformation onto the *SURE* (3.15)-(3.25) according to the timing functions from (3.30) and allocation functions from (3.39). That is:

$$\begin{array}{lcll} t_C(i,j) &=& i + 2j + 2 & \\ t_{A1}(i,j) &=& t_{B1}(i,j) &= i + 2j + 1 \\ t_{A2}(i,j) &=& t_{B2}(i,j) &= i + 2j \end{array} \qquad \begin{array}{lcll} p_C(i,j) &=& i & \\ p_{A1}(i,j) &=& p_{B1}(i,j) &= i \\ p_{A2}(i,j) &=& p_{B2}(i,j) &= i - 1 \end{array}$$

We have chosen this transformation because this is the one the application of which results in an online array.

The transformed $SURE$:

$$\forall t,p:\ p\geq 0; 3p+2\leq t\leq -p+2(m+n)-2; \frac{t-p}{2}\in\mathcal{Z}$$

$$\overline{A2}_{t-2,p-1} * \overline{B2}_{t-2,p-1} \qquad , t=3p+2 \tag{3.41}$$

$$\overline{C}_{t,p} = \begin{cases} \overline{A1}_{t-1,p} * \overline{B2}_{t-2,p-1} + \overline{A2}_{t-2,p-1} * \overline{B1}_{t-1,p} & , t=3p+4 \quad (3.42) \\ \overline{A1}_{t-1,p} * \overline{B2}_{t-2,p-1} + \overline{A2}_{t-2,p-1} * \overline{B1}_{t-1,p} + \overline{C}_{t-1,p+1} & , t>3p+4 \quad (3.43) \end{cases}$$

$$\overline{A1}_{t,p} = \begin{cases} \overline{A2}_{t-1,p-1} & , t=3p+2 \quad (3.44) \\ \overline{A1}_{t-2,p} & , t>3p+2 \quad (3.45) \end{cases}$$

$$\overline{B1}_{t,p} = \begin{cases} \overline{B2}_{t-1,p-1} & , t=3p+2 \quad (3.46) \\ \overline{B1}_{t-2,p} & , t>3p+2 \quad (3.47) \end{cases}$$

$$\overline{A2}_{t,p} = \begin{cases} A_{\frac{t}{2}} & , p=0 \quad (3.48) \\ \overline{A2}_{t-1,p-1} & , p>0 \quad (3.49) \end{cases}$$

$$\overline{B2}_{t,p} = \begin{cases} B_{\frac{t}{2}} & , p=0 \quad (3.50) \\ \overline{B2}_{t-1,p-1} & , p>0 \quad (3.51) \end{cases}$$

Note that this transformation is not unimodular, for this reason the domain of the system (3.41)-(3.51) is sparse (see the $(t-p)/2\in\mathcal{Z}$ condition). The resulted array can be optimised: by merging two neighbouring PEs we get the online array presented in Sect. 5.6.4.

3.5.5 Other Solutions to the Problem

The form of the $SURE$ used as starting point has a considerable impact on the result of the design. If we have started from the algorithm

$c_j = 0, \forall j, 0 \leq j \leq m+n-2$
for $i = 0$ to $n-1$
    for $j = i$ to $i+m-1$
        $c_j = c_j + a_i * b_{j-i}$

then the $SURE$ for the same problem could have been formulated in the form of (3.52)-(3.5.5), too. There are well known uniformisation techniques (see [QD89]) to deduce the $SURE$ from the given algorithm, but unfortunately they cannot be applied in a fully automatic way. Therefore we consider the system (3.52)-(3.5.5) as starting point to the design process.

Equations:

$$\begin{cases} C_{i,j} &= C_{i-1,j} + B_{i-1,j-1}A_{i,j-1} \\ B_{i,j} &= B_{i-1,j-1} \\ A_{i,j} &= A_{i,j-1} \end{cases} \tag{3.52}$$

where $0 \leq i \leq n-1$, $i \leq j \leq i+m-1$

Input equations:

$$\begin{cases} B_{-1,i} &= b_{i+1}, & -1 \leq i \leq m-2 \\ C_{-1,i} &= 0, & 0 \leq i \leq m-1 \\ C_{i-1,i+m-1} &= 0, & 1 \leq i \leq n-1 \\ A_{i,i-1} &= a_i, & 0 \leq i \leq n-1 \end{cases} \tag{3.53}$$

The results are considered to be the values of the following variables:

$$c_i = \begin{cases} C_{i,i}, & 0 \leq i \leq n-2 \\ C_{n-1,i}, & n-1 \leq i \leq n+m-2 \end{cases}$$

Figure 3.1 presents the dependence graph associated to the $SURE$ (3.52)-(3.5.5), when $n = 3, m = 4$.

Each of the points of the domain $D = \{(i,j) | 0 \leq i \leq 2, i \leq j \leq i+3\}$ corresponds to a computation, while the arrays represent the data dependencies. The placement of the input values can also be read from the figure, however this was determined after the computation of the timing function.

The small dots between the points $(i,j), (i+1, j+1)$ of the domain D indicate a delay, that is the b values need two time steps to move from point (i,j) to $(i+1, j+1)$.

Equation	lhs	rhs	Dependence vector
(3.17)	$C_{i,j}$	$C_{i+1,j-1}$	$(-1,1)$
(3.19)	$A1_{i,j}$	$A1_{i,j-1}$	$(0,1)$
(3.21)	$B1_{i,j}$	$B1_{i,j-1}$	$(0,1)$
(3.23)	$A2_{i,j}$	$A2_{i-1,j}$	$(1,0)$
(3.25)	$B2_{i,j}$	$B2_{i-1,j}$	$(1,0)$

Table 3.2: Dependence vectors

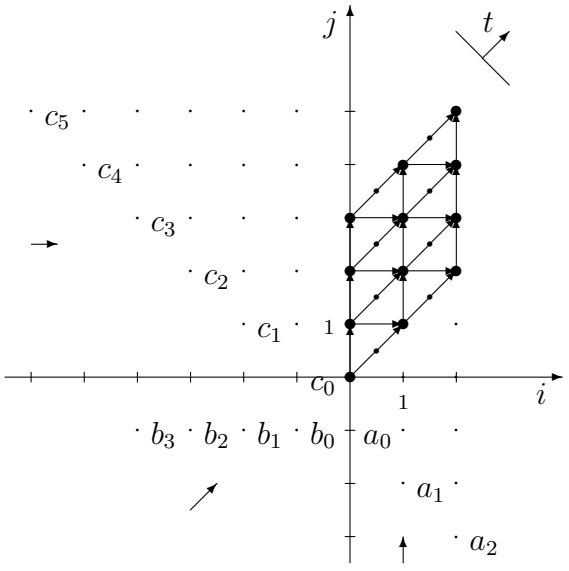


Figure 3.1: Dependence Graph for Polynomial Multiplication $n = 3, m = 4$

Now we have the dependencies shown in Table 3.5.5

If under the assumption of the positivity and dependency constraints we would like to minimise the computation time, we get the time function

$$t(i,j) = i + j \ .$$

We do not give the details of the computations in the description of this particular case, they were performed in the way described in [Fea92].

The general constraint that the allocation function has to satisfy is that its linear part P can not be parallel with the direction T corresponding to the time function (in our case $T = (1,1)$).

For the given time function we get the following allocation functions that satisfy the needed conditions and in addition the each transformation that we obtain is unimodular:

$$\begin{aligned} p(i,j) &= j - i \\ p(i,j) &= i \\ p(i,j) &= j \end{aligned}$$

Choosing the allocation function $p(i,j) = j - i$, after applying the space-time transformation we get the linear systolic array shown on Fig. 3.2.

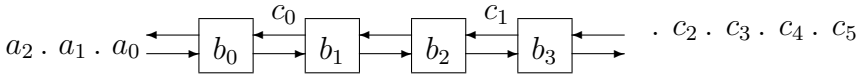


Figure 3.2: Systolic array for polynomial multiplication (the allocation function $p(i,j) = j - i$ was used)

With the allocation function $p(i,j) = i$ we get the systolic array from Fig. 3.3, while with $p(i,j) = j$ the array from Fig. 3.4 is obtained. The placement of the inputs is also depicted on the figures. In case of Fig. 3.3 the structure of the array respectively the transition function is also shown.

The data flow in the arrays of Fig. 3.3 and Fig 3.4 is unidirectional. In the case of the array of Fig. 3.3 the elements of the result appear after n time steps (where n is the number of PEs) as the output of the PE on the right edge of the array, while in the case of the array from Fig. 3.4 the results are computed in the local memories of the PEs.

The systolic array depicted on Fig. 3.2 is bidirectional, but the PEs work alternately and they only perform useful computation at each second time

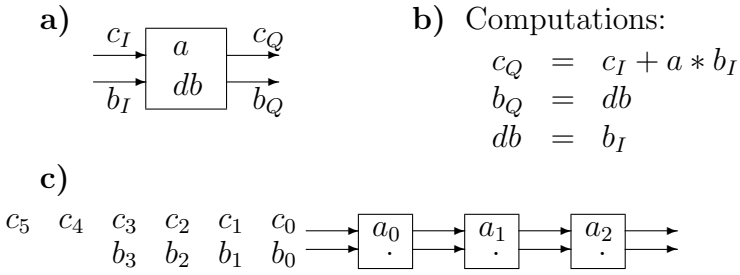


Figure 3.3: Unidirectional systolic array for polynomial multiplication (the allocation function $p(i,j) = i$ was used for the projection) a) structure of a PE, b) transition function, c) structure of the array and placement of the input values

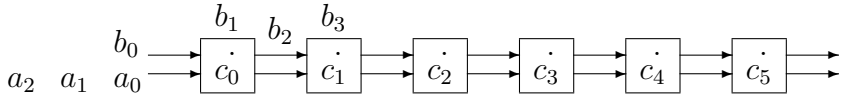


Figure 3.4: Systolic array for polynomial multiplication (the allocation function $p(i,j) = j$ was used)

step. There are some well-known techniques to transform such arrays into a more efficient one. Some ideas are presented in Sect. 5.5.4.

Chapter 4

Mapping Systolic Arrays onto Fixed Size Architectures

Various attempts have been made to overcome the drawbacks of the space-time transformation method.

We have chosen to present in this chapter an alternative method, that starts from an input of the same form as in the case of the space-time transformation methodology (recurrence equations), but the scheduling of the computations is obtained in a much simpler way.

This method is based on the ideas presented in [KRS95], [KRS96] that simplify the tedious task of finding an adequate timing function. The computations are represented by the nodes of a directed graph and the time function is given by the level of the nodes in the modified dependence graph after some empty nodes were introduces according to an algorithm based on rewrite rules, presented in [KRS94].

However, the price for the simplicity is that the size of the problem hs to be fixed in advance, thus the method does not work for parametrised problems.

Again, the method is exemplified with a significative case study: a linear systolic array is generated for the problem of sequences alignment.

4.1 Introduction

A significative disadvantage of the space-time transformation method is that it heavily depends on finding an affine timing function. The problem with finding such a timing function is that one needs to solve a system of linear recurrence equations (see the description form Sect. 3.2), which is generally a tedious and difficult task.

In this section we present an alternative method based on the ideas proposed in [KRS95], [KRS96], which solves the problem of scheduling without the need for solving a system of linear equations. Moreover, this method is not restricted to affine timing functions, but as it is shown, the size of the problem has to be fixed in advance, thus it cannot handle parametrised problems.

The method of Kazerouni, Rajan and Shyamasundar is extended to work with *Directed Linear Recurrence Equations* (DREs), a subclass of linear recurrence equations which properly includes the class off UREs, however in this presentation we will restrict ourselves to problems that can be expressed as UREs of the form (3.1)-(3.2) (see Def 3.1).

The main ideas used in the method presented hereafter are the following:

- It starts from the graphical abstraction of the recurrence equation, the dependence graph (DG) (see Def. 3.3). The edges of the DG are labelled and the input variables are also considered. To avoid confusion, we call this structure *Computation Graph* (CG). The Θ set of dependence vectors (see Def 3.2) is determined and each variablename is associated to the corresponding dependence vector.

- The set of *data-flow channels* is identified, that is the set of maximal directed paths of the labelled *DG*, starting with a node from the input domain D_{inp} and labelled with the same label.

- The labelled dependence graph is transformed into a so called *Modified Dependence Graph* (MDG) by simple, semantics-preserving rules. The level of the nodes in the MDG yields the timing function.

- The *General Systolic Architecture* (GSA) is determined by associating coordinates to the nodes.

- Finally, a *projection scheme* is chosen, which maps the GSA onto a given systolic architecture.

The method is exemplified with an interesting case-study. We describe the design steps of a bidirectional systolic array for sequences comparison.

The method is simple, thus its implementation is also much easier than that of the space-time transformation method. Unfortunately the method also has a major shortcoming: the price for the mentioned advantages is that the size of the problem has to be fixed in advance, thus it can not be applied to parametrised problems.

4.2 Steps of the Design Algorithm

We present in this section the details of the algorithm, step by step.

1st Step (Determination of the labelled DG and the dependencies)

• We determine the Θ set of dependence vectors of the system (3.1)-(3.2) and associate each variable name V_k with the corresponding dependence θ_{V_k} (this is possible if in the computation of V_k the same variablename does not appear twice on the rhs. of the equation).

Formally:

$$\Theta' = \{(V_k, \theta_{V_k}) | V_k \in V,\ \theta_{V_k} \in \Theta\}$$

• We construct the *Computation Graph* (CG), where the nodes are indices of $D \bigcup D_{inp}$, the edges indicate the dependencies and the edges are labelled.

Definition 4.1. (**Computation Graph – CG**)
The *computation graph* of a $SURE$ is a directed graph $CG = G(N, E')$, where
$N = \{z | z \in D \bigcup D_{inp}\}$ is the set of nodes, and
$E' = \{(z', z, V_k) |\ \exists (V_k, \theta_{V_k}) \in \Theta' \ \wedge z' = z - \theta_{V_k}\}$ is the set of directed edges labelled with the variable name V_k.

Note: The dependence vector associated with a variable name X will indicate the direction of the X data-flow through the array.

2nd Step (Data-Flow Channels)

• We determine the *set of Data-Flow Channels* (DFC), where each data-flow channel is a maximal directed path of the CG starting with a node from the input domain D_{inp} and labelled with the same label. The set DFC is determined using the following algorithm:

DFC=∅
for all pairs (z_i^j, V_k), where $(z_i^j \in D_{inp}) \wedge (V_k \in V)$
    if ∃ a path in CG starting with node z_i^j and labelled with V_k, then
        let *newc* be the path of maximal length.
    $DFC = DFC \bigcup \{newc\}$

Note:

- We represent a channel as an ordered list of pairs consisting of nodes, associated with the variable name corresponding to the label of the edges of the path, that is, a channel is represented as a succession of pairs of the form (V_k, z).

- We assumed that in the computation of V_k the same variable does not appear more than once on the rhs. of the equation, which assures that there could not be more then one edges starting from a node and labelled with the same label.

3rd Step (Determine the GSA)

In this step the CG is transformed into the so called *General Systolic Architecture* (GSA). At first the following transformations are performed, in order to find an adequate scheduling for the problem:

• We insert empty nodes into the channels in order to have all the nodes with the same indices on the same level (we consider that the first node of a channel has level 0).

Apply the following rule successively beginning with the nodes of the data-flow channels in DFC at the first level until *MaxDepht(DFC)*:
if (n_c is a node in channel c of DFC) $\wedge$ ($n'_{c'}$ is a node in channel c') $\wedge$ ($Level(n_c) > Level(n'_{c'})$) then
insert *empty* node into the channel c' at level $Level(n_c)$.

Note: This modified structure yields the timing function, that is, the level of a node z in a channel with edges labelled with V_k indicates the time instant when the computation of the variable $V_k(z)$ takes place. Thus the timing function is actually not computed explicitly.

The termination of the algorithm is proven in [KRS96]. Also note that maximal depth of the DFG has to be known, which means that the size of the problem has to be fixed in advance.

• Afterwards we determine the *General Systolic Architecture* (GSA) by associating coordinates with the nodes of the channels of DFC according to the following rules:

- If z is a node of the domain D (we call it *"real node"*), then:
replace it with the pair (z, z)
(that is, a real node is associated with coordinates corresponding to the

index itself)

- *empty nodes* between two real nodes:
  will be interpreted as a delay of one time instant for each of them, so they all will be mapped onto the real node preceding them and will be associated with its coordinate.
  (Note that the delay of a data can be modelled by using memory variables: output = memvar; memvar = f(input))
- Coordinates corresponding to the input nodes and the *empty* nodes between an input node and a real one of a channel labelled with V_k:
  are calculated by subtracting θ_{V_k} from the coordinate of the node following it

Let D^* be the set of coordinates of the GSA.

Note that the $SURE$ and the GSA are computationally equivalent (see the argumentation of [KRS96]).

4th Step (Projection)

The last step is the projection of the GSA onto a systolic architecture.

• We choose a *projection scheme* for the GSA. A projection scheme, that is a *'projection along an axis'* or *'translation along an axis'* is defined as follows:

Definition 4.2. (**Projection Scheme**)[KRS96]

1. *Projection along an axis.*
   For any point $q = (x_1, \ldots, x_n) \in D^*$ the projection along the x_i axis is defined as follows:
   $pr : (x_1, \ldots, x_n) \mapsto (x'_1, \ldots, x'_{n-1})$ such that
   $(\forall\ 1 \leq j < i,\ x'_j = x_j) \wedge (\forall i \leq j < n, x'_j = x_{j+1})$

2. *Translation along an axis.*
   For any point $q = (x_1, \ldots, x_n) \in D^*$ we define the translation along the x_i axis by
   $tr : (x_1, \ldots, x_n) \mapsto (x'_1, \ldots, x'_n)$ such that
   $(\forall\ 1 \leq j < i,\ x'_j = x_j) \wedge (\forall i < j \leq n,\ x'_j = x_j) \wedge (x'_i = x_i \pm x_k) \wedge$
   $(1 \leq k \leq n) \wedge (k \neq i)$.

The chosen projection scheme maps the coordinates of the GSA to coordinates of the processing elements (PE) of a systolic architecture, and it should satisfy the following condition:

- There should be not two distinct variables computed at the same time instant that are mapped onto the same PE.

Note that this last step of the algorithm, namely the choice for the projection scheme is not automatic.

4.3 The Problem of Sequences Comparison - Case Study

In this section we apply the method described in Sect. 4.2 for the problem of optimal alignment of two sequences [NW70], which appears to be particularly interesting for problems in computational biology. We presented this case study in [SC03].

The problem of similarity determination arises in comparing two sequences while allowing certain mismatches between them.

In order to get the similarity between two sequences, we firstly have to align them.

Given two sequences (e.g. DNA), the problem consists in finding the 'best' alignment between them. The problem is largely presented in the literature (according to [AFM03]). The most algorithms used are based on dynamic programming.

The problem is to make the sequences to be of the same size, inserting gaps. The best alignment is one that maximises some scoring function. For instance, we will score $+1$ for each match, -1 for each mismatch and -2 for each gap.

Example:

G	A	$-$	C	G	G	A	T	T	A	G	A	A
G	A	T	C	G	G	A	A	T	A	G	$-$	$-$
1	1	-2	1	1	1	1	-1	1	1	1	-2	2

Total score: $1+1-2+1+1+1+1-1+1+1+1-2-2=2$.

The basic algorithm

Let $s = s_1 s_2 \dots s_m$ and $t = t_1 t_2 \dots t_n$ be the two sequences. The matrix $A(m \times n)$ will contain the items of s along the rows and the items of t along the columns, and each entry $A(i,j)$ corresponds to the optimal alignment of the i^{th} prefix of s with the j^{th} prefix of t.

$$A(i,j) = \max \begin{cases} A(i-1,j) - 2 & \text{align } s(i) \text{ with a gap} \\ A(i,j-1) - 2 & \text{align } t(j) \text{ with a gap} \\ A(i-1,j-1) \pm 1 & \text{align } s(i) \text{ with } t(j), \\ & +1 \text{ for a match,} \\ & -1 \text{ for a mismatch.} \end{cases} \tag{4.1}$$

Note: The value $A(i,0)$ stands for aligning the i^{th} prefix of s with the 0^{th} prefix of t. The optimal score is -2. Analogous, $A(0,j)$ stands for aligning the 0^{th} prefix of s with the j^{th} prefix of t. The optimal score is also -2.

In Fig. 4.1 we indicate the matrix A for scoring alignments of $s = AACG$ and $t = AGG$.

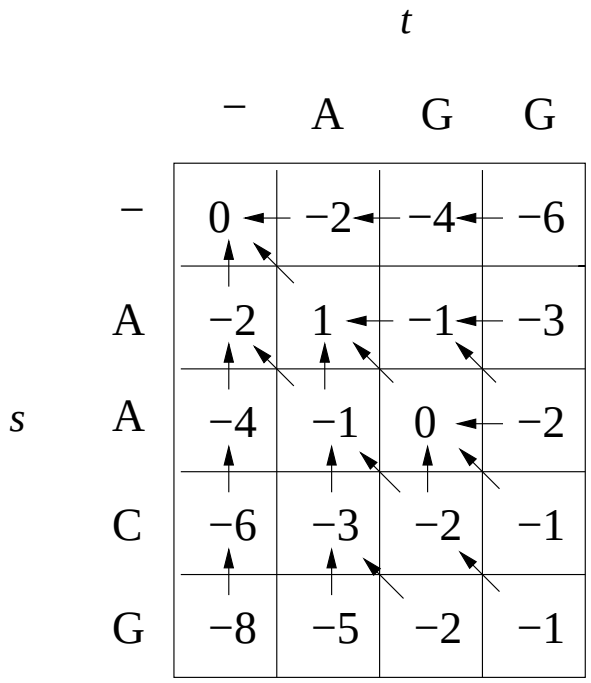


Figure 4.1: Matrix A for scoring alignments of $s = AACG$ and $t = AGG$

The best alignment is given by $A(m,n)$, and we want to know which one is that. In our example, $A(4,3)$ gives -1, which is the score for the best alignments. The arrows in Fig. 4.1 show from which entry in the matrix we derive the maximum score for a given entry.

In this particular case we have three optimal alignments for s and t. These are:

$$\begin{array}{ccc} -AGG & A-GG & AG-G \\ AACG & AACG & AACG \end{array}$$

4.3.1 Generating the Systolic Solutions for the Sequences Alignment Problem

In order to obtain the $SURE$ for the algorithm (4.1) we need to rename some variables, respectively to *normalise* the equation system (all variables should have an index set of the same dimension).

After the changes, the $SURE$ corresponding to the basic algorithm (4.1) has the following form (for the sake of simplicity we use the notation $X_{i,j}$ rather than $X((i,j))$, where X is a variable name and (i,j) an index from the domain of the $SURE$):

$$\begin{aligned}
A_{i,j} &= f_1(A_{i-1,j}-2, B_{i,j-1}-2, f_2(S_{i,j-1}, T_{i-1,j}) + C_{i-1,j-1}) && (4.2)\\
B_{i,j} &= f_1(A_{i-1,j}-2, B_{i,j-1}-2, f_2(S_{i,j-1}, T_{i-1,j}) + C_{i-1,j-1}) && (4.3)\\
C_{i,j} &= f_1(A_{i-1,j}-2, B_{i,j-1}-2, f_2(S_{i,j-1}, T_{i-1,j}) + C_{i-1,j-1}) && (4.4)\\
S_{i,j} &= S_{i,j-1} && (4.5)\\
T_{i,j} &= T_{i-1,j} && (4.6)
\end{aligned}$$

$$i = \overline{1,m},\ j = \overline{1,n},$$

where $f_1(a,b,c) = \mathrm{Max}(a,b,c)$, and $f_2(x,y) = \begin{cases} 1 & , \text{ if } x = y \\ -1 & , \text{ if } x \neq y \end{cases}$

The input equations are the following:

$$\begin{aligned}
A_{0,j} &= -2j, \quad j = \overline{1,n} && (4.7)\\
B_{i,0} &= -2i, \quad i = \overline{1,m} && (4.8)\\
C_{0,0} &= 0 && (4.9)\\
C_{i,0} &= -2i, \quad i = \overline{1,m-1} && (4.10)\\
C_{0,j} &= -2j, \quad j = \overline{1,n-1} && (4.11)\\
S_{i,0} &= s_i, \quad i = \overline{1,m} && (4.12)\\
T_{0,j} &= t_j, \quad j = \overline{1,n} && (4.13)
\end{aligned}$$

• We apply the **1st step** of the algorithm and we get three distinct dependence vectors: $(1,0) = (i,j) - (i-1,j)$, $(0,1) = (i,j) - (i,j-1)$ and $(1,1) = (i,j) - (i-1,j-1)$. After associating them with the variable names we get:

$$\Theta' = \{(A,(1,0)), (B,(0,1)), (C,(1,1)), (S,(0,1)), (T,(1,0))\}$$

• In the **2nd step** we determine the data-flow channels. From the input equations (4.7)-(4.13) we get $3(m+n)-1$ input variables serving as starting nodes for the data-flow channels. (In the example presented in subsection 4.3 $m = 4$ and $n = 3$.)

For example the path starting with node $(0,2)$ and labelled with the variable name C determines the channel
$\{(C,(0,2)),(C,(1,3))\}$,
while the path starting with node $(1,0)$ and labelled with B leads to channel
$\{(B,(1,0)),(B,(1,1)),(B,(1,2)),(B,(1,3))\}$.
Note that the difference between the indices of two adjacent elements in the channel is exactly the dependence vector corresponding to the given variable name (see Θ' determined in the 1st step), $(0,1)$ in case of variable name B.

Other two channels that we will use as example in the presentation of the following steps are
$\{(B,(2,0)),(B,(2,1)),(B,(2,2)),(B,(2,3))\}$ and
$\{(C,(0,1)),(C,(1,2)),(C,(2,3))\}$.

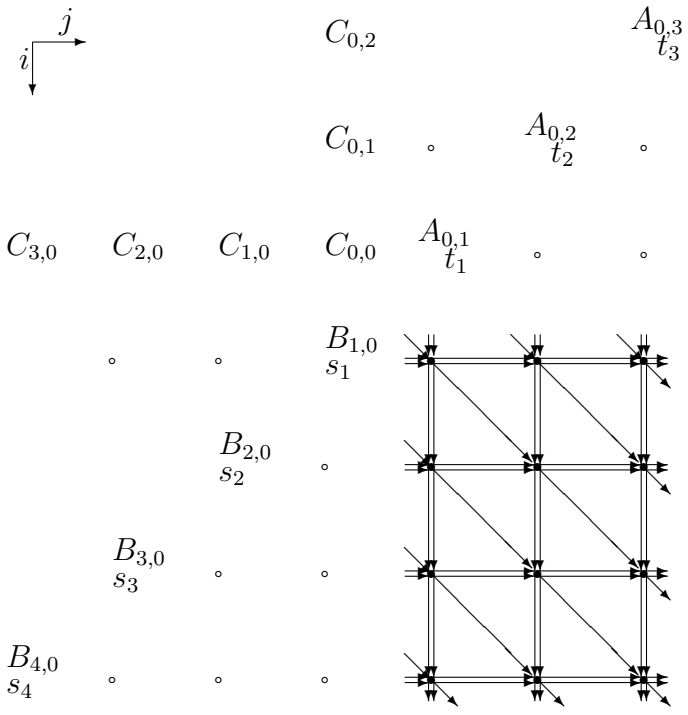


Figure 4.2: Bidimensional systolic array for sequences alignment

• We apply the rule corresponding to the **3rd step** of the algorithm successively beginning with the nodes of the channels on the first level.

In the channel starting with $(C,(0,2))$ the node $(1,3)$ is at the first level, while in the channel starting with $(B,(1,0))$ it is on level 3. After the application of the rule twice to the corresponding channel we get
$\{(C,(0,2)),(empty),(empty),(C,(1,3))\}$.

The channel with head $(B,(1,0))$ will remain unchanged, while the other two channels taken as example will be
$\{(B,(2,0)),(empty),(B,(2,1)),(B,(2,2)),(B,(2,3))\}$ and
$\{(C,(0,1)),(empty),(C,(1,2)),(empty),(C,(2,3))\}$.

Now the GSA is determined simply by associating coordinates to the elements of the channels according to the described rules.
Note that in our example in channel with head $(C,(0,1))$ we have an *empty* node between the two real nodes $(C,(1,2))$ and $(C,(2,3))$. This will be interpreted as a delay of one time instant of the data corresponding to variable name C and moving in direction $(1,1)$. The coordinate of a real node remains the corresponding index itself, while the coordinates for the other elements are calculated with the help of the dependence vectors associated to the corresponding variable names.

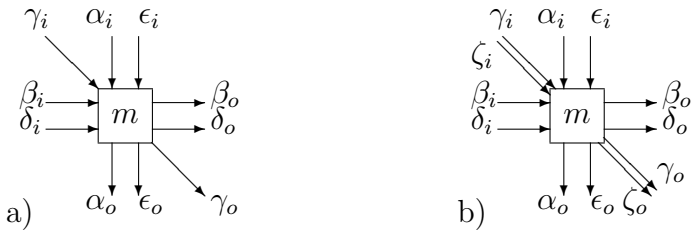
We can see the graphical representation of the GSA obtained as result of this step in Fig. 4.2. As the dimension of the URE system was 2 the obtained GSA can already be interpreted as a working two dimensional systolic architecture without applying any projection scheme from the 4th step.

The computations that take place at each PE in one time instant are deduced from the equations (4.2)-(4.6). The delay element which appears in the channels associated with the variable name C will be modelled by the introduction of a memory variable, as shown in Fig. 4.3. a).

$\alpha_i,\ \beta_i,\ \gamma_i,\ \delta_i,\ \epsilon_i$ are inputs to the PE corresponding to variable names $A,\ B,\ C,\ S$ respectively T.

Concerning the result, we get the score for the best alignment given by $A(m,n)$ (which corresponds in our systolic array to any of $A_{m,n}$, $B_{m,n}$ or $C_{m,n}$) at $\mathrm{PE}_{m,n}$ at time instant $m+n-1$, but actually we are interested not only in getting this final score: in order to reconstruct the best alignment(s) of the two sequences we need the data from the table in Fig. 4.1. More exactly we only need to know the position of the arrows that show from which entry in the matrix we derive the maximum score for a given entry.

We observe that the C values move with a delay of one time instant relative to the other values, which means that the variable $C_{i,j}$ is computed at a certain time instant t, but it is sent to the neighbouring $\mathrm{PE}_{i+1,j+1}$ only at time instant $t+2$. We can send the result in the direction $(1,1)$ of the



Computations:

$$\alpha_o = f_1(\alpha_i - 2, \beta_i - 2, f_2(\delta_i, \epsilon_i) + \gamma_i)$$
$$\beta_o = f_1(\alpha_i - 2, \beta_i - 2, f_2(\delta_i, \epsilon_i) + \gamma_i)$$
$$\gamma_o = m_{old}$$
$$\delta_o = \delta_i$$
$$\epsilon_o = \epsilon_i$$
$$m_{new} = f_1(\alpha_i - 2, \beta_i - 2, f_2(\delta_i, \epsilon_i) + \gamma_i)$$
$$\zeta_o = f_3(\alpha_i, \beta_i, \gamma_i, \delta_i, \epsilon_i, \zeta_i)$$

Figure 4.3: Bidimensional systolic structure for the problem of sequences alignment. Computations of a PE.

main diagonals on a new communication channel as shown in Fig. 4.3. b), where the function that computes the result is given by

$$f_3(a, b, c, x, y, z) = \begin{cases} z & , \text{ if } a = Null \\ \text{IndLstOfMax}(a - 2, b - 2, f_2(x, y) + c) & , \text{ if } a \neq Null \end{cases}$$

The function $\text{IndLstOfMax}(x_1, x_2, x_3)$ returns a list of indices of the maximal elements of the ordered list $\{x_1, x_2, x_3\}$
(e. g. $\text{IndLstOfMax}(4, 6, 9) = \{3\}$, $\text{IndLstOfMax}(4, 9, 9) = \{2, 3\}$).
We will get the results through the processing elements $\text{PE}_{m,j}$, $j = \overline{1, n}$ and $\text{PE}_{i,n}$, $i = \overline{1, m-1}$ in the order illustrated in Fig. 4.4. a), so that after $m+n$ steps we get the results for all the entries of the matrix. We have associated to each entry a subset of $\{1, 2, 3\}$ that shows from which neighbouring entry (or entries) we derive the maximum score (see Fig. 4.4. b)). The direction denoted by 1 should be interpreted as "align $s(i)$ with a gap", 2 means "align $t(j)$ with a gap", while 3 stands for "align $s(i)$ with $t(j)$".

We also observe that at a given time instant only the PEs from one secondary diagonal of the array are performing computations. In this case we can improve the efficiency of the array by using it for the computation of the best alignment for more sequences one after the other.

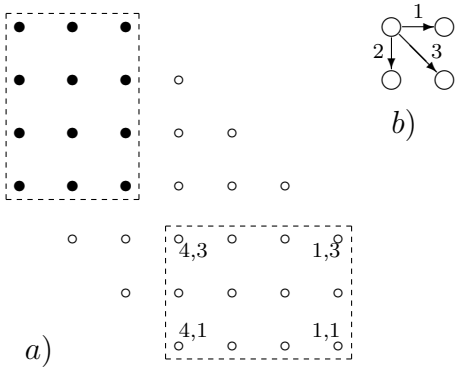


Figure 4.4: Getting the result for the sequences alignment problem

• Finally, we can also apply the **4th step** of the design algorithm choosing as projection scheme the translation along the j-axis followed by the projection along the i-axis: $prscheme : (i, j) \mapsto j - i$. This projection leads to a linear systolic architecture with $m+n-1$ processing elements, indexed with $1-m$, $2-m$, ..., $n-1$ (see indices above the PEs in Fig. 4.5.).

Figure 4.5. illustrates the structure of the array, respectively the placement of the input data at the beginning (note: '$*$' stands for the place of the C values written under the PEs).

The elements of the chains labelled with the variable C are all mapped onto the same PE (as $prscheme(i, j) = prscheme(i + 1, j + 1) = j - i$), thus the communication is replaced by a simple memory assignment.

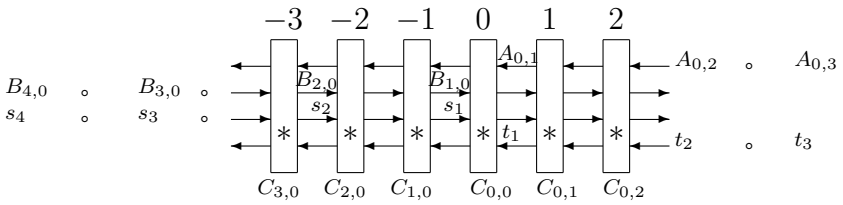


Figure 4.5: Linear systolic array for sequences alignment

Figure 4.6. a) shows the computations performed in a PE. Note that instead of γ_i and γ_o (as in the case of the bidimensional array) we have $m2$,

i.e. the C values will be evaluated in the local memories of the PEs of the array.

a) b)

$$\alpha_o = f_1(\alpha_i - 2, \beta_i - 2, f_2(\delta_i, \epsilon_i) + m2_{old})$$
$$\beta_o = f_1(\alpha_i - 2, \beta_i - 2, f_2(\delta_i, \epsilon_i) + m2_{old})$$
$$m1_{new} = f_1(\alpha_i - 2, \beta_i - 2, f_2(\delta_i, \epsilon_i) + m2_{old})$$
$$m2_{new} = m1_{old}$$
$$\delta_o = \delta_i$$
$$\epsilon_o = \epsilon_i$$
$$m3_{new} = \text{IndLstOfMax}(\alpha_i - 2, \beta_i - 2, f_2(\delta_i, \epsilon_i) + m2_{old})$$

Figure 4.6: Computations of a PE in the linear array for sequences comparison

Again, if we are interested in the reconstruction of the table from Fig. 4.1. one more computation is needed, as shown on Fig. 4.6. b).

In comparison with the bidimensional array, where only the PEs from the secondary diagonal were performing computations at a time instant, here every second PE is working at each time instant, when the array is loaded.

Chapter 5

Functional-Based Systolic Array Design

We introduce a *functional view* (or inductive view) of systolic arrays: a systolic array is composed of a *head processor* and an identical *tail array* of a smaller size. By exploiting the similarity between the inductive structure of a systolic array and the inductive decomposition of the argument by a functional program, it is possible to develop an elegant and efficient method for the automatic synthesis of systolic arrays.

After a detailed presentation of the formal background used, we study the functioning of different systolic array-types.

By formal analysis, the structure of the functions which can be realised by arrays having certain properties is identified. Then, by equational rewriting, the expression of the list function which must be realised is transformed into an expression having the required structure. The resulting expression reveals the scalar function which must be implemented by each individual processor.

The utility and efficiency of the design method is demonstrated through examples and representative case studies.

5.1 Introduction

Most of the systolic array design methods available in the literature (see a short survey presented in Chap. 2) follow an *iterative view* of systolic arrays (and systolic computations): the arrays (and the computations) are represented as [multidimensional] matrices of a certain size (see the space-time transformation methodology described in Chap. 3).This leads to complex operations over the multidimensional index space, and in fact to many repetitions in the synthesis process. Other methods that imply more simple computations only work for a fixed size (see the design method presented in Chap. 4).

In this chapter we introduce a *functional view* (or inductive view): an infinite systolic array is composed of a *head processor* and an identical *tail array*. Similarly, functional programs for list operations describe how to compute the head and the tail of the result in function of the head and the tail of the argument. By exploiting this similarity, we will show that in some cases the synthesis problem can be solved by [essentially] rewriting of the functional programs.

5.1.1 Functional View of Systolic Arrays

Informally, a linear systolic array with n processing elements (PEs) can be seen as a device, that is composed of a head processor (PE_0), connected to a tail-array, which is an identical array of size $n-1$, as shown on Fig. 5.1.

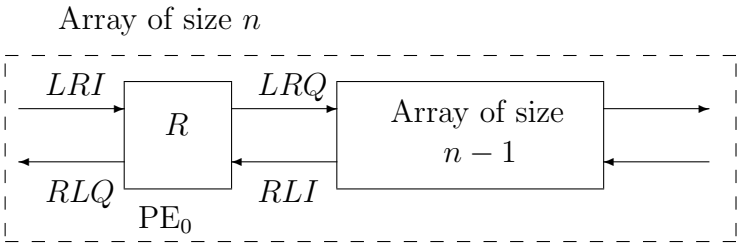


Figure 5.1: Informal view of a linear systolic array

The arrows indicate the direction of the data-flow, from left to right (LR) or from right to left (RL). The letter I stands for *input channels*, Q indicates the *output channels* and R stands for the *internal state registers* (also called local memory).

At each time step the PEs update their internal state (the values of the output channels, respectively that of the internal registers) in function of the input, respectively the value of the internal state registers in the previous time step. The computations performed by a PE are given by the so called *transition function*.

The global input is fed step by step into the array through the input channels of the PEs on the edge, while the result appears at one or more output channels of the marginal PEs (in some cases the result may be computed in the internal state registers).

Note that the LRQ output of PE_0 is the input of the array of size $n-1$, and the output of this array in right–to–left direction is input to PE_0, denoted RLI.

When all the data advance in the same direction we are talking about *unidirectional array*s. When there are data channels in both (left-to-right and right-to-left) directions, then the array is a *bidirectional array*.

In the subsequent sections we will study the behaviour of different types of linear systolic arrays. In Sect. 5.4 we study the behaviour of arrays with unidirectional data-flow, while in Sect. 5.5 and Sect. 5.6 we present the arrays with bidirectional data-flow.

Because the building blocks of systolic arrays are [identical] PEs, in Sect. 5.3 we start with the presentation of the most simple PE-structure, and proceed towards more complex ones. We detect the class of functions that can be realised by a certain type of PE. We will make use of this knowledge in the design process for systolic arrays. In some cases the design of a systolic array can be reduced to the design of a single PE.

The goal of the formal analysis is to find a recursive description for the formal process that characterises the functioning of a certain type of array.

This logic can be also used backwards: if we can describe a problem in such a recursive form, that fits to one (or more) of the recursive descriptions characterizing a certain type of array, we can easily map the problem to that type of array.

5.2 Formal Background

In this section we introduce the notations that we use for the well known notions of lists, programs, and systolic arrays. We define the class of functions that are candidate for the transition function of a PE and analyze their properties.

5.2.1 Scalars and Lists

Both the systolic arrays and the functional programs which we consider act upon (finite or infinite) lists of fixed-size objects.

- **Scalars and scalar types:**
  A fixed-size object is an object of a *scalar type*: A scalar type is an elementary type or a fixed-size tuple of scalar types. An elementary type (such as a finite set of symbols or a fixed-precision number type) can have only a finite number of instantiations. A fixed-size object will be called a *scalar*.

- **Lists and list types:**
  A *list type* over a certain scalar type characterises all the tuples (finite and infinite) of objects of that scalar type. A *list* is an object of a certain list type.

  We denote the list type over the scalar type st by $\langle st \rangle$.

- **Length of a list:**
  The length of a list X is denoted by $\|X\|$ and it is ∞ if the list is infinite.

- **List notation:**
  We denote by X_i the infinite list $\langle x_i, x_{i+1}, x_{i+2} \ldots \rangle$ (note that i can also be negative). X stands for X_0. $X_{n,n+m}$ (where $n \in \mathbb{Z}$ and $m \in \mathbb{N}$) denotes the finite list having $m+1$ elements: $\langle x_n, x_{n+1}, \ldots x_{n+m} \rangle$.

  We will denote by a^n the list of n elements all equal to a and by a^∞ the infinite constant list with all elements equal to a.

- **Head and Tail function:**
  For any list $X = \langle x_0, x_1, \ldots, x_n, \ldots \rangle$, we denote by $H[X] = x_0$ the *head* of it, and by $T[X] = \langle x_1, \ldots, x_n, \ldots \rangle$ the *tail* of it.

  The k^{th} tail of X: $T_k[X] = \langle x_k, x_{k+1}, \ldots, x_n, \ldots \rangle$ is obtained by iterating T k times and removes the first k elements of X, when k is positive.

By convention, $T_0[X] = X$, and note that $T_1 = T$. We extend the definition of the tail function of order k (T_k) also for negative k numbers: $T_{-1}[X_i] = X_{i-1}$, which means that a new value (x_{i-1}) is inserted in the front of the list. If this value is not known, then a "blank" value will be inserted. T_k, when $k < 0$ is obtained by iterating T_{-1} $|k|$ times.

The k^{th} head of X is $H_k[X] = H[T_k[X]]$ and gives the $(k+1)^{\text{th}}$ element of X (thus $H_0 = H$).

- **Prefix of order n:**
  The *prefix* of order n of a list is $P_n[X] = \langle x_0, \ldots, x_{n-1} \rangle = X_{0,n-1}$, that is, it selects the first n elements of the list.

- **Concatenation of lists:**
  The *concatenation* of two lists is denoted by "$\frown$":

$$\langle a_0, a_1, \ldots, a_k \rangle \frown X = \langle a_0, a_1, \ldots, a_k, x_0, x_1, \ldots \rangle.$$

  The first operand must be finite, but the second may also be infinite. We also use "$\stackrel{\bullet}{\frown}$" for *prepending* a scalar to a (finite or infinite) list:

$$a \stackrel{\bullet}{\frown} X = \langle a \rangle \frown X.$$

Since in practice one actually uses only finite lists, we consider here only lists having a finite number of "interesting" values. Namely, we use (as in the theory of cellular automata) a special *quiescent symbol* "$" (which belongs to all scalar types) in order to encode the "blank" values. Thus, an infinite list will start to have only blank values after a certain finite number of elements. We also allow a list to start with a certain number of blanks, but we will not allow "$" to be arbitrarily interspersed among other elements. In case of *sparse lists* the occurencies of "$" follow a regular pattern.

- **Sparse lists:**
  We define the term *sparse list* in a functional way, with the help of the list function $Sparse_k$, where the parameter k ($k \in \mathbb{N}$) indicates the frequency of the "interesting" data:

$$Sparse_k[X] = H[X] \stackrel{\bullet}{\frown} (\$^k \frown Sparse_k[T[X]])$$

  For brevity we denote $Sparse_k[X]$ by $X_{\$k}$.
  $X_{\$1} = X_{\$} = \langle x_0, \$, x_1, \$, \ldots \rangle$. By convention $X_{\$0} = X$.

Properties:

$$
\begin{array}{lcl}
T\ Sparse_k & = & \$^k \dot{\smile} Sparse_k\ T \\
H\ Sparse_k & = & H \\
T_m\ Sparse_k & = & Sparse_k\ T_{\frac{m}{k+1}}\ ,\ \text{if } (k+1)|m \\
\text{in particular } (k=1): & & \\
T_m\ Sparse_1 & = & \begin{cases} Sparse_1\ T_{\frac{m}{2}}\ , & \text{if } m \text{ is even} \\ \$ \dot{\smile} Sparse_1\ T_{\frac{m+1}{2}}\ , & \text{if } m \text{ is odd} \end{cases}
\end{array}
$$

- **k-step lists:**
  k-step lists are defined with the help of the $Step_k$ function in the following way:

  $$Step_k[X] = H[X] \dot{\smile} Step_k[T_k[X]],$$

  where $k \in \mathbb{N}, k \geq 1$.

  That is, the function $Step_k$ selects every k^{th} element of a list, starting with the first one: $Step_k[X] = \langle x_0, x_k, x_{2k}, x_{3k}, \ldots \rangle$.

  For brevity we denote $Step_k[X]$ by $X_{\{k\}}$. Note that $X_{\{1\}} = X$

  Properties:

  $$
  \begin{array}{lcll}
  T\ Step_k & = & Step_k\ T_k & \\
  H\ Step_k & = & H & \\
  T_m\ Step_k & = & Step_k\ T_{m*k} & \\
  Step_k\ T_m & = & T_{\frac{m}{k}}\ Step_k, & \text{if } k|m
  \end{array}
  $$

  $Step_k Sparse_{k-1} X = X \neq Sparse_{k-1} Step_k X$, for $k \geq 1$
  in particular :
  $X_{\$\,\{2\}} = X \neq X_{\{2\}\,\$}$

5.2.2 Functions

We are operating with scalar-, list- and mixed type functions. Hereafter we present the class of functions used and some useful properties characterising these functions.

- **Scalar functions:**
  Functions from scalar types to scalar types will be called *scalar functions*. Informally, scalar functions can be computed in constant time.

  Ex.: $f : \{0,1\}, \{0,1\} \rightarrow \{0,1,10\}, f[a,b] = a \dot{\dot{+}} b$,
  where " $\dot{\dot{+}}$ " denotes the addition of two binary digits.

- **List functions:**
  Functions from list types to list types will be called *list functions*. We will consider *only list functions acting upon infinite lists and producing infinite lists*.

  Ex.: $F : \langle\{0,1\}\rangle, \langle\{0,1\}\rangle \to \langle\{0,1\}\rangle$, $F[A,B] = A \wedge B$,
  where "$\wedge$" represents the bitwise logical "and" operation applied to the corresponding elements of A and B.

  List functions are defined in a functional way:

$$F[a \overset{\cdot}{\smile} A, b \overset{\cdot}{\smile} B] = (a \overset{\cdot\cdot}{\wedge} b) \overset{\cdot}{\smile} F[A,B] \ ,$$

  where $\overset{\cdot\cdot}{\wedge}$ is the logical "and" operation applied on bits (scalars from the set $\{0,1\}$).

  We use the following convention: if *op* is an operation on two lists, then we denote by $\dot{op}$ the corresponding operation between a scalar and a list (this kind of operation is usually also defined in a functional way), while $\dot{\dot{op}}$ stands for an operation between two scalars.

- **Mixed type functions:**
  Functions having both, scalar and list type arguments and producing [scalar and] list type results are *mixed type functions*.

  Ex.: $F : \{0..9\}, \langle\{0..9\}\rangle \to \langle\{0..9\}\rangle$, $F[c,A] = c \overset{\cdot}{+}\vec{} \ A$,
  where "$\overset{\cdot}{+}\vec{}$" denotes the addition with carry propagation. That is

$$F[c, a \overset{\cdot}{\smile} A] = (c \overset{\cdot\cdot}{+} a)_{\mathrm{mod}10} \overset{\cdot}{\smile} F\left[\left\lfloor (c \overset{\cdot\cdot}{+} a)/10 \right\rfloor, A\right]$$

We will also assume that our scalar functions produce blanks when applied to blanks, and that our mixed type functions or list functions are producing [blanks and] lists of blanks when applied to [blanks and] lists of blanks.

By convention we will denote scalar functions by lowercase letters and list- or mixed type functions by capital letters, unless we want to indicate that the list function F is the *transitive extension* of the scalar function f (see Def. 5.6). In this case we will also use the notation $\vec{f}$ for F.

Since we have in mind concrete computations over lists, it is reasonable to consider those list functions F whose values can be computed in an incremental fashion: the values of any finite prefix of $F[X]$ can always be computed from some finite prefix of X.

Definition 5.1. Regular function
A function F on infinite lists is called *regular* iff, for any natural n, there exists a natural m, such that for any list $X_{0,n+m-1}$ of length $n+m$, and for any infinite lists Y, Y':

$$P_n[F[X_{0,n+m-1} \frown Y]] = P_n[F[X_{0,n+m-1} \frown Y']] \ .$$

The minimal m as above is called the *regularity index* of F, denoted $r_F[n] = m$.

Definition 5.2. k–look–ahead function
A function whose regularity index is constant is called *look–ahead* function. The constant $r_F[n] = k$ is called the *look–ahead* index.
We also use the term *k–look–ahead* function for a look–ahead function whose look–ahead index is k.

We have a special interest towards functions with a look–ahead index equal to 0.

Definition 5.3. Online function
A 0–look–ahead function is called *online* function.
That is, the n–th value of an online function depends only on the first n elements of the input.

Note: for an online function, the regularity condition for $n = 0$ (note that the head function gives exactly the prefix of order 0) is:

$$H[F[x \stackrel{\cdot}{\frown} X]] = H[F[x \stackrel{\cdot}{\frown} X']] \ ,$$

thus one may define the function

$$F_H[x] = H[F[x \stackrel{\cdot}{\frown} \$^\infty]] \ .$$

Definition 5.4. Scalar projection of an online function
Let $F : \langle L_1 \rangle \to \langle L_2 \rangle$ be an online function.
The scalar function $F_H : L_1 \to L_2$, such that

$$F_H[x] = H[F[x \stackrel{\cdot}{\frown} \$^\infty]] \ .$$

is called the *scalar projection* of the online function F,

Obviously:

$$F_H[x] = H[F[x \stackrel{\cdot}{\frown} X]] \text{ for any } X, \text{ and}$$
$$F[x \stackrel{\cdot}{\frown} X] = F_H[x] \stackrel{\cdot}{\frown} T[F[x \stackrel{\cdot}{\frown} X]] \ .$$

However, one must take care that only online functions have a scalar projection.
A special role will be played by list functions which commute with T. We will call these functions *transitive*.

Definition 5.5. Transitive function
A list function F is called *transitive* if it commutes with the tail function:

$$F[T[X]] = T[F[X]] \text{ or simply } \mathrm{FT} = \mathrm{TF} \ .$$

(Note that we use square brackets for function application, moreover we will sometimes omit them when the context is sufficiently clear.)
For online transitive functions we have:

$$F[x \dot{\smile} X] = F_H[x] \dot{\smile} F[X] \ ,$$

thus the function F is "constructed" by its scalar projection F_H.

Definition 5.6. Transitive extension of a scalar function
Let f be a scalar function. We call the *transitive extension* of f the list function denoted by $\vec{f}$, constructed in the following way

$$\vec{f}[x \dot{\smile} X] = f[x] \dot{\smile} \vec{f}[X] \ .$$

Note: the syntactic restriction to one argument (and one value) is not essential. Indeed, a multiple scalar can be assigned a new scalar type, and a multiple list can be seen as single list by transposition:

$$\langle x \dot{\smile} X,\ y \dot{\smile} Y, \ldots \rangle^T = \langle x, y, \ldots \rangle^T \dot{\smile} \langle X, Y, \ldots \rangle^T .$$

Functions with mixed-type (scalar and list) argument and/or mixed-type value reduce to functions taking one scalar and one list and producing one scalar and one list.

In the sequel we will use sometimes functions having multiple arguments, however these are understood to be of the form exhibited above. That is, if several scalars occur as arguments, the function is assumed to have only one scalar argument, which is the tuple of those scalars. Similarly, if several lists occur as arguments, the function is assumed to have only one list argument, which is the transposed of the tuple of those list arguments.

Note: one can easily verify that if $W = \langle X^1, X^2, \ldots X^n \rangle^T$ is a multiple list composed of X^1, X^2, ..., X^n then the selector functions that return the corresponding component, $F_{X^i}[W] = X^i$, $1 \leq i \leq n$, are online transitive functions.

Online and mixed-type functions

The mixed-type functions that we will consider can also be associated to some special online transitive list functions.
Given an initial value r^0, and an infinite list X let us consider the list $R = \langle r_0, r_1, \ldots \rangle$ generated by a function f_r in the following way:

$$r_0 = r^0, \tag{5.1}$$
$$r_i = f_r[r_{i-1}, x_{i-1}], \forall i \geq 1 \tag{5.2}$$

R can also be described in a functional way. Let F_R be a mixed-type function and f_r a scalar function such that

$$F_R[r, x \dot{\smile} X] = f_r[r, x] \dot{\smile} F_R[f_r[r, x], X] \ .$$

Then

$$R = r_0 \dot{\smile} F_R[r_0, X]$$

If we apply an online transitive function F on an argument composed of R and X, then it has the following property:

$$F[r_0 \dot{\smile} R_1, x_0 \dot{\smile} X_1] = F_H[r_0, x_0] \dot{\smile} F[R_1, X_1] \ , \tag{5.3}$$

where F_H is the scalar projection of F and R is generated in the way described in (5.1)–(5.2).

In this case (5.3) can be described in a more concise way with the help of the mixed type function F', that is defined in the following way:

$$F'[r, x \dot{\smile} X] = f[r, x] \dot{\smile} F'[f_r[r, x], X] \ , \tag{5.4}$$

where $f = F_H$ and the function $F'[r^0, X]$ with mixed type argument, respectively the list function $F[R, X]$ are computing the same result.

$$F'[r^0, X] = F[R, X] \ .$$

The reason for using this kind of mixed-type functions is explained is Sect. 5.3.

5.2.3 Properties of Online Transitive Functions

In this section we describe some useful properties of online transitive functions.
From the considerations in the previous section follows:

Property 5.1. An online function is transitive if and only if it is a transitive extension of its scalar projection.

Proposition 5.1. The composition of online transitive functions is also online and transitive.

Proof: (... in fact just a simple verification...)
Let $F : \langle L_1 \rangle \rightarrow \langle L_2 \rangle$ and $G : \langle L_2 \rangle \rightarrow \langle L_3 \rangle$ be two online transitive functions. Let us denote the composition of G on F by F', $F' : \langle L_1 \rangle \rightarrow \langle L_3 \rangle$.

F is online, that is for any natural n and for any infinite list Y and Y'

$$P_n[F[X_{0,n-1} \frown Y]] = P_n[F[X_{0,n-1} \frown Y']] \ . \tag{5.5}$$

The same holds for G.

We want to prove that the composite function $F' = G \circ F$ is also an online function, that is, for any natural n and any infinite list Y and Y'

$$P_n[F'[X_{0,n-1} \frown Y]] = P_n[F'[X_{0,n-1} \frown Y']] \ . \tag{5.6}$$

Indeed, because G is online, the n^{th} prefix of $G[F[X_{0,n-1} \frown Y]]$ only depends on the first n elements of the argument of G. That is,

$$P_n[G[F[X_{0,n-1} \frown Y]]] = P_n[G[P_n[F[X_{0,n-1} \frown Y]] \frown Y'']] \ ,$$

for any infinite list Y'' (let us take the infinite list of blank values, $\$^\infty$).

According to (5.5) the lhs. of (5.6) is equal to

$$P_n[G[P_n[F[X_{0,n-1} \frown Y'] \frown \$^\infty]] \ ,$$

for any infinite list Y' (we can consider again the infinite list $\$^\infty$), then the rhs. of (5.6) can be transformed into the same expression, using the same logic for any infinite list Y and Y', which means that the composite function is also online.

According to Prop. 5.1 F and G are transitive extensions of their scalar projections, thus we have

$$F[x \dot\frown X] = H_F[x] \dot\frown F[X] \tag{5.7}$$

and

$$G[y \dot\frown Y] = H_G[y] \dot\frown G[Y] \tag{5.8}$$

Then we have

$$\begin{array}{lll} F'[x \dot\frown X] & = & G[F[x \dot\frown X]] \overset{(5.7)}{=} \\ = G[H_F[x] \dot\frown F[X]] & \overset{(5.8)}{=} & H_G[H_F[x]] \dot\frown G[F[X]] \ . \end{array}$$

It follows that $F'[x \dot{\smile} X] = H_{F'}[x] \dot{\smile} F'[X]$, where $H_{F'} = H_G \circ H_F$, that is F' is the transitive extension of its scalar projection, consequently it is transitive.

Example: Let X be an infinite list having three components (of the same scalar type), $X = \langle A, B, C \rangle^T$ and the list function F, defined in the following way

$$F[X] = F_A[X] * F_B[X] + F_C[X] \ , \tag{5.9}$$

where

- F_A, F_B and F_C are the functions that select the corresponding component of X: $F_A[X] = A$, $F_B[X] = B$ and $F_C[X] = C$.
- the addition "+" of two lists is defined as:

$$(a \dot{\smile} A) + (b \dot{\smile} B) = (a \ddot{+} b) \dot{\smile} (A + B) \ .$$

- the product "$*$" of two list is considered to be the dot product, defined as:

$$(a \dot{\smile} A) * (b \dot{\smile} B) = (a \ddot{*} b) \dot{\smile} (A * B) \ .$$

The selector functions F_A, F_B and F_C are online transitive functions. From the definition of "+" and "$*$" follows that they are also online and transitive functions, consequently F is also an online transitive function.

In the sequel we will omit the explicit use of the selector functions for the sake of a more simple and concise notation. In our example we can write instead of $F[X]$:

$$F[A, B, C] = A * B + C \ ,$$

which is understood to be of the form (5.9).

Online Transitive Functions and the Sparse Function

Proposition 5.2. Let F be an online transitive function. Then

$$(F[X])_\$ = F[X_\$] \ .$$

Verification:
We can easily verify Prop. 5.2 by showing that i^{th} head of the *lhs* respectively the *rhs* is equal $\forall i \geq 0$.

Proposition 5.3. Let F' be a mixed type function having property (5.4). Then

$$(F'[r, X])_\$ = F'[r, X_\$] \ .$$

Verification:
We will verify Prop. 5.3 by showing that the i^{th} head of the *lhs* respectively the *rhs* is equal $\forall i \geq 0$.
We assume, that i is even. For the *lhs* we have

$$H_i[(F'[r,X])_\$] = HT_i[(F'[r,X])_\$] = H[(T_{\frac{i}{2}}[F'[r,X]])_\$] =$$
$$= H[T_{\frac{i}{2}}[F'[r,X]]] = H_{\frac{i}{2}}[F'[r,X]] \ .$$

Let F be the corresponding online transitive function of the form (5.3) such that $F'[r,X] = F[R,X]$.
Then we have

$$H_{\frac{i}{2}}[F'[r,X]] = H_{\frac{i}{2}}[F[R,X]] = H[T_{\frac{i}{2}}[F[R,X]]] =$$
$$= H[F[T_{\frac{i}{2}}R, T_{\frac{i}{2}}X]] = F_H[r_{\frac{i}{2}}, x_{\frac{i}{2}}] \ .$$

For the *rhs* let us consider again the corresponding online transitive function F of the form (5.3) such that $F'[r,X_\$] = F[R',X_\$]$.
We have

$$H_i[F'[r,X_\$]] = H[T_i[F'[r,X_\$]]] = H[T_i[F[R',X_\$]]] = H[R'_i, X_{\frac{i}{2}}] = F_H[r'_i, x_{\frac{i}{2}}] \ .$$

Thus the problem reduces to showing that $r'_i = r_{\frac{i}{2}}$.

If F_R is defined as

$$F_R[r, x \dot{\smile} X] = f_r[r,x] \dot{\smile} F_R[f_r[r,x], X] \ ,$$

then

$$R = r_0 \dot{\smile} F_R[r_0, X]$$

and

$$R' = r_0 \dot{\smile} F_R[r_0, X_\$] \ .$$

Because $f_r[r,\$] = r$, it follows that

$$\begin{array}{rcl} R' & = & r_0 \dot{\smile} F_R[r_0, X_\$] = r_0 \dot{\smile} (f_r[r_0,x_0] \dot{\smile} F_R[r_1, \$ \dot{\smile} X_{1\$}]) = \\ & = & \langle r_0, r_1 \rangle \smile (f_r[r_1,\$] \dot{\smile} F_R[r_1, X_{1\$}]) = \ldots = \\ & = & \langle r_0, r_1, r_1, r_2, r_2, \ldots, r_i, r_i \rangle \smile F_R[r_i, X_{i\$}] \Rightarrow \\ \Rightarrow & & R'_{\{2\}} = R \end{array}$$

Thus

$$r_{\frac{i}{2}} = H_{\frac{i}{2}}[R] = HT_{\frac{i}{2}}[R'_{\{2\}}] = H[R'_{i\{2\}}] = H[R'_i] = r'_i \ .$$

We can verify in the same manner that the property also holds for the case when i is odd.

5.2.4 Functional Programs

The input to our design method is given in form of functional programs. Hereafter we specify what a functional program consists in.

- *A program describing a scalar function f* is an expression involving elementary scalar functions (considered as "known"):

$$f[x] = E \ .$$

- *A program describing a list function F* must indicate how to compute the head and the tail of the result, by starting from the head and the tail of the argument.

$$F[x \overset{\cdot}{\smile} X] = \mathcal{E}[x, X] \ ,$$

where $\mathcal{E}$ is a mixed (scalar–list) expression involving already known functions, but also F. The simplest definitions have the shape:

$$F[x \overset{\cdot}{\smile} X] = E[x] \overset{\cdot}{\smile} \mathcal{E}[X] \ ,$$

where E is a scalar expression and $\mathcal{E}$ is a list expression.

For parametrised list functions a recursive description with respect to the size parameter should also be given.

- The most general case for a *functional program describing a mixed type function* is:

$$F[x, y \overset{\cdot}{\smile} Y] = \langle E[x, y], \mathcal{E}'[x, y, Y] \rangle \ ,$$

where E is a scalar expression and $\mathcal{E}'$ is a mixed type expression.

5.2.5 Unfolding

A very important transformation of expressions describing list [or mixed type] functions is unfolding. This consists in isolating the scalar expression which represents the first element of the list computed by the list function, by transformations of the expression of the function. In other terms, the purpose of unfolding a list expression $\mathcal{E}$ is to transform it into $E \overset{\cdot}{\smile} \mathcal{E}'$, where E is a scalar expression representing the first element of the list.

The transformations use certain straightforward unfolding rules (also presented in [JS05]), as well as the functional definitions of the functions which occur in the expressions.

Unfolding provides a systematic method for the detection of online transitive functions: if we manage to transform the expression of $F[x \overset{\cdot}{\smile} X]$ into the expression $E[x] \overset{\cdot}{\smile} \mathcal{E}[x, X]$, then the function is online and $E[x]$ is its scalar projection. Moreover, the function is transitive if and only if $F[X] = \mathcal{E}[x, X]$.

We will show hereafter how this transformation can be implemented as a set of equational rewrite rules.

The unfolding function $\mathcal{U}$ is implemented in the following way:

- Scalar expressions, as well as already unfolded ones remain unchanged:

$$\mathcal{U}[E] = E, \tag{5.10}$$

$$\mathcal{U}[E \overset{\cdot}{\smile} \mathcal{E}] = E \overset{\cdot}{\smile} \mathcal{E}. \tag{5.11}$$

- A list variable or the tail of it is further decomposed into head and tail:

$$\mathcal{U}[X] = H_0[X] \overset{\cdot}{\smile} T_1[X]. \tag{5.12}$$

$$\mathcal{U}[T_i[X]] = H_i[X] \overset{\cdot}{\smile} T_{i+1}[X]. \tag{5.13}$$

- If $\vec{g}$ is a list function, then one first unfolds the [list] argument, and then applies the recursive definition of $\vec{g}$:

$$\mathcal{U}[\vec{g}[\mathcal{E}]] = \mathcal{U}[\vec{g}[\mathcal{U}[\mathcal{E}]]] \ , \tag{5.14}$$

$$\mathcal{U}[\vec{g}[E \overset{\cdot}{\smile} \mathcal{E}]] = E' \overset{\cdot}{\smile} \mathcal{E}' \ , \tag{5.15}$$

  where $E, \mathcal{E}, E'$ and $\mathcal{E}'$ are the corresponding instances of the expressions occurring in the definition of $\vec{g}$.

- Similar rules apply to mixed scalar–list functions:
  If G is a mixed type function having property (5.4), then

$$\mathcal{U}[G[E, \mathcal{E}]] \ = \ U[G[E, U[\mathcal{E}]]]$$

$$\mathcal{U}[G[E, E' \overset{\cdot}{\smile} \mathcal{E}]] \ = \ E'' \overset{\cdot}{\smile} \mathcal{E}'$$

5.3 Systolic Processors

We use here the term "systolic processor" for designating just a fixed size processor (or hardware unit), with or without internal memory just in order to emphasise the fact that such units may be used as individual processors in a systolic array.

In this section we study the behaviour of several types of such processors, namely we investigate the properties of the functions which are realised by them.

5.3.1 Systolic Processor without Internal State

A systolic processor without internal state is the simplest building block of a systolic array. It is also called *combinatorial cell* in the literature.

The processor depicted in Figure 5.2 receives as input the list $X = \langle x_0, x_1, \ldots \rangle$ and computes the output list $Y = \langle y_0, y_1, \ldots \rangle$. The transition function, that is the computation performed by the processor at each time step is f, such that $y = f[x]$.

If the processor computes the function $Y = F[X]$, then the following equation characterises its functioning:

$$F[x \overset{\cdot}{\smile} X] = f[x] \overset{\cdot}{\smile} F[X] \tag{5.16}$$

From the discussion in Sect. 5.2.2, it follows that the class of functions which can be realised by a systolic stateless processor is exactly the class of online transitive functions.

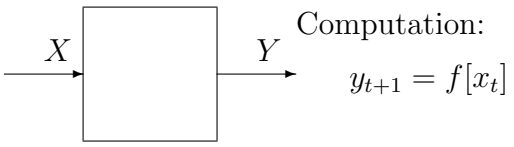


Figure 5.2: Systolic processor without internal state

Note: the scalar argument x of f stands for the value(s) of all input channels of the PE, that is it also can be a multiple of simple scalar types. If we denote by $\vec{f}$ the *transitive extension* of f, then $\vec{f}$ is the list function constructed by f if we pipeline X through the PE: $F = \vec{f}$.

Hereafter we specify the design problem for such a simple PE.

Problem:
The design problem consists in finding f, when F is given.

Method:
Unfold F and if the result is an equation of the form (5.16), then f is found by projection.

Example 5.1. - *Polynomial addition*
Let $A = \langle a_0, a_1, a_2, \ldots \rangle$ and $B = \langle b_0, b_1, b_2, \ldots \rangle$ be the list of the coefficients of the univariate polynomials

$\overline{A} = a_0 + a_1 x + a_2 x^2 + \ldots + a_{n-1} x^{n-1}$, respectively

$\overline{B} = b_0 + b_1 x + b_2 x^2 + \ldots + b_{m-1} x^{m-1}$,

such that $a_i = \$ = 0, \forall i \geq n$ and $b_j = \$ = 0, \forall j \geq m$.

The function F to be computed is $F[A, B] = A + B$.

We unfold F:

$$F[A, B] = (a_0 \overset{\cdot}{\smile} A_1) + (b_0 \overset{\cdot}{\smile} B_1) = (a_0 \overset{\cdot\cdot}{+} b_0) \overset{\cdot}{\smile} F[A_1, B_1] \quad (5.17)$$

Equation (5.17) is of the form of equation (5.16), thus we conclude that the transition function should be $f[a, b] = a \overset{\cdot\cdot}{+} b$.

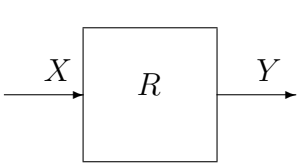
5.3.2 Systolic Processor with Internal State

The architecture of a systolic processor with internal state, depicted in Figure 5.3 differs from that of Fig. 5.2 only in having one more additional element: the internal state register r. The list of values of the internal state register is denoted by R.

The processor will not only output a new y element of the result at each time step, but will also update the value of its internal state register r depending on the input, respectively the value of the internal state register r at the previous time step.

The output list Y computed by the processor is characterised by the following equation:

$$F[r, x \overset{\cdot}{\smile} X] = f_y[r, x] \overset{\cdot}{\smile} F[f_r[r, x], X] \ . \quad (5.18)$$



Computations:

$$\begin{aligned} y_{t+1} &= f_y[r_t, x_t] \\ r_{t+1} &= f_r[r_t, x_t] \end{aligned}$$

Figure 5.3: Systolic processor with internal state register

The processor updates its internal state (consisting of the output y and the internal state register r) at each time step according to the given computations, also shown on Figure 5.3.

We denote by $G[R, X]$ the function computed by the array, that includes all the values of the internal state:

$$G[r \overset{\cdot}{\smile} R, x \overset{\cdot}{\smile} X] = \langle f_y[r, x], f_r[r, x] \rangle \overset{\cdot}{\smile} G[R, X] \ . \tag{5.19}$$

One notes that G is the transitive extension of $g = \langle f_y[r, x], f_r[r, x] \rangle$ (in other terms $G = \vec{g}$), and g is the transition function of the PE. The f_y component of the transition function calculates the values of the output channels, respectively the f_r component updates the value(s) of the internal state register(s).

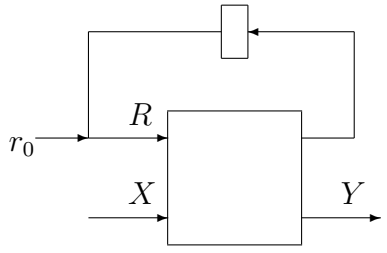
Figure 5.4 presents an alternative view of the systolic processor with internal state register. From this point of view such a processor is similar to a processor without internal state register with input $\langle R, X \rangle$. The only difference is that in case of R, only the first element r_0 is given from outside, the rest of the elements of R are computed step by step, according to the rule:

$$r_{t+1} = f_r[r_t, x_t]$$

Equation (5.19) is of the form (5.16). Thus we conclude that equation (5.16) characterises the transition function of both PE-types (with and without internal state register), however in case of the transition function of a PE with internal state register the following form is more concise:

$$G'[r, x \overset{\cdot}{\smile} X] = g[r, x] \overset{\cdot}{\smile} G'[f_r[r, x], X] \ . \tag{5.20}$$

Problem:
The design problem consists in finding $F[r, X]$, r_0, f_y and f_r, when $F'[X]$ is given, such that (5.18) holds and $F'[X] = F[r_0, X]$.



Computations:

$$\begin{aligned} r_0 &- \text{given}, \\ r_{t+1} &= f_r[r_t, x_t], \forall t \geq 0 \\ y_{t+1} &= f_y[r_t, x_t] \end{aligned}$$

Figure 5.4: An alternative view of the systolic processor with internal state register

Method:
We unfold $F'[X]$ at first: $F'[x \dot{\smile} X] = f'[x] \dot{\smile} \mathcal{F}'[x, F'[X]]$. From $\mathcal{F}'$ we "guess" F and r_0, then we unfold F.

Example 5.2. - *Integer addition*
Let $A = \langle a_0, a_1, a_2 \ldots \rangle$ and $B = \langle b_0, b_1, b_2 \ldots \rangle$ be the list representation of two arbitrary large integers. A represents $a_0 + a_1\beta + a_2\beta^2 + \ldots$, where $\beta > 1$ is the radix for integer representation. The function F', that computes the sum of the two integers is given:

$F'[A, B] = A + B$.

By unfolding F' we get:

$$\begin{aligned} F'[A,B] &= A + B = (a_0 \dot{\smile} A_1) + (b_0 \dot{\smile} B_1) = \\ &= (a_0 \dddot{+} b_0)_{mod\beta} \dot{\smile} \underbrace{(\lfloor \frac{a_0 \ddot{+} b_0}{\beta} \rfloor \dot{+} A_1 + B_1)}_{F} \end{aligned}$$

From the result of unfolding, that is in our case $\lfloor \frac{a_0 \ddot{+} b_0}{\beta} \rfloor \dot{+} A_1 + B_1$, we choose for the function F the form $F[c, A, B] = c \dot{+} (A + B)$.

By unfolding F we get:

$$\begin{aligned} F[c,A,B] &= c \dot{+} (a_0 \dot{\smile} A_1) + (b_0 \dot{\smile} B_1) = \\ &= (a_0 \ddot{+} b_0 \ddot{+} c)_{mod\beta} \dot{\smile} (\lfloor \tfrac{a_0 \ddot{+} b_0 \ddot{+} c}{\beta} \rfloor \dot{+} A_1 + B_1) \end{aligned}$$

From here we have

$$\begin{cases} f_y[c,a,b] &= (a \dot{+} b \dot{+} c)_{mod\beta} \\ f_c[c,a,b] &= \lfloor \frac{a \dot{+} b \dot{+} c}{\beta} \rfloor \end{cases}$$

The last step is to find out the initialization value for c_0, such that the equality $F[c_0, A, B] = F'[A, B]$ holds:

$$F[c,A,B] = c \dot{+} (A+B) \Rightarrow F'[A,B] = A + B = F[0,A,B] \Rightarrow c_0 = 0.$$

Note that the difference of two integers can be computed in a similar way.

5.3.3 Systolic Processor with Delay

In Sect. 5.3.1 we have seen that a function $F[X]$, having property (5.16) can be realised by a single processor.

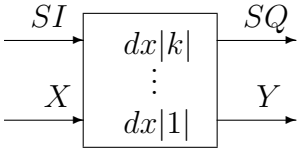
For a positive constant k, let us consider functions $F[X]$ having the property:

$$F[X] \;=\; G[X, T[X], T[T[X]], \ldots, T_k[X]] \tag{5.21}$$

for some online transitive function G. We show now that such a function can be computed by a so called systolic processor with k-delay.

Since G can be computed by a single processor, we only need to use k input channels for the shifted elements of the same input X.

One can in fact avoid the multiplication of the input by introducing k transition registers, denoted by $dx|1|, dx|2|, \ldots, dx|k|$, which perform the computation shown on Figure 5.5. This solution will result in a delay of the result with k steps.



Computations:

$$\begin{aligned} dx|1|_{t+1} &= x_t \\ dx|j|_{t+1} &= dx|j-1|_t, \forall 2 \le j \le k \\ y_{t+1} &= f[x_t, dx|1|_t, \ldots, dx|k|_t] \\ sq_{t+1} &= si_t \end{aligned}$$

Figure 5.5: Systolic processor with k-delay

We denote the initial values of the transition registers $dx|1|$, $dx|2|$, ..., $dx|k|$ with $x_{-1} = T_{-1}[X]$, $x_{-2} = T_{-2}[X]$, ..., $x_{-k} = T_{-k}[X]$ respectively.

These are blank values, which do not contribute to the computation of the result.

Register $dx|k|$ realises the list X_{-k}, $dx|k-1|$ the list $X_{-(k-1)}$ and so on..., finally $dx|1|$ realises the list X_{-1}, while the input x generates the list X. However such a processor can output the first result only beginning with the $k+1^{th}$ time step. If we consider the generated list expressions after k steps, then register $dx|k|$ can be associated with the list $T_k[X_{-k}] = X$, register $dx|k-1|$ with the list $T_k[X_{-(k-1)}] = T[X]$ ..., register $dx|1|$ with the list $T_k[X_{-1}] = T_{k-1}[X]$ and the input x with the list $T_k[X]$.

The processor has an output y at each time step, but as already explained, in case of this processor-type the first "interesting" output appears only at the $k+1^{th}$ time step. If the constant k is known, one can obtain the result by simply dropping the first k elements of Y (that is one considers only $T_k[Y]$), but a more general solution is to introduce a control signal S that indicates the appearance of the real results.

The elements of the control signal SI are output unchanged at the next time step (SQ) as shown on Figure 5.5. The the first $k-1$ elements of the SI input representing the control signal are 0, while the other elements starting with the k^{th} one are 1, thus an output y at time step t is considered to be a valid output if the corresponding sq_t is 1.

Problem:
The design problem consists in finding the scalar projection of G when $F[X] = G[X, T[X], \ldots T_k[X]]$ is given.

Method:
Unfold F and verify that the resulting equation has the property (5.21).

Introduce the transition registers $dx|1|, \ldots, dx|k|$ and add the control signal SI, then project the occurrence of $T_k[X]$ into register x, and all occurrences of $T_i[X]$ $(0 \leq i \leq k-1)$ into the corresponding transition register $dx|k-i|$ to find f.

Note that some of the arguments $X, T[X], \ldots, T_{k-1}[X]$ could also be missing. In this case we only need the transition registers from the one corresponding to $T_{k-1}[X]$ to the register associated with $T_{min}[X]$.

In the particular case when $F[X] = G[T_k[X]]$ no transition register is needed, but the control signal still can be used to indicate the appearance of the first result.

Example 5.3. - *The shift operation*
Let $A = \langle a_0, a_1, \ldots \rangle$ and $B = \langle b_0, b_1, \ldots \rangle$ be two lists of any scalar type and $X = \langle A, B \rangle$ is considered to be the input to the problem. We want to compute $Y = \langle A, T_2[B] \rangle$, that is the component A of the input X will be associated with the component B shifted by two positions.

We define the function F_A and F_B that select the list A respectively B from the input $X = \langle A, B \rangle$:

$$\begin{aligned} F_A[X] &= F_A[\langle A, B \rangle] = A \\ F_B[X] &= F_B[\langle A, B \rangle] = B \end{aligned}$$

Then the list of the results is $Y = \langle F_A[X], F_B[T_2[X]] \rangle$. Also note that both F_A and F_B have the property (5.16), from here results that $F[X, T_2[X]] = \langle F_A[X], F_B[T_2[X]] \rangle$ has property (5.21).

Because in this example $k = 2$, we introduce two transition registers $dx|1|$ and $dx|2|$ (each of them has a component for A and one for B, thus we also can talk about four registers denoted by $da|1|$, $db|1|$, $da|2|$ and $db|2|$. In the same way we can talk about the input channels a and b, being the two components of the input x).

By projecting X into $dx|2|$ and $T_2[X]$ into the input channel x we get the computations performed by the processor:

$$y = \langle F_A[dx|2|], F_B[x] \rangle$$

In other terms:

$$y = \langle da|2|, b \rangle$$

The list corresponding to the control signal is $SI = \langle 0, 0, 1, 1, 1, \ldots \rangle$.

In this particular case a further optimization is possible: $db|1|$ and $db|2|$ is not used, thus they can be eliminated.

5.3.4 Auto-configurable Systolic Processor (with Delay)

Now let us suppose that $F[X]$ depends on a fixed length (say k) prefix of X. More exactly, $F[X] = G[\langle H[X], H_1[X], \ldots, H_{k-1}[X] \rangle, T_k[X]]$, where G has the property

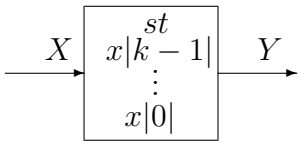
$$G[a, x \overset{\cdot}{\smile} X] = g[a, x] \overset{\cdot}{\smile} G[a, X]. \tag{5.22}$$

Note that in the case of a mixed type function $G[a, X]$, this property can be rewritten (by considering the scalar a as a parameter) as:

$$G_a[x \overset{\cdot}{\smile} X] = g_a[x] \overset{\cdot}{\smile} G_a[X],$$

which is of the form of (5.16) – thus G_a is the transitive extension of g_a.

We used the notation $a = \langle H[X], H_1[X], \ldots, H_{k-1}[X] \rangle$, in order to write the property (5.22) in the form of (5.16), however this means that the processor "knows" the $H[X], H_1[X], \ldots H_{k-1}[X]$ scalar values, or at least these values are preloaded in some local registers of the processor.



Computations:

$$\begin{aligned}
x|j|_{t+1} &= x|j|_t, \text{ if } (st_t > j) \vee (st_t < j) \\
&= x_t, \text{ if } st_t = j, \forall 0 \leq j \leq k-1 \\
st_{init} &= 0 \\
st_{t+1} &= st_t + 1, \text{ if } st_t < k \\
&= st_t, \text{ otherwise} \\
y_{t+1} &= f[x_t, x|0|_t, \ldots, x|k-1|_t]
\end{aligned}$$

Figure 5.6: Systolic processor that computes $F[x_0, x_1, \ldots, x_{k-1}, T_k[X]]$

Alternatively, we can use k static registers (denoted by $x|0|$, $x|1|$, $\ldots x|k-1|$, as shown on Fig. 5.6) that will store the first k values of the input, then beginning with the $k+1^{th}$ time step the "interesting" results start to appear on the output.

A static register $x|j|$ stores the input x at time step $t = j$, then it keeps its value unchanged. Because the processor is not aware of the time, a state register st is introduced, that is initialised with 0, then incremented at each time step up to the value of k.

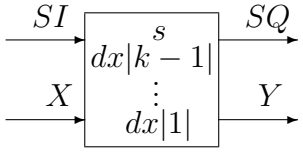
An *auto-configurable processor* constitutes a particular case of such processors:
If in the expression of the function $G[\langle x_0, x_1, \ldots, x_{k-1} \rangle, T_k[X]]$ the scalars x_0, x_1, ..., x_{k-1} do not contribute directly to the computation of the result, they are only used to verify whether a condition holds or not, then the functioning of the processor can be optimised by introducing a state register s. Rather then verifying the condition at each time step, the state register s will be set in the k^{th} time step according to the condition, that depends on $x_0, x_1, \ldots, x_{k-1}$. Afterwards, beginning with time step $k+1$, when the first interesting result should be computed, the computation will be performed in the function of the value of the state register s which is already set in the proper way.

Note that in this case we actually do not need the values $x_0, x_1, \ldots, x_{k-1}$ after the k^{th} time step, so it is useless to store them. That is why in this case we can use transition registers instead of static registers, and the state register st is not needed. $k-1$ transition registers are sufficient, then in the k^{th} time step $H_{k-1}[X] = x_{k-1}$ is input to the processor, while the elements x_0, x_1, ..., x_{k-2} are stored in the transition registers $dx|k-2|$, $dx|k-3|$, ..., $dx|1|$, respectively. At this time step the state register s can be set according to the first k values of X, then beginning with the $k+1^{th}$ step the results can be output in function of the s state register's value.

Except for the state register, the processor is functioning just like a processor with $k-1$-delay, which means that it also can compute a function of the form $F[x_0, \ldots, x_{k-1}, X, T[X], \ldots, T_{k-1}[X]]$ (with the restriction that in the case of the first $k-1$ elements of the result – that is $H_j[F], 0 \leq\ j\ \leq k-1$ – the j^{th} result depends only on the first j elements of the input, that is on $x_0, \ldots x_j$, the state register is not involved). Thus we can call it auto-configurable systolic processor with $k-1$ delay. Moreover an auto-configurable processor, that sets its state register according to the first k values of the input can be combined wit a processor with m-delay. The number of transition registers used should be $max(k-1, m)$.

The only question is how does the processor know when should the state register s be set. This problem can be solved using the same control signal SI as in the case of a processor with delay. The list of input signals SI will have $k-2$ leading 0 elements, then starting with position $k-1$ the value of the elements will be 1 and the first 1 value will indicate the moment when s has to be set. The state register s is initialised with a blank value, we denote it by \$.

The computations performed by such a processor are shown on Fig. 5.7.



Computations:

$$
\begin{array}{lcl}
dx|1|_{t+1} & = & x_t \\
dx|j|_{t+1} & = & dx|j-1|_t, \forall 2 \leq j \leq k-1 \\
y_{t+1} & = & f_y[s_t, x_t(, dx|1|_t, \ldots, dx|k-1|_t)] \\
sq_{t+1} & = & si_t \\
s_{init} & = & \$ \text{ (blank value)} \\
s_{t+1} & = & f_{cond}[x, dx|1|_t, \ldots, dx|k-1|_t], \\
 & & \quad \text{if } (s_t = \$) \wedge (si_t = 1) \\
 & = & s_t, \text{ otherwise}
\end{array}
$$

Figure 5.7: Auto-configurable systolic processor (with $k-1$-delay)

Hereinafter we specify the design problem for such a processor.

Problem:
The design problem consists in finding $f = \langle g_y, g_{cond} \rangle$ when
$F[X] = G[x_0, \ldots, x_{k-1}, (X, T[X], \ldots,)T_k[X]]$
is given, such that for G the property (5.16) (respectively property (5.21)) holds.

Method:
Unfold F and verify that the appropriate property holds.

Introduce the $k-1$ transition registers.

g_{cond} is a function that associates to "if" statements integer values from $\{1, 2, \ldots m\}$, where m is the number of different cases that depend on $x_0, x_1, \ldots, x_{k-1}$ in the definition of F.

Obtain g_y by projection. Each "if $cond_i(x_0, x_1, \ldots, x_{k-1})$" statement is projected to a corresponding "if $s = i$" statement. (The other projection rules are similar to the rules for processors with delay.)

Neither of the following two examples is a typical one, but we will use them in building up the systolic array for GCD computation. A more complex example for such a processor is presented in Sect. 5.4.5.

Example 5.4. - *conditional exchange*
Let $A = \langle a_0, a_1, \ldots \rangle$ and $B = \langle b_0, b_1, \ldots \rangle$ be two lists of any scalar type and $X = \langle A, B \rangle$ is considered to be the input to the problem. If a_0 is equal to a given constant (we denote it by $exch$) of the same scalar type as that of the list A, the output should be $\langle B, A \rangle$ otherwise it will remain $\langle A, B \rangle$. This means that the two components of the input are exchanged depending on the first element of A:

$$\begin{aligned} F[a_0, \langle A, B \rangle] &= \langle B, A \rangle, \quad \text{if } a_0 = exch \\ &= \langle A, B \rangle, \quad \text{otherwise} \end{aligned}$$

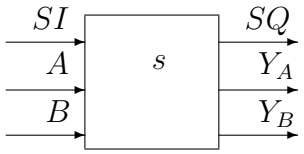
Using notation $F_A[X] = A$ and $F_B[X] = B$ we can write:

$$\begin{aligned} F[H[X], X] &= \langle F_B[X], F_A[X] \rangle, \quad \text{if } F_A[H[X]] = exch \\ &= X, \qquad\qquad\qquad\quad \text{if } F_A[H[X]] \neq exch \end{aligned}$$

Both, F_A and F_B have property (5.16), so one can easily verify that $F_{H[X]}$ also satisfy property (5.16). Because $k = 1$ no transition register

is needed, but because F depends on X, the first output will already appear at the first time step, at the same moment when the state register s is set.

The function F involves two "if"-statements. The first one will be projected to $s = 1$ and the second one to $s = 2$. The computations performed by the processor are described on Figure 5.8. For the list of results we use the notation $Y = \langle Y_A, Y_B \rangle$. The list corresponding to the control signal is $SI = \langle 1, 1, \ldots \rangle$



Computations:

$$
\begin{array}{lcll}
y_{t+1} & = & \langle b_t, a_t \rangle, & \text{if } (s_t = 1) \vee \\
 & & & ((s_t = \$) \wedge (a_t = exch)) \\
 & = & \langle a_t, b_t \rangle, & \text{if } (s_t = 2) \vee \\
 & & & ((s_t = \$) \wedge (a_t \neq exch)) \\
sq_{t+1} & = & si_t & \\
s_{init} & = & \$ & \\
s_{t+1} & = & 1 & \text{if } (s_t = \$) \wedge (si_t = 1) \wedge \\
 & & & (a_t = exch) \\
 & = & 2 & \text{if } (s_t = \$) \wedge (si_t = 1) \wedge \\
 & & & (a_t \neq exch) \\
 & = & s_t, & \text{otherwise}
\end{array}
$$

Figure 5.8: Auto-configurable systolic PE for input-exchange

Example 5.5. - *Cutting off the least significant zeroes of a binary integer*
Let $A = \langle a_0, a_1, \ldots \rangle$ be the list representation of a binary integer. We want to cut off the least significant zeroes of A, which means we want to compute $Y = F[A] = T_k[A]$ such that $H_k[A] = 1$ and $H_j[A] = 0, \forall 0 \leq j \leq k-1$.

The first remark is that here only the k^{th} tail of the input appears in the computation of the result, which means that we do not need any transition register.

The second remark is that here k is not known in advance, but it can be computed in function of the input, which induces a slight modification in the computation of the control signal, that indicates the appearance of the first result, respectively in the computation of the state register

s: both of them will be computed in function of the input A rather then using the input SI, which is unimportant. The output SQ is computed in the following way:

$$\begin{aligned} sq_{t+1} &= 0 \quad \text{, if} \quad (s_t = \$) \wedge (x_t = 0) \\ &= 1 \quad \text{, if} \quad (s_t \neq \$) \vee (x_t = 1) \end{aligned}$$

The computations for s are:

$$\begin{aligned} s_{t+1} &= 1, \quad \text{if } (s_t = \$) \wedge (x_t = 1) \\ &= s_t, \quad \text{if } (s_t \neq \$) \vee (x_t = 0) \end{aligned}$$

The first 1 value of SQ indicates the beginning of the output.

$T_k[A]$ is projected to the input register a, thus the computation of the result (we denote the output register associated to it with y) is very simple:

$$y_{t+1} = a_t$$

5.4 Unidirectional Arrays

Figure 5.9. depicts the architecture of a unidirectional systolic array with n processing elements. The initialization scheme is also given. The term "unidirectional" refers to the data-flow that moves in only one direction. The input data (denoted by X) as well as the (partial) results advance (through the channels denoted by y) in the same direction. The list $X = \langle x_0, x_1, x_2, \ldots \rangle$ represents the input to the array, introduced element by element at each time step, respectively Y^0 denotes the list of initial values, which contribute to the computation of the results (usually the same y^0 value is introduced repeatedly, then $Y^0 = (y^0)^\infty$, or simply a blank value, that we denote by \$, then $Y^0 = \$^\infty$).

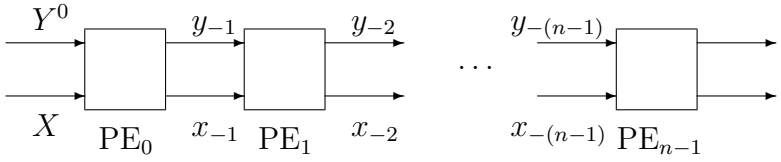


Figure 5.9: Unidirectional array

The x_i, respectively y_i values, where $i < 0$ represent the initialization values of the array. At each time step t a new result leaves the array at PE_{n-1}.

We use the following notation: whenever a variable w_t^i is mentioned, the upper index i denotes the number of PE, the variable belongs to (that is the *space index*), while the lower index is the *time index*. Thus w_t^i is the value of variable w at PE_i at the time instant t. w_t (variable with only the lower index) is either an input value or an initialization value, while w^i (variable with only the upper index) represents either an input channel or an internal state register that belongs to PE_i.

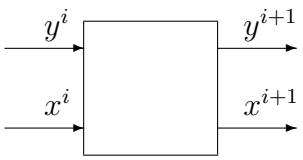
The transition function (that indicates the computations performed by a PE) depends on the input to the PE, in this case x and y, respectively on the values of the internal state registers in the previous time step.

A PE may or may not have internal state registers and the data can move with the same or with different speed. In the sequel we study in detail these cases.

5.4.1 Unidirectional Array with Input "Pass-Through"

We consider here that the x inputs are just "passing through" the array from PE_i to PE_{i+1}, that is $x_{t+1}^{i+1} = x_t^i$, while the outputs move with the same speed. We will formalise at first the computation performed by such a simple array that does not have any other registers.

In Fig. 5.10. the computations of a PE are given for an array of this type.



Computations:

$$\begin{aligned} y_{t+1}^{i+1} &= f[x_t^i, y_t^i] \\ x_{t+1}^{i+1} &= x_t^i \end{aligned}$$

Figure 5.10: Unidirectional array with input "pass-through". Computations of a PE

The transition function is composed of two functions, one for the computation of the partial results (y), and another one for the computation of the x component, which is the identity function ($f_x[x] = x$ is considered to be known). We also call the latter one the *trivial part* of the transition function.

The first $n-1$ elements of the result that leave the array at PE_{n-1} depend only on the internal initialization values:

$$\begin{aligned} y_1^n &= f[x_0^{n-1}, y_0^{n-1}] = f[x_{-(n-1)}, y_{-(n-1)}] \\ y_2^n &= f[x_1^{n-1}, y_1^{n-1}] = f[x_0^{n-2}, f[x_0^{n-2}, y_0^{n-2}]] = \\ &= f[x_{-(n-2)}, f[x_{-(n-2)}, y_{-(n-2)}]] \\ &\dots \\ y_{n-1}^n &= \dots = \underbrace{f[x_{-1}, f[x_{-1}, \dots f}_{n-1 \text{ times}}[x_{-1}, y_{-1}] \dots] \end{aligned}$$

The first element, that also involves the input X is y_n^n:

$$\begin{array}{lcllcll} y_n^n & = & f[x_{n-1}^{n-1}, y_{n-1}^{n-1}] & = & f[x_{n-2}^{n-2}, f[x_{n-2}^{n-2}, y_{n-2}^{n-2}]] & = & \ldots \\ & = & \underbrace{f[x_0^0, f[x_0^0, \ldots f}_{n \text{ times}}[x_0^0, y_0^0] \ldots] & = & \underbrace{f[x_0^0, f[x_0^0, \ldots f}_{n \text{ times}}[x_0^0, y^0] \ldots] & & \\ \ldots & & & & & & \\ y_{n+i}^n & = & f[x_i^0, f[x_i^0, \ldots f[x_i^0, y_i^0] \ldots] & = & f[x_i^0, f[x_i^0, \ldots f[x_i^0, y^0] \ldots] & & \end{array}$$

If Y is the list of the outputs, then we denote by $Y_n = T_n[Y]$ the n^{th} tail of the list of results, that contains the elements depending on the input X. We consider the list extension $\vec{f}$ of f, then we can write the equation that characterises this type of array in a more concise form:

$$Y_n = \underbrace{\vec{f}[X, \vec{f}[X, \ldots \vec{f}}_{n \text{ times}}[X, Y^0] \ldots] \ . \tag{5.23}$$

Note that $\vec{f}$ satisfies property (5.16). Equation (5.23) can be expressed recursively. Let $F[n, X]$ be the function that computes Y_n (n is the size parameter of the computation), then (5.23) can be expressed by the following recursive description:

$$\begin{cases} F[n, X] & = & \vec{f}[X, F[n-1, X]] \\ F[1, X] & = & \vec{f}[X, Y^0] \ . \end{cases}$$

The previous considerations allow us to formulate the design problem for such an array.

Design Problem 5.1. (Unidirectional array with input "pass-through")

Given a functional program describing $F[n, X]$

find y^0 (where $Y^0 = (y^0)^\infty$) and the nontrivial part of the transition function f such that

$$F[n, X] = \vec{f}[X, F[n-1, X]] \tag{5.24}$$

$$F[1, X] = \vec{f}[X, Y^0] \ , \tag{5.25}$$

where $\vec{f}$ satisfies condition (5.16).

Note: One can observe that y_{n+i}^n only depends on x_i (and y^0). It follows that for an input of finite length $X_{0,k} = \langle x_0, \ldots, x_k \rangle$, a function $F[n, X_{0,k}]$ that satisfies the conditions (5.24)-(5.25) can be computed by a SIMD vector

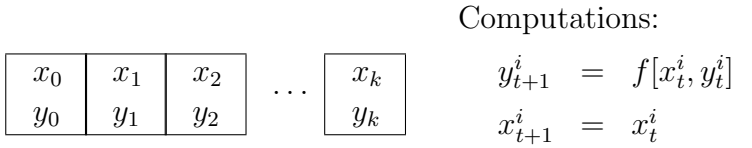


Figure 5.11: An alternative for the computation performed by an array characterised by (5.23)

processor in n time steps, as shown on Fig. 5.11 (no communication is needed between the neighbouring processing elements).

Example 5.6. - *Computing the n^{th} power of the elements of a list*

Given a list X, we want to compute $F[n, X] = X^n = \langle x_0^n, x_1^n, x_2^n \ldots \rangle$.
We can define $F[n, X]$ recursively, in the following way:

$$F[n, X] = X * F[n-1, X] \ , \tag{5.26}$$

where the operation $*$ is the scalar product of two lists. This means

$$(a \dot{\smile} A) * (b \dot{\smile} B) \stackrel{def}{=} \langle a \ddot{*} b \rangle \dot{\smile} A * B$$

Equation (5.26) is of the form (5.24). Indeed, in equation (5.26) we have

$$X * F[n-1, X] = x_0 * H[F[n-1, X] \dot{\smile} X_1 * T[F[n-1, X]]] =$$
$$= x_0 * x_0^{n-1} \dot{\smile} X_1 * X_1^{n-1} \Rightarrow$$

$\Rightarrow$ the function from the *rhs.* of (5.26) and the corresponding scalar function $f[x, y] = x \ddot{*} y$ satisfies condition (5.16).

We obtained the transition function f from (5.26) by projecting the input list X into the input register x and the function F that computes the result into the output register y.

We now verify condition (5.25).
The following equality should hold: $F[1, X] = \vec{f}[X, Y^0]$.
That is $X = X * Y^0$. From $x = x \ddot{*} y^0 \Rightarrow y^0 = 1$. Thus $Y^0 = 1^\infty$.

Internal state registers having constant values

The most simple internal state register, a PE could have, is that one having a constant value. We analyse this particular case separately. In this case we have in addition to the PE presented in Fig. 5.10. an internal state register q^i, that does not change its value ($q^i_{t+1} = q^i_t = q_i$).
Let $Q_0, n-1 = \langle q_0, q_1, \ldots, q_{n-1} \rangle$ be the finite list of the internal state register values corresponding to $\mathrm{PE}_0, PE_1 \ldots PE_{n-1}$ respectively.
Because $q_0, q_1, \ldots, q_{n-1}$ are constant values, we can treat them as "parameters" of the problem. Now the computation for y^{i+1}_t also involve q_i. The computations for a PE are:

$$\begin{cases} y^{i+1}_{t+1} &= f[q^i_t, x^i_t, y^i_t] &= f[q_i, x^i_t, y^i_t] \\ x^{i+1}_{t+1} &= x^i_t & \\ q^i_{t+1} &= q^i_t &= q_i \end{cases}$$

Now the computation of the register q also belongs to the trivial part of the transition function. The equation that characterises the array is of the following form:

$$Y_n = \underbrace{\vec{f}[q_{n-1}, X, \vec{f}[q_{n-2}, X, \ldots \vec{f}}_{n \text{ times}}[q_0, X, Y^0] \ldots] \ . \tag{5.27}$$

The design problem can be now formulated in the following way:

Design Problem 5.2. (Unidirectional arrays with input "pass-through" and constant internal state registers)

Given a functional program describing $F_{w_0,w1,\ldots,w_{n-1}}[n, X]$,
(where $W_{0,n-1} = \langle w_0, w1, \ldots, w_{n-1} \rangle$ is the list of parameters)

find $Q_{0,n-1}$ which is a permutation of $W_{0,n-1}$, y^0 (where $Y^0 = (y^0)^\infty$) and the nontrivial part f of the transition function such that

$$F_{Q_{0,n-1}}[n, X] = \vec{f}[q_{n-1}, X, F_{Q_{0,n-2}}[n-1, X]] \tag{5.28}$$

$$F_{q_0}[1, X] = \vec{f}[q_0, X, Y^0], \tag{5.29}$$

such that if we consider q a parameter of the function f (for this reason we use the notation f_q), then $\vec{f_q}[\langle X, Y \rangle]$ satisfies property (5.16).

Note: In this case each value of the result also depends on the values $q_0, q_1, \ldots q_{n-1}$, which means that if we would like to perform the computations with a SIMD vector processor (similar to the one shown on Fig. 5.11), then

the values $q_{n-1}, \ldots, q_0$ respectively should be broadcasted at the consecutive time steps to each of the processors.

For that reason the systolic solution to the problem might be more reasonable, because it avoids global broadcasting.

Example 5.7. - *Evaluating a given polynomial for several distinct values*

Let $P_n(x) = a_0 + a_1 * x + a_2 * x^2 + \ldots a_{n-1} * x^{n-1}$ be a univariate polynomial of degree $n-1$.
We want to evaluate the polynomial for some given values. That is, given the input list
$X = \langle x_0, x_1, \ldots \rangle$, and the list of coefficients $A_{0,n-1} = \langle a_0, a_1, \ldots a_{n-1} \rangle$, we want to compute $Y_n = \langle P_n(x_0), P_n(x_1) \ldots \rangle = P_n(X)$.

We have

$$Y_n = a_0 + a_1 * X + \ldots a_{n-1} * X^{n-1} = a_0 + X * (a_1 + a_2 * X + \ldots + a_{n-1} * X^{n-2})$$

From here we conclude that with

$$Q_{0,n-1} = \langle a_{n-1}, a_{n-2}, \ldots, a_1, a_0 \rangle$$

and

$$F_{Q_{0,n-1}}[n, X] = \sum_{j=0}^{n-1} q_{n-1-j} * X^j$$

we get

$$F_{Q_{0,n-1}}[n, X] = q_{n-1} \dot{+} X * F_{Q_{0,n-2}}[n-1, X],$$

that is of the form (5.28). (We use the notation $\dot{+}$ to explicitly indicate that this is an operation between a scalar and a list. Int the same way $\ddot{+}$ denotes the addition between two scalars.) By projecting the list expression into the corresponding scalar expression, we obtain the transition function $f[q, x, y] = q \ddot{+} x \ddot{*} y$ and $Q_{0,n-1} = \langle q_0, \ldots q_{n-1} \rangle$ such that $q_i = a_{n-1-i}, \forall 0 \leq i \leq n-1$.

Verifying condition (5.29) we have $F_{q_0}[1, X] = \vec{f}[q_0, X, Y^0]$. That is $a_{n-1} = a_{n-1} + X * Y^0$. From $a_{n-1} = a_{n-1} \ddot{+} x \ddot{*} y^0$ we get $y^0 = 0$. Thus $Y^0 = 0^\infty$.

Internal state registers having variable values

The next step is to consider also internal state registers with variable value. The case of registers having constant values studied in section 5.4.1 can be also considered a particular case of this more general one.

We denote the internal state register of variable value by r. A PE may have, of course, more internal state registers, however we will talk hereafter -without loss of generality- about only one (this can be of a simple or a multiple scalar type).

The (nontrivial part of the) transition function, denoted by f is now composed of : f_y, which computes the (partial) results and f_r, which updates the value of register r at each time step. More formally:

$$\begin{aligned} f[r_t^i, x_t^i, y_t^i] &= \langle r_{t+1}^i, y_{t+1}^{i+1} \rangle \text{ , such that} \\ r_{t+1}^i &= f_r[r_t^i, x_t^i, y_t^i] \\ y_{t+1}^{i+1} &= f_y[r_t^i, x_t^i, y_t^i] \end{aligned}$$

The function that characterises a unidirectional linear array with input "pass through", having internal state register with variable value is of the following form:

$$Y_n = \underbrace{\vec{f_y}[R_{n-1}^{n-1}, X, \vec{f_y}[R_{n-2}^{n-2}, X, \ldots \vec{f_y}}_{n \text{ times}}[R_0^0, X, Y^0] \ldots] \tag{5.30}$$

$$\begin{aligned} R_i^i &= \langle r_i^i, r_{i+1}^i, \ldots \rangle, \text{ such that} \qquad (5.31) \\ r_{t+1}^i &= f_r[r_t^i, x_t^i, y_t^i], \ t \geq i \\ r_t^i &= r^i, \ t \leq i \end{aligned}$$

We suppose that the initialization of the array is uniform. Let us denote the initialization value for r^i with $r_0, \forall i, 0 \leq i \leq n-1$. Moreover, as already mentioned, the scalar functions produce blank values when applied to blank arguments, thus $f_r[\$, \$, \$] = \$$. Let us assume that $f_r[\$, r_0, \$] = r_0$, which means that the value of r_t^{n-1} will remain r_0 in the fist $n-1$ time steps, until the first "interesting" input appears at PE_{n-1} (see Fig. 5.12).

If the array computes the function $F[n, X]$, then the tail array depicted on Fig. 5.12 computes $F[n-1, X]$.

This kind of view leads to the following recursive description that characterises the functioning of the array:

$$\begin{cases} F[n, X] = G[r_0, X, F[n-1, X]] \\ F[1, X] = G[r_0, X, Y^0] \ , \end{cases}$$

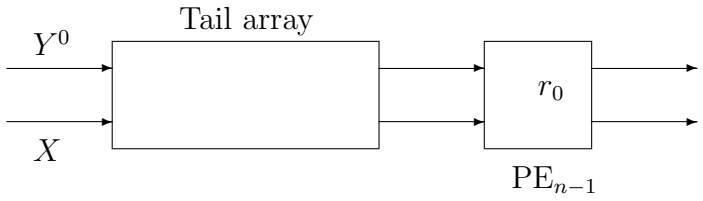


Figure 5.12: Functional view of a unidirectional array with internal state register

where the function G, having one scalar argument r and a composed list argument $\langle X, Y \rangle$ has property (5.20).

Design Problem 5.3. (Unidirectional arrays with input "pass-through" and internal state registers)
Given a functional program describing $F[n, X]$
find

- the initialization value for the internal state registers: r_0
- y^0 (where $Y^0 = (y^0)^\infty$) and the (nontrivial part of the) transition function $f = \langle f_y, f_r \rangle$ such that

$$F[n, X] = \vec{f}[r_0, X, F[n-1, X]] \quad (5.32)$$
$$F[1, X] = \vec{f}[r_0, X, Y^0] \ , \quad (5.33)$$

and $\vec{f}[r, \langle X, Y \rangle]$ satisfies property (5.20)

Example 5.8. - *Alternating two functions that depend on the same size parameter n*

Given the input list $X = \langle x_0, x_1, x_2, \ldots \rangle$ we want to compute the list obtained by alternatively applying on the consecutive elements of X two different functions that depend on the same size parameter n (let these functions be the n^{th} multiple and the n^{th} power of a scalar x). The function to be computed is:

$$F[n, X] = \langle n * x_0, x_1^n, n * x_2, x_3^n, \ldots \rangle$$

The two functions that are alternatively applied on the elements of the input are:

$$\begin{array}{lcl} mult[n,x] & = & n * x \\ power[n,x] & = & x^n \end{array}$$

They both can be expressed recursively:

$$\begin{array}{lcll} mult[n,x] & = & x + (n-1) * x & = & x + mult[n-1,x],\ \forall n, n > 1 \\ mult[1,x] & = & x & & \\ power[n,x] & = & x * x^{n-1} & = & x * power[n-1,x],\ \forall n, n > 1 \\ power[1,x] & = & x & & \end{array}$$

Now we can also write $F[n,X] = \langle mult[n,x_0], power[n,x_1], \ldots \rangle$, but if we want to express the function F in a recursive manner, an auxiliary variable s should be used to indicate which one of the two functions has to be applied.

We define the function $MultPower$ in an inductive way:

$$\begin{array}{l} MultPower[n,s,x \dot{\smile} X] = \\ = \left\{ \begin{array}{ll} mult[n,x] & \text{if } s = 0 \\ power[n,x] & \text{if } s = 1 \end{array} \right. \dot{\smile} MultPower[n, 1-s, X] \end{array}$$

Now $F[n,X] = MultPower[n,0,X]$, where $MultPower$ is a parametrised function, and $MultPower_n[s,X]$ satisfies property (5.20).

By using the recursive description of the scalar functions *mult* and *power* we can define the function F in a recursive way with respect to the size parameter. If

$$PlusMult[s, x \dot{\smile} X, y \dot{\smile} Y] = \left\{ \begin{array}{ll} x + y, & \text{if } s = 0 \\ x * y, & \text{if } s = 1 \end{array} \right. \dot{\smile} PlusMult[1-s, X, Y]$$

then
$F[n,X] = PlusMult[0, X, F[n-1,X]]$ (which is of form (5.32)) and
$F[1,X] = PlusMult[0, X, Y^0]$ (which is of the form of (5.33)).

From here we have the initialization value for s: $s_0 = 0$. And the transition function composed of f_y and f_s is:

$$\left\{ \begin{array}{lcl} f_y[s,x,y] & = & \left\{ \begin{array}{ll} x + y, & \text{if } s = 0 \\ x * y, & \text{if } s = 1 \end{array} \right. \\ f_s[s] & = & 1 - s \end{array} \right.$$

We obtain Y^0 by verifying condition (5.33)):
$F[1,X] = X = PlusMult[0, X, Y^0] = \langle x_0 + y_0^0, x_1 * y_1^0, a_2 + y_2^0, \ldots \rangle \Rightarrow Y^0 = \langle 0, 1, 0, 1, \ldots \rangle$.

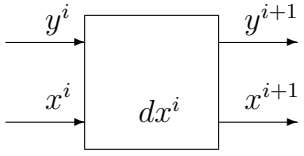
Note that in this case the elements of the result Y are computed by two alternating functions, that is why the initialization value also depends on two functions. Thus $Y^0 = (\langle y_0^0, y_1^0 \rangle)^\infty = (\langle 0, 1 \rangle)^\infty$.

5.4.2 Unidirectional Arrays with Delayed Input

We consider here unidirectional arrays with delayed input. This means that we have one or more special registers in a PE serving as delay elements, which are temporarily storing the input data. We denote a delay element by dx. Generally, the delay of the input with k time steps is realised by k delay elements, denoted by $dx|1|, \ldots, dx|k|$. The following equations describe the computations performed by a PE:

$$\begin{cases} y_{t+1}^{i+1} & = & f[x_t^i, dx|1|_t^i, \ldots, dx|k|_t^i, y_t^i] \\ x_{t+1}^{i+1} & = & dx|k|_t^i \\ dx|j|_{t+1}^i & = & dx|j-1|_t^i, & 2 \leq j \leq k \\ dx|1|_{t+1}^i & = & x_t^i \end{cases}$$

Figure 5.13. presents the computation of a PE, where the input is delayed by one time step. For the sake of simplicity we only consider the equations describing such an array, afterwards the generalization is straightforward.



Computations:

$$\begin{aligned} y_{t+1}^{i+1} &= f[x_t^i, dx_t^i, y_t^i] \\ x_{t+1}^{i+1} &= dx_t^i \\ dx_{t+1}^i &= x_t^i \end{aligned}$$

Figure 5.13: Computation of a PE in an array with delayed input

Figure 5.14. depicts how the array is initialised. The initialization values are the following:

- we denoted by $x_{-1}, x_{-3}, \ldots$ the initial value of the delay elements dx ($dx_0^i = x_{-(2i+1)},\ \forall\ i = 0,\ \ldots, n-1$).
- the value of the input channel to PE_i is x_{-2i}, ($x_0^i = x_{-2i}, \forall i = 1, \ldots, n-1$)
- the values of the output channels are $y_{-1}, y_{-2}, \ldots$ ($y_0^i = y_{-(i+1)}, \forall i = 1, \ldots, n$)

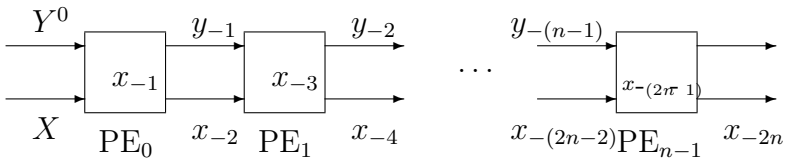


Figure 5.14: Initial state of the array with delayed input

The computation of x and dx belongs to the trivial part of the transition function.

The first $n-1$ elements of the result Y depend only on the initial values of the internal registers:

$$
\begin{aligned}
y_1^n &= f[x_0^{n-1}, dx_0^{n-1}, y_0^{n-1}] = f[x_{-(2n-2)}, x_{-(2n-1)}, y_{-(n-1)}] \\
y_2^n &= f[x_1^{n-1}, dx_1^{n-1}, y_1^{n-1}] = f[dx_0^{n-2}, x_0^{n-1}, f[x_0^{n-2}, dx_0^{n-2}, y_0^{n-2}]] \quad = \\
&= f[x_{-(2n-3)}, x_{-(2n-2)}, f[x_{-(2n-4)}, x_{-(2n-3)}, y_{-(n-2)}]] \\
&\dots \\
y_{n-1}^n &= \dots = \underbrace{f[x_{-n}, x_{-(n+1)}, f[x_{-(n-1)}, x_{-n}, \dots f}_{n-1 \text{ times}}[x_{-2}, x_{-3}, y_{-1}] \dots]
\end{aligned}
$$

The first result that also involves the input X is y_n^n:

$$
\begin{aligned}
y_n^n &= f[x_{n-1}^{n-1}, dx_{n-1}^{n-1}, y_{n-1}^{n-1}] \quad = \\
&= f[dx_{n-2}^{n-2}, x_{n-2}^{n-1}, f[x_{n-2}^{n-2}, dx_{n-2}^{n-2}, y_{n-2}^{n-2}]] \quad = \\
&= \underbrace{f[x_{n-3}^{n-2}, dx_{n-3}^{n-2}, f[x_{n-4}^{n-3}, dx_{n-3}^{n-4}, f[\dots f}_{n \text{ times}}[x_0^0, dx_0^0, y_0^0] \dots] \quad = \\
&= f[x_{-(n-1)}^0, x_{-n}^0, f[x_{-(n-2)}^0, x_{-(n-1)}^0, \dots, f[x_0^0, x_{-1}^0, y_0^0] \dots] \\
&\dots \\
y_{n+i}^n &= f[x_{-(n-1)+i}^0, x_{-n+i}^0, f[x_{-(n-2)+i}^0, x_{-(n-1)+i}^0, \dots, f[x_i^0, x_{i-1}^0, y_i^0] \dots]
\end{aligned}
$$

The equation that characterises the array is of the following form:

$$
Y_n = \underbrace{\vec{f}[X_{-(n-1)}, X_{-n}, \vec{f}[X_{-(n-2)}, X_{-(n-1)}, \dots \vec{f}}_{n \text{ times}}[X, X_{-1}, Y^0] \dots] \qquad (5.34)
$$

Note: The first result that depends only on relevant values of the input (we mean by "relevant values" the elements of X) is

$$y_{2n}^n = f[x_1, x_0, f[x_2, x_1, \ldots f[x_n, x_{n-1}, y_n^0] \ldots]$$

Design Problem 5.4. (Unidirectional arrays with delayed input)
Given a functional program describing $F[n, X]$

find y^0, (where $Y^0 = (y^0)^\infty$) and the (nontrivial part of the) transition function, denoted by f such that

$$F[n, X] = \vec{f}[X_{-(n-1)}, X_{-n}, F[n-1, X]] \quad (5.35)$$
$$F[1, X] = \vec{f}[X, X_{-1}, Y^0] \ , \quad (5.36)$$

where $\vec{f}$ fulfills condition (5.16).

Example 5.9. - *Computing* $\sum_{i=0}^{n-1} W_i$

Given a list $W = \langle w_0, w_1, \ldots \rangle$, we would like to compute

$$\sum_{i=0}^{n-1} W_i = \langle w_0 + w_1 + \ldots + w_{n-1}, w_1 + w_2 + \ldots + w_n, w_2 + w_3 + \ldots + w_{n+1}, \ldots \rangle$$

Extracting the first element of the sum, we can write $\sum_{i=0}^{n-1} W_i = W_0 + \sum_{i=1}^{n-1} W_i$. In order to obtain a recursive description of the form (5.35)-(5.36) we use the following notation:
Let $X = W_{N-1}$ and $F[n, X] = \sum_{i=N-n}^{N-1} X_{i-(N-1)}$. (Note that for $N = n$ the function $F[n, X]$ computes exactly the sum, we are interested in, that is $F[n, X] = \sum_{i=0}^{n-1} W_i$.)
Then $F[n, X] = X_{-(n-1)} + \sum_{i=N-(n-1)}^{N-1} X_{i-(N-1)} = X_{-(n-1)} + F[n-1, X]$, which is of the form of equation (5.35). From here we get by projecting $X_{-(n-1)}$ into x and $F[n-1, X]$ into y, the transition function $f[x, dx, y] = x + y$.

We now verify condition (5.36): $F[1, X] = \vec{f}[X, X_{-1}, Y^0]$, that is $X = X + Y^0$. From here we conclude that $y^0 = 0$. Thus $Y^0 = 0^\infty$.

Note: Because $X = W_{N-1}$ the initialization values $x_{-1}, x_{-2}, \ldots$ are not all blank values, but are equal to the value of the corresponding element of W.

Internal registers

Considering that a PE also has an internal register with constant value, we can reason in the same way as in section 5.4.1. If we introduce a constant

state register $q^i = q_i$ to each PE_i, the equation (5.34), that describes the functioning of the array will change in the following way:

$$Y_n = \underbrace{\vec{f}[q_{n-1}, X_{-(n-1)}, X_{-n}, \vec{f}[q_{n-2}, X_{-(n-2)}, X_{-(n-1)}, \ldots \vec{f}}_{n \text{ times}}[q_0, X, X_{-1}, Y^0] \ldots] \tag{5.37}$$

Design Problem 5.5. (Unidirectional arrays with delayed input and constant internal state registers)
Given $F_{U_{0,n-1}}[n, X]$

find $Q_{0,n-1}$, which is a permutation of $U_{0,n-1}$, y^0 (such that $Y^0 = (y^0)^\infty$) and the (nontrivial part of the) transition function, denoted by f such that

$$F_{Q_{0,n-1}}[n, X] = \vec{f}[q_{n-1}, X_{-(n-1)}, X_{-n}, F_{Q_{0,n-2}}[n-1, X]] \tag{5.38}$$

$$F_{q_0}[1, X] = \vec{f}[q_0, X, X_{-1}, Y^0] \ , \tag{5.39}$$

where the list function $\vec{f}_q[\langle X, X', Y\rangle]$ satisfies condition (5.16). In other terms $\vec{f}[q, \langle X, X', Y\rangle]$ satisfies condition (5.20) with $f = \langle f_y, f_q\rangle$ and $f_q[x] = x$.

Example 5.10. - *Convolution of a finite and an infinite sequence*

Given a finite sequence $A_{0,n-1} = \langle a_0, a_2, \ldots, a_{n-1}\rangle$, and an infinite one, $X_{-(n-1)} = \langle x_{-(n-1)}, x_{-(n-2)}, \ldots, x_0, x_1, \ldots\rangle$, the convolution of $A_{0,n-1}$ and $X_{-(n-1)}$ is
$Y = \langle y_0, y_1, \ldots\rangle$, where

$$y_i = \sum_{j=0}^{n-1} a_j * x_{i-j}, \forall i = 0, 1, 2, \ldots$$

In other terms:

$$Y = \sum_{j=0}^{n-1} a_j * X_{-j} \tag{5.40}$$

(where $a*(x \overset{\cdot}{\smile} X) = \langle a*x\rangle \overset{\cdot}{\smile} (a*X)$)

We can write

$$\begin{aligned} F_{A_{0,n-1}}[n, X] &= \textstyle\sum_{j=0}^{n-1} a_j * X_{-j} = a_{n-1} * X_{-(n-1)} + \sum_{j=0}^{n-2} a_j * X_{-j} = \\ &= a_{n-1} * X_{-(n-1)} + F_{A_{0,n-2}}[n-1, X], \end{aligned}$$

which is of the form (5.38). From here we conclude by projection that the transition function is $f[x, dx, a, y] = a*x + y$.

Verifying condition (5.39) we have: $F_{a_0}[1, X] = \vec{f}[a_0, X, X_{-1}, Y^0]$, that is $a_0 * X = a_0 * X + Y^0 \Rightarrow y^0 = 0$ and $Y^0 = 0^\infty$.

We now consider the more general case, when a PE also has internal register(s) with variable value. This register-type is denoted by r, while q denotes the registers with constant value. As already mentioned, a register with constant value can be seen as a special case of the former one, however we distinguish between the two register-types in the following description, because we consider the case when the constant register values are given as parameter of the problem.

Design Problem 5.6. (Unidirectional arrays with delayed input and internal state registers)
Given a functional program describing $F_{U_{0,n-1}}[n, X]$

find $Q_{0,n-1}$, which is a permutation of $U_{0,n-1}$, the initialization value for the internal state registers: r_0, y^0 (such that $Y^0 = (y^0)^\infty$) and the (nontrivial part of the) transition function, $f = \langle f_y, f_r \rangle$ such that

$$F_{Q_{0,n-1}}[n, X] = \vec{f_y}[r_0, q_{n-1}, X_{-(n-1)}, X_{-n}, F_{Q_{0,n-2}}[n-1, X]] \quad (5.41)$$

$$F_{q_0}[1, X] = \vec{f_y}[r_0, q_0, X, X_{-1}, Y^0] \ , \quad (5.42)$$

where the function $\vec{f_y}[\langle r, q \rangle, \langle X, X', Y \rangle]$ satisfies condition (5.20).

Example 5.11. - *multiplication of two fixed size integers*
Let $A_{0,n-1} = \langle a_0, a_1, a_2 \ldots, a_{n-1} \rangle$ and $B_{0,m-1} = \langle b_0, b_1, b_2 \ldots, b_{m-1} \rangle$ be the list representation of two fixed size integers. A represents $a = a_0 + a_1\beta + a_2\beta^2 + \ldots + a_{n-1}\beta^{n-1}$, where $\beta > 1$ is the radix for integer representation.

We want to compute the product of the two integers: $c = a * b$.

The first task is to express the computation as a functional program, that is to express the computations using list representation and also find a recursive description for the function to be computed.

One of the input lists (let us take $B_{0,m-1}$) will be extended to an infinite list by adding an infinite number of trailing zeroes, which does not change the value of the represented number.
$B = B_{0,m-1} \frown 0^\infty = \langle b_0, b_1, \ldots b_{m-1}, 0, 0, 0, \ldots \rangle$.

There are more solutions to express the product recursively. We use the

following idea:

$$\begin{aligned}
a * b &= \overline{a_{n-1}a_{n-2}\dots\ a_1a_0} * \overline{b_{m-1}b_{m-2}\dots b_1b_0} = \\
&= a_0 * \overline{b_{m-1}b_{m-2}\dots b_1b_0} + \\
&+ a_1 * \overline{b_{m-1}b_{m-2}\dots b_1b_00} + \\
&\dots \\
&+ a_{n-1} * \overline{b_{m-1}b_{m-2}\dots b_1b_0 \underbrace{00\dots0}_{n-1 \text{ times}}} = \\
&= a_{n-1} * \overline{b_{m-1}b_{m-2}\dots b_1b_0 \underbrace{00\dots0}_{n-1 \text{ times}}} + \\
&+ \overline{a_{n-2}\dots\ a_1a_0} * \overline{b_{m-1}b_{m-2}\dots b_1b_0}
\end{aligned}$$

Using list representation $\overline{b_{m-1}b_{m-2}\dots b_1b_00}$ is expressed by B_{-1}, where the introduced blank value is 0. In the same way B_{-i}, where $i > 0$ represents the number b shifted with i positions to the left ($B_{-i} = 0^i \smile B$).

The product of the two numbers can be written as

$$F_{A_{0,n-1}}[n, B] = \sum_{j=0}^{n-1} a_j * B_{-j} \ ,$$

where the sum operator applied on lists does not stand for the simple addition of lists but involves the addition with carry propagation.

The function that defines the addition with carry propagation of two lists (denoted by $\vec{ac}$) is defined in the following way:

$$\vec{ac}[r, u \dot{\smile} U, v \dot{\smile} V] = (r + u + v)_{mod\beta} \dot{\smile} \vec{ac}\left[\left\lfloor \frac{r+u+v}{\beta} \right\rfloor, U, V\right] \quad (5.43)$$

Note that the $\vec{ac}$ function has property (5.20).
Because $F_{A_{0,n-1}}[n, B] = \vec{ac}[0, a_{n-1} * B_{-(n-1)}, \sum_{j=0}^{n-2} a_j * B_{-j}]$, we can write

$$F_{A_{0,n-1}}[n, B] = \vec{ac}[0, a_{n-1} * B_{-(n-1)}, F_{A_{0,n-2}}[n-1, B]],$$

which is of the form of (5.41).

From here we conclude that the initialization value for the internal state register r that stores the carry is 0. $Q_{0,n-1}$ of (5.41) is equal to $A_{0,n-1}$, and the transition function $f = \langle f_y, f_r \rangle$ is:

$$\begin{cases} f_y[r, a, x, dx, y] &= (r + a * x + y)_{mod\beta} \\ f_r[r, a, x, dx, y] &= \lfloor \frac{r+a*x+y}{\beta} \rfloor \end{cases}$$

Verifying condition (5.42) we have:

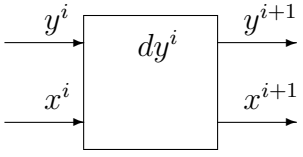
$$a_0 * B = \vec{ac}[0, a_0 * B, Y^0] \ ,$$

from here $y^0 = 0$ and $Y^0 = 0^\infty$.

5.4.3 Unidirectional Arrays with Delayed Output and Input "Pass-Through"

We consider here unidirectional linear arrays, where the input is "passing through" the array, in the same way as described in section 5.4.1, but the output is delayed with one or more time steps. Delay elements, denoted by $dy|1|, \ldots dy|k|$ are introduced, that temporarily store the y values for k time steps. We describe in the sequel in detail the case when the output is delayed by one time step.

Figure 5.15. describes the computations performed by a typical PE of such an array. Here the computation of x and y belongs to the trivial part of the transition function.



Computations:

$$\begin{aligned} dy^i_{t+1} &= f[x^i_t, y^i_t] \\ y^{i+1}_{t+1} &= dy^i_t \\ x^{i+1}_{t+1} &= x^i_t \end{aligned}$$

Figure 5.15: Computation of a PE in an array with delayed output

On Fig. 5.16. the initialization of the array is presented.

In case of such an array the first $n-1$ results that leave the array at PE_{n-1} depend only on the initialization values, but even if the n^{th} element of the result already involves elements of the input X, the first intermediary result computed at PE_0 in the first time step will be part of the result that leaves the array at PE_{n-1} only at the time instant $2n$.

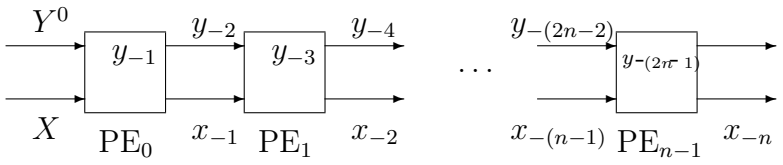


Figure 5.16: Initialization of the array with delayed output

The first $2n-1$ elements of the result are:

$$
\begin{array}{lcll}
y_1^n & = & dy_0^{n-1} & = y_{-(2n-1)} \\
y_2^n & = & dy_1^{n-1} & = f[x_0^{n-1}, y_0^{n-1}] = f[x_{-(n-1)}, y_{-(2n-2)}] \\
y_3^n & = & f[x_1^{n-1}, y_1^{n-1}] & = f[x_0^{n-2}, dy_0^{n-2}] = f[x_{-(n-2)}, y_{-(2n-3)}] \\
\ldots & & & \\
y_{2n-2}^n & = & \ldots & = \underbrace{f[x_{n-3}, f[x_{n-4}, \ldots f}_{n-1 \text{ times}}[x_{-1}, y_{-2}] \ldots] \\
y_{2n-1}^n & = & \ldots & = \underbrace{f[x_{n-2}, f[x_{n-3}, \ldots f}_{n-1 \text{ times}}[x_0, y_{-1}] \ldots]
\end{array}
$$

The common characteristic of the results that appear beginning with the $2n^{th}$ time instant is that they all involve the application of the transition function f n times:

$$
\begin{array}{lcl}
y_{2n}^n & = & \ldots = \underbrace{f[x_{2n-2}^{n-1}, f[x_{2n-4}^{n-2}, \ldots, f}_{n \text{ times}}[x_0^0, y_0^0]] \quad = \\
 & = & \underbrace{f[x_{n-1}, f[x_{n-2}, \ldots, f}_{n \text{ times}}[x_0, y_0]] \\
\ldots & & \\
y_{2n+i}^n & = & \ldots = \underbrace{f[x_{n-1+i}, f[x_{n-2+i}, \ldots, f}_{n \text{ times}}[x_i, y_i]]
\end{array}
$$

The equation that characterises the array is of the following form:

$$
Y_{2n} = \underbrace{\vec{f}[X_{n-1}, \vec{f}[X_{n-2}, \ldots \vec{f}}_{n \text{ times}}[X, Y^0] \ldots] \ . \tag{5.44}
$$

Design Problem 5.7. (Unidirectional arrays with delayed output)
Given a functional program describing $F[n, X]$

find y^0 (where $Y^0 = (y^\infty)$ and the (nontrivial part of the) transition function, f, such that

$$F[n, X] = \vec{f}[X_{n-1}, F[n-1, X]] \quad (5.45)$$
$$F[1, X] = \vec{f}[X, Y^0] \ , \quad (5.46)$$

where $\vec{f}$ has property (5.16).

In case if the PEs of the array also have a constant internal state register q (see the description in section 5.4.1), the equation that characterises the array will be the following:

$$Y_{2n} = \underbrace{\vec{f}[q_{n-1}, X_{n-1}, \vec{f}[q_{n-2}, X_{n-2}, \ldots \vec{f}}_{n \text{ times}}[q_0, X, Y^0] \ldots] \quad (5.47)$$

Design Problem 5.8. (Unidirectional arrays with delayed output and constant internal state registers)
Given a functional program describing $F_{U_{0,n-1}}[n, X]$

find $Q_{0,n-1}$ which is a permutation of $U_{0,n-1}$, y^0, (where $Y^0 = (y^0)^\infty$) and the (nontrivial part of the) transition function f such that

$$F_{Q_{0,n-1}}[n, X] = \vec{f}[q_{n-1}, X_{n-1}, F_{Q_{0,n-2}}[n-1, X]] \quad (5.48)$$
$$F_{q_0}[1, X] = \vec{f}[q_0, X, Y^0] \ , \quad (5.49)$$

where $\vec{f_q}[\langle X, Y \rangle]$ has property (5.16).

Example 5.12. - *Convolution of a finite and an infinite sequence*
We are now revisiting the problem presented in example 5.10.

We use the following notation:
$W = X_{-(N-1)}$ and $F_{Q_{0,n-1}}[n, W] = \sum_{j=N-n}^{N-1} a_j * W_{N-1-j}$, where $Q_{0,n-1}$ is a permutation of $A_{0,n-1}$. Equation (5.40) can be written in the following way:

$$F_{Q_{0,n-1}}[n, W] = a_{N-n} * W + \sum_{j=N-n}^{N-1} a_j * W_{N-1-j} = a_{N-1-n} * W + F_{Q_{0,n-2}}[n-1, W],$$

which is of the form (5.48) if we choose $Q_{0,n-1} = \langle a_{n-1}, \ldots, a_1, a_0 \rangle$. Note that for $N = n$ the computation performed by the function $F_{Q_{0,n-1}}[n, W]$ is equivalent with (5.40).

By projection we get the transition function $f[w, q, y] = q*w + y$. Then we verify condition (5.49):

$$F_{q_0}[1, W] = \vec{f}[W, q_0, Y^0]$$

That is $q_0*W = q_0*W + Y^0 \Rightarrow y^0 = 0$ and $Y^0 = \langle 0, 0, \ldots \rangle$.

In the even more complex situation when the PE also has internal state register(s) we can formulate the design problem in the following way.

Design Problem 5.9. (Unidirectional arrays with delayed output and internal state registers)
Given a functional program describing $F_{U_{0,n-1}}[n, X]$

find $Q_{0,n-1}$, which is a permutation of $U_{0,n-1}$, the initialization value for the internal state registers: r_0, y^0 (such that $Y^0 = (y^0)^\infty$) and the (nontrivial part of the) transition function, $f = \langle f_{dy}, f_r \rangle$, such that

$$F_{Q_{0,n-1}}[n, X] = \vec{f}_{dy}[r_0, q_{n-1}, X_{n-1}, F_{Q_{0,n-2}}[n-1, X]] \quad (5.50)$$

$$F_{q_0}[1, X] = \vec{f}_{dy}[r_0, q_0, X, Y^0] \ , \quad (5.51)$$

where the function $\vec{f}_{dy}[\langle r, q \rangle, \langle X, Y \rangle]$ satisfies condition (5.20).

Example 5.13. - *Multiplication of fixed size integers*
We are revisiting the problem presented in example 5.11.

We use the same idea and the notation $X = 0^{n-1} \frown B$. Now the function to be computed can be written in the following form:

$$F_{Q_{0,n-1}}[n, X] = \sum_{j=0}^{n-1} a_{n-1-j} * X_j \ ,$$

where $Q_{0,n-1}$ is a permutation of $A_{0,n-1}$ and the sum operator again involves the addition with carry propagation (5.43).

Because $F_{Q_{0,n-1}}[n, B] = \vec{ac}[0, a_0 * X_{n-1}, \sum_{j=0}^{n-2} a_{n-1-j} * X_j]$, we can write

$$F_{Q_{0,n-1}}[n, X] = \vec{ac}[0, q_0 * X_{n-1}, F_{Q_{0,n-2}}[n-1, X]] \ ,$$

with $Q_{0,n-1} = \langle a_{n-1}, a_{n-2}, \ldots, a_1, a_0 \rangle$, which is of the form (5.50).

From here we get the (unknown part of the) transition function $f = \langle f_{dy}, f_r \rangle$:

$$\begin{cases} f_{dy}[r, q, x, dy, y] = (r + q * x + y)_{mod\beta} \\ f_r[r, q, x, dy, y] = \lfloor \frac{r+q*x+y}{\beta} \rfloor \end{cases}$$

The initialization value for the internal state register r that stores the carry is 0. By verifying condition (5.51) we conclude that $Y = 0^{\infty}$.

Note: we added a number of $n-1$ zeroes to the input, thus we obtained a recursive description of the product that involves the application of the function $\vec{ac}$ n times. Because in the case of integer multiplication the first $n-1$ elements of the result can be actually computed by applying the same function only $1, 2, \ldots, n-1$ times respectively, the resulted array can be optimised by using directly B as input instead of $X = 0^{n-1} \smile B$.

5.4.4 Unidirectional Pass–Through Array

We present here a very simple type of unidirectional array consisting of processing elements (PEs) that modify the input and send its elements to the next PE.

Figure 5.17 presents the functional view of such an array: we can say that it is composed of a head processor (PE_0), while the rest of PEs form the tail array (the part of the array marked with dashed line), the functioning of which is similar to the whole array.

The list X is the global input to the array (its elements are fed into the array through PE_0 at each time step). If the array computes the function $F[X]$ and the head processor PE_0 outputs the modified list $K[X]$ that satisfies property (5.16), then the tail array will compute $F[K[X]]$.
Y is the list of results.

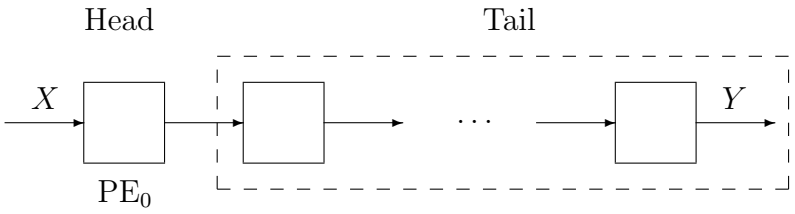


Figure 5.17: Unidirectional Pass-Through array

The recursive equation that characterises the functioning of the array is of the following form

$$F[X] = F[G[X]] \ , \tag{5.52}$$

where G is an online transitive function.

Specification:
The design problem consists in finding the scalar projection of G when a recursive description of the form (5.52) of F is given.

Design method:
Determine $G[X]$ from the recursive description of F.

The problem reduces to the design problem of a single systolic processor that computes G.

The condition for termination has to be analyzed separately. Note that the equation (5.52) does not tell anything about the termination of the problem. We either know the number of iterations (and from here we can conclude the number of PEs) or a special termination condition is given, that also depends on the input, then the transition function should also verify whether this condition holds.

5.4.5 Unidirectional Array for GCD Computation – Case Study

In order to demonstrate the use of our functional–based method on a more complex example, we give here the detailed description of the design steps of a unidirectional pass-through array which computes the greatest common divisor (GCD) of multiple precision integers. We presented this case study in [RJ06].

Definition of the algorithm for GCD Computation

Let $a = a_0 + 2 * a_1 + 2^2 * a_2 + \ldots$ and $b = b_0 + 2 * b_1 + 2^2 * b_2 + \ldots$ be two integers expressed in radix two[1]. We use the lists of digits as representations of the numbers, and we assume that the least significant digits are at the beginning. The lists may be considered infinite, since padding zeroes at the end of the finite representation does not change the value of the number. In fact, the GCD algorithm below may produce intermediate negative values, and it is designed for numbers in *complement representation*. The representation of a negative number will be padded with ones.

The *PlusMinus* algorithm for GCD computation (from [Jeb94a], which improves [BK85], which is based on [Ste67]) proceeds in three steps:

[1]The *PlusMinus* GCD algorithm has be generalised to representations using as radix an arbitrary power of two [Jeb93], which can be implemented on a systolic array in the same manner as shown here – but this version would unnecessarily complicate the presentation.

First step: We remove the common least significant null digits of a and b and obtain a_s and b_s, that is, we divide both a and b by the same power of 2, say 2^k. The GCD of these numbers, $GCD' = GCD(a_s, b_s)$ is called the "pseudo-GCD" of a and b, and obviously

$$GCD = 2^k * GCD'.$$

Moreover, GCD' is not divisible by 2.

Our systolic device will perform this operation, and then compute the pseudo-GCD, but will not handle the computation of k, neither the multiplication by 2^k. In a practical situation, it is reasonable to assume that the systolic array is under the control of some main device (either a usual computer or some other complex hardware), which supplies the input operands and collects the result. We assume that the final multiplication (in fact a shift) by 2^k is performed by this main device, and also one sees that it is straightforward to add to the systolic array a counter which identifies the value of k and sends it to the main device.

Second step: We interchange a_s and b_s if necessary, such that the least significant digit of a_s is 1. This is an invariant throughout the rest of the algorithm, which simplifies the operations.

Third step: We calculate now the pseudo GCD of a and b (the a and b values from the following algorithm are actually the a_s and b_s obtained in the previous steps).

$$\begin{aligned} GCD[a,b] &= a, \text{ if } b = 0 && (5.53)\\ &= GCD\left[a, \frac{b}{2}\right], \text{ if } b_0 = 0 && (5.54)\\ &= GCD\left[b, \frac{a+b}{4}\right], \text{ if } (b_0 = 1) \wedge (a_1 \neq b_1) && (5.55)\\ &= GCD\left[b, \frac{a-b}{4}\right], \text{ if } (b_0 = 1) \wedge (a_1 = b_1) && (5.56) \end{aligned}$$

The last three transformations are correct because the pseudo GCD is not divisible by 2. Note also that these transformations preserve the invariant $(a_0 = 1)$, thus a cannot become 0 (therefore $GCD[0, b] = b$ is not necessary).

Termination follows from the decrease of maximum of the significant lengths of a and b. Indeed, by adding or subtracting the arguments in the respective cases, the last two bits of the numbers always become zero. Since the sum or the difference will be at most one bit longer, after two shifts it will become one bit shorter. (The full details of the analysis are presented in [Jeb94b].)

The reader may note that the usage of complement arithmetic is essential for the correct implementation of this algorithm. Indeed, since only the least significant digits of a and b are inspected, in cannot be known whether $a-b$ is negative or positive, thus at a certain moment one or both of the arguments will be negative.

Design steps of the Unidirectional Array for GCD Computation

In the previous section we described the algorithm for GCD computation. In the sequel we describe the systematic design of a unidirectional systolic array, able to solve the problem.

Let $A = \langle a_0, a_1, \ldots \rangle$ and $B = \langle b_0, b_1, \ldots \rangle$ be the list representation of the two integers, a respectively b.

Step 1 can be computed by a single processor (see example 5.5).

Step 2 can also be computed by a single processor (see example 5.4).

Step3:

In equation (5.54) $b/2$ appears, which in case of an even binary number means a shift to the right of the digits of b. In the list representation this is equivalent to $T[B]$. In the same way the equivalent in the list notation for $b/4$ is the second tail of B, that is $T_2[B]$.

Using the list notation we can rewrite equations (5.53)-(5.56) in the following way:

$$\begin{array}{rcll} GCD[A,B] & = & A, & \text{if } B = 0 \\ & = & GCD[A, T[B]], & \text{if } H[B] = 0 \\ & = & GCD[B, T_2[A+B]], & \text{if } (H[B] = 1) \wedge (H_1[A] \neq H_1[B]) \\ & = & GCD[B, T_2[A-B]], & \text{if } (H[B] = 1) \wedge (H_1[A] = H_1[B]) \end{array}$$

The function GCD appears three times on the rhs. of the definition. Thus the computation of the GCD function can be also written in the following way:

$$\begin{array}{rcll} GCD[\langle A,B \rangle] & = & A, & \text{if } B = 0 \\ & = & GCD[& \\ & & \langle A, T[B] \rangle, & \text{if } H[B] = 0, \\ & & \langle B, T_2[A+B] \rangle, & \text{if } (H[B] = 1) \wedge (H_1[A] \neq H_1[B]), \\ & & \langle B, T_2[A-B] \rangle, & \text{if } (H[B] = 1) \wedge (H_1[A] = H_1[B]) \\ & &] & \end{array}$$

which is of the form (5.52) (except for the first equation, which is the stop condition and will be discussed later).

We know that if $a_0 = b_0 = 1$, then $T_2[A-B] = T_2[A]-T_2[B]$, when $a_1 = b_1$ and $T_2[A+B] = 1 \dot{+} T_2[A] + T_2[B]$, when $a_1 \neq b_1$. If we use the notation $X = \langle A, B\rangle$ and F_A and F_B are the already used functions that return the A respectively B component of the input, the K function computed by a PE is:

$$
\begin{aligned}
& K[H[X], H_1[X], X, T[X], T_2[X]] = \\
= \quad & \langle F_A[X], F_B[T[X]]\rangle && \text{if } F_B[H[X]] = 0 && (5.57)\\
= \quad & \langle F_B[X], 1 \dot{+} F_A[T_2[X]] + F_B[T_2[X]]\rangle && \text{if } (F_B[H[X]] = 1) \wedge && (5.58)\\
& && (F_A[H_1[X]] \neq F_B[H_1[X]]) && \\
= \quad & \langle F_B[X], F_A[T_2[X]] - F_B[T_2[X]]\rangle && \text{if } (F_B[H[X]] = 1) \wedge && (5.59)\\
& && (F_A[H_1[X]] = F_B[H_1[X]]) &&
\end{aligned}
$$

Equation (5.57) satisfies property (5.16), while the equations (5.58) and (5.59) satisfy property (5.18). The latter two equations induce the introduction of an internal state register, that we denote by r. It will store the carry from the addition (respectively subtraction) operation (for details see example 5.2). The carry is initialised with 1 respectively 0 when s is set, depending whether addition or subtraction has to be performed.

On the other hand we can conclude from the form of the definition of the K function (equations (5.57)-(5.59)) that the function can be computed by an auto-configurable processor with 2-delay. Two transition variables are introduced to delay the input. Because in the "if" statements appear only the first two elements of the input, the state register will be set at the time step when the second input is fed into the PE. The list of control signals will be $SI = \langle 0, 1, 1, 1, \ldots\rangle$.

Note that here we are not interested in indicating the appearance of the first result ($k = 2$ is known), but it is very important that the list of control signals SQ that leaves the processor PE_0 is an adequate sequence of control signals which are input to the next PE. This means that the control signals associated to the first element of the result should be 0, the other values 1. To obtain such a list of output signals one needs to delay the control signal, too by $k = 2$ elements (two transition registers for the control signal have to be included, as shown on Figure 5.18, in the same way as the inputs are delayed).

Stop condition: The condition $GCD[A, B] = A$ if $B = 0$ as it is can not be verified in a systolic way. We should provide some information about the length of the inputs. If the processor knows the beginning of the input (this can be deduced with the help of the control signal) and another signal

indicates the end of the input, then the processor can decide when the input is null (note that in our case this can be only the input B).

The indication of the size of the input can be achieved by associating an additional tag-bit to each binary digit of the input. The value of the tag-bit will be 0 for each significant digit of the binary number and 1 beginning with the sign bit of the number.

Now the input lists A and B both have two components $xx = \langle tx, x\rangle$, where tx is the tag-bit (that indicates the end of the input) and x is the digit. Let Tag, and Dig be the list functions that select from the input the list of tag-bits and digits, respectively.

Because A and B are binary numbers, in the expression $(A+B)/4$ ($T_2[A+B]$ in list notation) $A + B$ can have at most one more digit than the longer number out of A and B. This means that $(A + B)/4$ is at least one digit shorter than the longer number. This property assures the termination of the algorithm as shown in [Jeb94b].

The list of tag-bits that indicate the longer number out of A and B is determined by the expression $Tag[A] \wedge Tag[B]$ (where $\wedge$ is the bitwise and operation). That is why the tag-list corresponding to a number that is "one digit shorter" than the longer out of A and B is

$$T[Tag[A] \wedge Tag[B]] = T[Tag[A]] \wedge T[Tag[B]] \ .$$

The computation of the GCD can now be rewritten in the following way:

$$\begin{array}{rcll}
GCD[\langle A, B\rangle] & = & A, & \text{if } (H[Dig[B]] = 0)\wedge \\
 & & & \wedge(H_1[Tag[B]] = 1) \\
 & = & GCD[& \\
 & & \langle A, T[B]\rangle, & \text{if } (H[Dig[B]] = 0)\wedge \\
 & & & \wedge(H_1[Tag[B]] = 0), \\
 & & \langle B, & \\
 & & \langle T[Tag[A]] \wedge T[Tag[B]], & \\
 & & 1 \dot{+} T_2[Dig[A]] + T_2[Dig[B]]\rangle\rangle, & \text{if } (H[Dig[B]] = 1)\wedge \\
 & & \quad \wedge(H_1[Dig[A]] \neq H_1[Dig[B]]), & \\
 & & \langle B, & \\
 & & \langle T[Tag[A]] \wedge T[Tag[B]], & \\
 & & T_2[Dig[A]] - T_2[Dig[B]]\rangle\rangle, & \text{if } (H[Dig[B]] = 1)\wedge \\
 & & \quad \wedge(H_1[Dig[A]] = H_1[Dig[B]]) & \\
 & &] &
\end{array}$$

The final expression for K can be automatically projected into the scalar space -using the rules described in Sect. 5.3.4- to obtain the computations of a PE.

Figure 5.18 presents the structure of a PE for the 3rd step of the GCD computation. $dxx|j|$ (where x stands for either a or b and j is 1 or 2) denotes the transition register with two components: $\langle dtx|j|, dx|j|\rangle$, where the first component is the correspondent of the tag-bit and the second one corresponds to the digit. We use the notation xxi for the two components of the input $\langle ta_i, a_i\rangle$ or $\langle tb_i, b_i\rangle$.

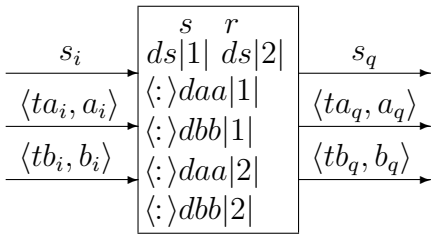


Figure 5.18: PE for the problem of GCD computation

Note that $T[Dig[X]] = Dig[T[X]]$ and $H[Dig[X]] = Dig[H[X]]$, which holds for the Tag function, too.

To avoid the repetition of the same computation for a and b, in the following description x stands for either a or b.

Computations of a PE:

$$
\begin{aligned}
\langle aq_{t+1}, bq_{t+1}\rangle &= \langle \$, \$\rangle \text{ if } s_t = \$ && (5.60)\\
&= \langle da|2|_t, \$\rangle \text{ if } s_t = 1 && (5.61)\\
&= \langle da|2|_t, db|1|_t\rangle \text{ if } s_t = 2 && (5.62)\\
&= \langle db|2|_t, low[ai_t \overset{\cdot\cdot}{+} bi_t \overset{\cdot\cdot}{+} r_t]\rangle \text{ if } s_t = 3 && (5.63)\\
&= \langle db|2|_t, low[ai_t \overset{\cdot\cdot}{-} bi_t \overset{\cdot\cdot}{-} r_t]\rangle \text{ if } s_t = 4 && (5.64)\\
\langle taq_{t+1}, tbq_{t+1}\rangle &= \langle \$, \$\rangle \text{ if } s_t = \$ && (5.65)\\
&= \langle dta|2|_t, \$\rangle \text{ if } s_t = 1 && (5.66)\\
&= \langle dta|2|_t, dtb|1|_t\rangle \text{ if } s_t = 2 && (5.67)\\
&= \langle dtb|2|_t, dta|1|_t \wedge dtb|1|_t\rangle \text{ if } (s_t = 3) \vee (s_t = 4) && (5.68)\\
r_{init} &= \$ && (5.69)\\
r_{t+1} &= 1 \text{ if } (s_t = \$) \wedge (si_t = 1) \wedge (db|1|_t = 1) \wedge && (5.70)\\
&\quad \wedge (bi_t \neq ai_t)\\
&= high[ai_t \overset{\cdot\cdot}{+} bi_t \overset{\cdot\cdot}{+} r_t] \text{ if } s_t = 3 && (5.71)\\
&= 0 \text{ if } (s_t = \$) \wedge (si_t = 1) \wedge (db|1|_t = 1) \wedge && (5.72)\\
&\quad \wedge (bi_t = ai_t)\\
&= high[ai_t \overset{\cdot\cdot}{-} bi_t \overset{\cdot\cdot}{-} r_t] \text{ if } s_t = 4 && (5.73)\\
&= r_t \text{ otherwise} && (5.74)\\
dxx|1|_{t+1} &= xxi_t && (5.75)\\
dxx|2|_{t+1} &= dxx|1|_t && (5.76)\\
sq_{t+1} &= ds|2|_t && (5.77)\\
ds|2|_{t+1} &= ds|1|_t && (5.78)\\
ds|1|_{t+1} &= si_t && (5.79)\\
s_{init} &= \$ && (5.80)\\
s_{t+1} &= 1 \text{ if } (s_t = \$) \wedge (si_t = 1) \wedge (db|1|_t = 0) \wedge && (5.81)\\
&\quad \wedge (tbi_t = 1)\\
&= 2 \text{ if } (s_t = \$) \wedge (si_t = 1) \wedge (db|1|_t = 0) \wedge && (5.82)\\
&\quad \wedge (tbi_t = 0)\\
&= 3 \text{ if } (s_t = \$) \wedge (si_t = 1) \wedge (db|1|_t = 1) \wedge && (5.83)\\
&\quad \wedge (bi_t \neq ai_t)\\
&= 4 \text{ if } (s_t = \$) \wedge (si_t = 1) \wedge (db|1|_t = 1) \wedge && (5.84)\\
&\quad \wedge (bi_t = ai_t)\\
&= s_t, \quad \text{otherwise} && (5.85)
\end{aligned}
$$

The result is given by the PE, where the state register s is set to 1.

Note that the stop condition only detects the termination of the transformations of the argument: from this point on, all the PE's must leave the arguments unchanged. Effective termination of the algorithm on a concrete systolic array is a closely related, but different issue. Namely, since a concrete systolic array has a finite number of PE's, the result (i. e. pseudo GCD) will be generated at the right-hand-side of the array only if the number of PE's exceeds the number of steps which are necessary for the particular arguments of the respective computation. In practice, this problem can be tackled in various ways, which are in fact the same for all the arrays of this type. One solution is to handle arguments of a maximum known size by an array having a number of processors superior to the upper bound of the number of steps. An alternative solution is to pick-up the output even if the computation is not finished, and then to re-enter it again into the re-initialised array. The latter allows to handle arguments of arbitrary type, even though the array has a fixed size, which is an interesting feature specific to the pass-through arrays.

5.5 Bidirectional Arrays

We study in this section bidirectional linear systolic arrays, that is arrays with two directional data-flow: left-to-right and right-to-left. Some of the arrays have two input PEs for the global input: the leftmost and the rightmost PE. In case of other bidirectional arrays the global input is introduced only at the leftmost PE. The left-to-right output of the rightmost PE may be fed in again into the array from right to left, in this case the rightmost PE is slightly different from the other PEs of the array (see Fig. 5.30).
The case of online–arrays, a special class of bidirectional arrays, is addressed separately in Sect. 5.6.

5.5.1 Bidirectional Array with One Directional "Pass-Through" Input

Figure 5.19 represents a bidirectional array with input list X and the list of results Y moving in opposite directions with the same velocity. $Y^n = Y^{init} = y^{\infty}_{init}$ denotes the list of initial values that contribute to the computation of the result. We denoted by x_i, respectively y_i, $-(n-1) \leq i \leq -1$ the initial values of the input, respectively output data channels.

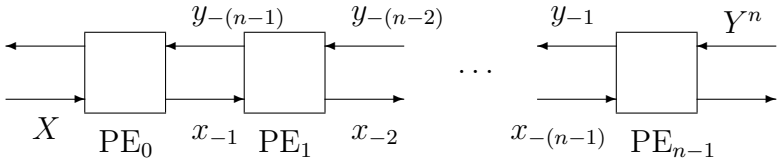
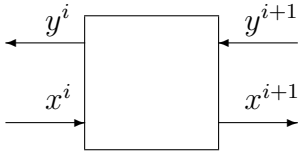


Figure 5.19: Bidirectional array with one "pass-through" input. Initial state.

The computations performed by a PE are presented in Fig. 5.20.
The list of results leaves the array at PE_0. The first result that involves y_{init} is y^0_n, that is the y value which leaves PE_0 at time step n:

$$y^0_n = \underbrace{f[X_{n-1}, f[X_{n-3}, \ldots, f[X_{-(n-3)}, f}_{n \text{ times}}[X_{-(n-1)}, y_{init}]] \ldots]$$

However the first result that involves only the elements of the input (no initial



Computations:

$$\begin{aligned} y^i_{t+1} &= f[x^i_t, y^{i+1}_t] \\ x^{i+1}_{t+1} &= x^i_t \end{aligned}$$

Figure 5.20: Computation of a PE in a bidirectional array with one input "pass-through"

value) appears only at time step $2n-1$:

$$\begin{aligned} y^0_{2n-1} &= \underbrace{f[x_{2n-2}, f[x_{2n-4}, \ldots, f[x_2, f}_{n \text{ times}}[x, y_{init}]] \ldots] \\ &\vdots \\ y^0_{2n-1+i} &= \underbrace{f[x_{2n-2+i}, f[x_{2n-4+i}, \ldots, f[x_{2+i}, f}_{n \text{ times}}[x_i, y_{init}]] \ldots] \\ &\vdots \end{aligned}$$

In the more concise list notation this is

$$Y^0_{2n-1} = \underbrace{\vec{f}[X_{2n-2}, \vec{f}[X_{2n-4}, \ldots, \vec{f}[X_2, \vec{f}}_{n \text{ times}}[X, Y^{init}]] \ldots] \ .$$

The functioning of the array can now be formulated in a functional way. Let $F[n, X] = Y^0_{2n-1}$, then

$$F[n, X] = \vec{f}[X_{2n-2}, F[n-1, X]] \tag{5.86}$$

$$F[1, X] = \vec{f}[X, Y^{init}] \tag{5.87}$$

Note that $y_i, i \geq 2n-1$ depends on $x_{i-1}, x_{i-3}, \ldots, x_{i-(2n-1)}$, that is it depends on $P_n[X_{i-(2n-1)\{2\}}]$. This means that the list of results computed by such an array can be split into two distinct parts, the elements of which can be computed using two independent lists. $Y^0_{2n-1\{2\}}$ depends on the elements of $X_{\{2\}}$ while $Y^0_{2n\{2\}}$ depends on the elements of $X_{1\{2\}}$.
Let F' be a list function such that $Y^0_{2n-1\{2\}} = Step_2[F[n, X]] = F'[n, X_{\{2\}}]$.

The functional description of the computation of $Y^0_{2n-1\{2\}}$ is

$$F'[n, X_{\{2\}}] = \vec{f}[X_{2n-2\{2\}}, F'[n-1, X_{\{2\}}]] \quad (5.88)$$
$$F'[1, X_{\{2\}}] = \vec{f}[X_{\{2\}}, Y^{init}] \ . \quad (5.89)$$

Let $X_{\{2\}} = W$. Note that

$$Step_2\ T_{2k}\ X = T_k\ Step_2\ X = T_k\ X_{\{2\}} = T_k\ W = W_k \ .$$

Thus (5.88)-(5.89) can be written in the following form

$$F'[n, W] = \vec{f}[W_{n-1}, F'[n-1, W]] \quad (5.90)$$
$$F'[1, W] = \vec{f}[W, Y^{init}] \ , \quad (5.91)$$

which is of the form of (5.45)-(5.46), the functional description that characterises unidirectional arrays with input "pass-through" and delayed output.

In the same way for $Y^0_{2n\{2\}} = Step_2[T[F[n, X]]] = F'[n, X_{1\{2\}}]$, with $X_{1\{2\}} = V$ we get a functional description of the same form.

Note: Let us define the function I that interleaves two lists as

$$I[a \overset{\cdot}{\smile} A, b \overset{\cdot}{\smile} B] = \langle a, b\rangle \smile I[A, B] \ .$$

For $X = I[W, V]$ the array computes simultaneously two identical (independent) problems, $F'[n, W]$ and $F'[n, V]$, where F' is of the form (5.45)-(5.46). We get the results every second time step, the elements of $F'[n, W]$ leave the array beginning with time step $2n-1$, and those of $F'[n, V]$ starting with time step $2n$.

It turns out that a bidirectional array with one directional pass-through input can in fact solve the same class of problems as a unidirectional pass-through array with delayed output. The difference is that in case of the unidirectional one we get the elements of the result at each time step, starting with time step $2n$, while the bidirectional array solves simultaneously two problems of the same type and the elements of the results of these two problems appear interleaved, at each second time step, starting with time step $2n-1$ respectively $2n$. This means that if we want to solve a single problem, we can do it twice as faster if we use the unidirectional array, but if one solves the same type of problem (with different input) repeatedly, then we cannot spare time with either of them, because the throughput of the two arrays is the same.

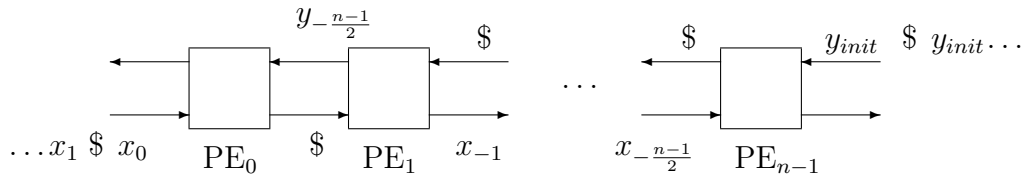


Figure 5.21: Bidirectional array with sparse input (n is odd). Initial state.

Figure 5.21 depicts the bidirectional array with sparse input $X_{\$}$ for solving a single problem of type (5.90)-(5.91). The array is an inefficient one, because the PEs are idle at each second time step.

There are common techniques for the optimization of arrays with sparse input. Some ideas are presented in Sect. 5.5.4.

Note:

- If we assume that $f[\$, y] = y$ (that is, if the input of the PE is a blank value, then the output is not changed) and $y_i = y_{init}, \forall -(n-1) \leq i \leq -1$ (the initial values of the output channels and the initial values y_{init} are the same), then the array can be reduced to a single PE (see the considerations from Sect. 5.6.1).

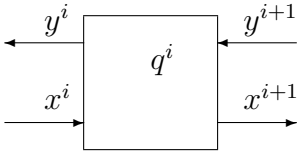
- If we are only interested in the elements of the results which involve only the input values of X then the computation of these elements start only after n timesteps. Thus we can consider the first $n-1$ steps the initialisation steps, while the real computations begin at time step n. If we would like to study only the computation-part, then we can consider that the initialization values of the array on Fig. 5.19 are the first $n-1$ elements of X (that is $P_{n-1}[X]$) instead of $X_{-(n-1),-1}$, and the global input to the array is X_{n-1} instead of X.

5.5.2 Bidirectional Array with Internal State

We study separately the case of constant and variable internal state registers. In case of constant internal state, the values of the local memory variables can be considered as parameters of the problem, while in case of variable internal state values only a scalar is given, which serves az initial value of the internal state registers.

Internal State Registers with Constant Value

Each PE has an internal state register, denoted by q, the value of which remains unchanged during the computations. For PE_i $(0 \leq i \leq n-1)$ we have $\forall t \geq 0$, $q_t^i = q^i = q_i$, where $Q_{0,n-1} = \langle q_0, q_1, \ldots q_{n-1} \rangle$ is the finite list of the internal state register values. These values can be considered as "parameters" of the problem solved by the array.



Computations:

$$\begin{aligned} y_{t+1}^i &= f[x_t^i, q_t^i, y_t^{i+1}] \\ x_{t+1}^{i+1} &= x_t^i \\ q_{t+1}^i &= q_t^i = q_i \end{aligned}$$

Figure 5.22: Computation of a PE in a bidirectional array with one input "pass-through" and constant internal state variables

The computations performed by a PE are presented in Fig. 5.22.
The $2n-1^{th}$ tail of the list of results (which consists of elements that does not depend on the initial values, only on the elements of the input) is:

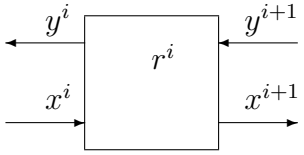
$$Y_{2n-1}^0 = \underbrace{\vec{f}[X_{2n-2}, q_0, \vec{f}[X_{2n-4}, q_1, \ldots, \vec{f}[X_2, q_{n-2}, \vec{f}}_{n \text{ times}}[X, q_{n-1}, Y^{init}]] \ldots] \ .$$

We can conclude that such an array can solve simultaneously two problems of the form (5.48)-(5.49), which depend on the same set of parameters $W_{0,n-1} = \langle q_{n-1}, q_{n-2}, \ldots, q_1, q_0 \rangle$.

Internal State Registers with Variable Value

After discovering the connection between uni- and bidirectional arrays without internal state registers respectively with constant internal state registers, it would be interesting to find out what is the relation between a bidirectional array with variable internal state registers and a corresponding unidirectional one with delayed output. The internal state register with variable value is denoted by r. Figure 5.23 presents the computations performed by a PE.

We assume that the initial value for r^i is r_0, $\forall 0 \leq i \leq n-1$ and $f_r[r_0, \$, \$] = f_r[r_0, \$, y] = f_r[r_0, x, \$] = r_0$, that is r^i will not change its value if at least one of the inputs of PE_i is a blank value.



Computations:

$$\begin{aligned} y^i_{t+1} &= f_y[x^i_t, r^i_t, y^{i+1}_t] \\ r^i_{t+1} &= f_r[x^i_t, r^i_t, y^{i+1}_t] \\ x^{i+1}_{t+1} &= x^i_t \end{aligned}$$

Figure 5.23: Computation of a PE in a bidirectional array with one input "pass-through" and variable internal state registers

The firs result which involves the application of the transition function n times will appear at PE_0 after n time steps:

$$\begin{aligned} y^0_n &= f_y[r^0_{n-1}, x^0_{n-1}, y^1_{n-1}] = f_y[r^0_{n-1}, x^0_{n-1}, f_y[r^1_{n-2}, x^1_{n-2}, y^1_{n-1}]] = \\ &= \ldots = \underbrace{f_y[r^0_{n-1}, x^0_{n-1}, f_y[r^1_{n-2}, x^1_{n-2}, \ldots f_y}_{n \text{ times}}[r^{n-1}_0, x^{n-1}_0, y^n_0] \ldots] \end{aligned}$$

Note that if the input and output channels of the array are initialised with blanks ($x_i = y_i = \$, \forall i, -(n-1) \leq i \leq -1$), then $r^i_k = r_0$, $\forall k, 0 \leq k \leq n-1-i$.

We consider as results the elements of the output starting with time step n. Thus the list of results, Y^0_n is:

$$Y^0_n = \underbrace{\vec{f_y}[R^0_{n-1}, X_{n-1}, \vec{f_y}[R^1_{n-2}, X_{n-3}, \ldots \vec{f_y}}_{n \text{ times}}[R^{n-1}_0, X_{-(n-1)}, Y^n_0] \ldots] \ , \quad (5.92)$$

where the R lists are generated by the scalar function f_r involving the corresponding X respectively Y lists, as follows:

Let F_R be a mixed-type function and f_r a scalar function such that

$$F_R[r, x \overset{\cdot}{\smile} X, y \overset{\cdot}{\smile} Y] = f_r[r, x, y] \overset{\cdot}{\smile} F_R[f_r[r, x, y], X, Y] \ .$$

Then

$$R^i_{n-1-i} = r_0 \overset{\cdot}{\smile} F_R[r_0, X_{n-1-2i}, Y^{i+1}_{n-1-i}] \ .$$

We denote by $\vec{f'_y}$ the mixed type function characterised by (5.4), then (5.92) can be written in the more concise form:

$$Y^0_n = \underbrace{\vec{f'_y}[r_0, X_{n-1}, \vec{f'_y}[r_0, X_{n-3}, \ldots \vec{f'_y}}_{n \text{ times}}[r_0, X_{-(n-1)}, Y^n_0] \ldots] \ . \quad (5.93)$$

Let $F'[n, r_0, X]$ be the function which computes the list of results. Then $F'[n, r_0, X] = Y_n^0$ can be expressed recursively in the following way:

$$F'[n, r_0, X] = \vec{f}'_y[r_0, X_{n-1}, F'[n-1, X_{-1}]] \quad (5.94)$$

$$F'[1, r_0, X] = \vec{f}'_y[r_0, X, Y_0^n] \ . \quad (5.95)$$

Design Problem 5.10. (Bidirectional arrays with input "pass-through" and internal state registers)
Given a functional program describing $F[n, X]$
find

- the initialization value for the internal state registers: r_0, and the mixed type function F' such that $F'[n, r, X] = F[n, X]$
- y_0 (where $Y^n = (y_0)^\infty$) and the (nontrivial part of the) transition function $f = \langle f'_y, f_r \rangle$ such that

$$F'[n, r_0, X] = \vec{f}'_y[r_0, X_{n-1}, F'[n-1, r_0, X_{-1}]] \quad (5.96)$$

$$F'[1, r_0, X] = \vec{f}'_y[r_0, X, Y^n] \ , \quad (5.97)$$

and $\vec{f}[r, \langle X, Y \rangle]$ satisfies property (5.20)

In this case we cannot split anymore the result into two independent parts as in the case of bidirectional arrays without internal state registers or with constant internal state registers. Indeed, the first element of Y_n^0 depends only on the elements of $X_{-(n-1)\$}$, but the r register values are also updated in function of these input elements, thus the next element of the result will already depend on each element of $X_{-(n-2)}$.

...

An interesting case: $f_r[r, x, y] = x$. ← idea to find a meaningful example. (a very (too) simple example (but maybe good enough for an example) which can be solved in this way: $F[n, X] = \sum_{i=0}^{2n} X_i$).

...

The question whether such an array can solve a problem of the same type as a corresponding unidirectional array, might be still interesting.

Let us consider the same array, but with sparse input $X_\$$. Because we assumed that f_r keeps the value of r unchanged, if at least one of the inputs is blank, it follows that each PE will update the value of its internal state register only at each second time step.

We assume that n is odd.
The list of results will now be the sparse list $Y^0_{n\$}$:

$$\begin{aligned} Y^0_{n\$} &= F'[n, r_0, X_\$] = \vec{f}'_y[r_0, (X_\$)_{n-1}, F'[n-1, r_0, (X_\$)_{-1}]] = \\ &= \vec{f}'_y[r_0, X_{\frac{n-1}{2}\$}, F'[n-1, r_0, \$ \overset{\bullet}{\smile} X_\$]] = \ldots = \\ &= \vec{f}'_y[r_0, X_{\frac{n-1}{2}\$}, \vec{f}'_y[r_0, T_{n-2}[\$ \overset{\bullet}{\smile} X_\$], \ldots, \vec{f}'_y[r_0, X_{-\frac{n-1}{2}\$}, Y^n_{0\,\$}] \ldots] = \\ &= \vec{f}'_y[r_0, X_{\frac{n-1}{2}\$}, \vec{f}'_y[r_0, X_{\frac{n-3}{2}\$}, \ldots, \vec{f}'_y[r_0, X_{-\frac{n-1}{2}\$}, Y^n_{0\,\$}] \ldots] \ . \end{aligned}$$

The list which contains the relevant elements of the result, that is $(Y^0_{n\$})_{\{2\}} = Y^0_n$, can be determined by applying Prop. 5.3 recursively:

$$Y^0_n = \vec{f}'_y[r_0, X_{\frac{n-1}{2}}, \vec{f}'_y[r_0, X_{\frac{n-3}{2}}, \ldots, \vec{f}'_y[r_0, X_{-\frac{n-1}{2}}, Y^n_0] \ldots] \ . \tag{5.98}$$

Let W be a list such that $X = T_{\frac{n-1}{2}} W$, that is $W = T_{-\frac{n-1}{2}} X$. Then from (5.98) we get:

$$\begin{aligned} Y^0_n &= \vec{f}'_y[r_0, W_{\frac{n-1}{2}+\frac{n-1}{2}}, \vec{f}'_y[r_0, W_{\frac{n-1}{2}+\frac{n-3}{2}}, \ldots, \vec{f}'_y[r_0, W_{\frac{n-1}{2}-\frac{n-1}{2}}, Y^n_0] \ldots] = \\ &= \vec{f}'_y[r_0, W_{n-1}, \vec{f}'_y[r_0, W_{n-2}, \ldots, \vec{f}'_y[r_0, W, Y^n_0] \ldots] \ , \end{aligned}$$

which can be expressed recursively in the following form

$$F'[n, r_0, W] = \vec{f}_y[r_0, W_{n-1}, F'[n-1, r_0, W]] \tag{5.99}$$

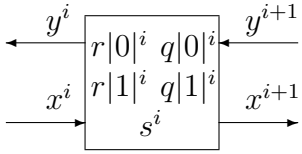
$$F'[1, r_0, W] = \vec{f}_y[r_0, W, Y^0] \ . \tag{5.100}$$

Note that (5.99)-(5.100) is also the description for the functioning of a unidirectional array with delayed output and variable internal state register. (See the similarity between (5.99)-(5.100) and (5.50)-(5.51). In the latter case we assumed that the PE also has constant internal state registers). This means that with a bidirectional array with sparse input and variable internal state registers we do can solve the same problem as a corresponding unidirectional array, the input of which is the input list of the array in the case of the bidirectional array, shifted by $(n-1)/2$ positions.

A bidirectional array with the same structure, having dense input, however, cannot solve two problems of the same type. For this purpose the PEs would need an additional internal state register, the values of which are updated alternately. A control sinal (initialised as $\langle 0, 1 \rangle^\infty$) can be used in order to determine which one of the two internal state registers should be updated.

Figure 5.24 depicts the structure and the computations of a PE which can solve simultaneously two different problems (buy having the same list of parameters $Q_{0,n-1}$) of the form (5.50)-(5.51).

See Example 5.13. With an array composed of PEs of Fig 5.24 we can perform simultaneously two multiplications: $a * b$ and $a * c$.



Computations:

$$
\begin{aligned}
y^i_{t+1} &= \begin{cases} f_y[x^i_t, r|0|^i_t, q|0|^i, y^{i+1}_t], & \text{if } s^i_t = 0 \\ f_y[x^i_t, r|1|^i_t, q|1|^i, y^{i+1}_t], & \text{if } s^i_t = 1 \end{cases} \\
r|k|^i_{t+1} &= \begin{cases} f_r[x^i_t, r|k|^i_t, q|k|^i, y^{i+1}_t], & \text{if } s^i_t = k \\ r|k|^i_t, & \text{if } s^i_t = 1-k \end{cases} \\
x^{i+1}_{t+1} &= x^i_t \\
s^i_{t+1} &= 1 - s^i_t \\
q|k|^i_{t+1} &= q|k|^i_t = q|k|^i \\
& \quad k = 0, 1
\end{aligned}
$$

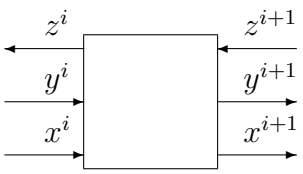
Figure 5.24: Computations of a PE in a bidirectional array with constant and variable internal state registers, which can compute two problems of the form (5.50)-(5.51).

5.5.3 Bidirectional Array with Two Directional "Pass–Through" Input

We consider here a bidirectional array with multiple input, where the input data flow is also bidirectional. One can see the computations of a PE on Fig. 5.25.

If X is the list of inputs to the array at the leftmost PE (that is PE_0) and Z is the input list introduced at PE $_{n-1}$, then the first elements of the two inputs will meet "in the middle", that is the first "relevant" computation will be performed at the time step $(n+1)/2$, if n is odd (respectively $n/2+1$, for n even), when x_0 meets z_0 at $\text{PE}_{(n-1)/2}$, (respectively x_0 meets z_1 at $\text{PE}_{n/2}$ and x_1 meets z_0 at $\text{PE}_{n/2-1}$). This would again lead to a list of results, that can be divided into two independent parts.

We will however start with a simpler input scheme, namely we will assume, that the two input lists are sparse and the elements of $Z_\$$ are preloaded



Computations:

$$\begin{aligned} y_{t+1}^{i+1} &= f[x_t^i, z_t^{i+1}, y_t^i] \\ x_{t+1}^{i+1} &= x_t^i \\ z_{t+1}^i &= z_t^{i+1} \end{aligned}$$

Figure 5.25: Computation of a PE in a bidirectional array with two directional input "pass-through"

into the array, thus the first relevant computation (computation which involves the elements of the given input list) will take place at PE_0 and we do not have to handle two distinct cases. The initialisation of the array is the one shown on Fig. 5.26.

$Y_{\$}^0 = \langle y^{init}, \$ \rangle^\infty$ is the sparse list of initial values, which contribute to the computation of the result.

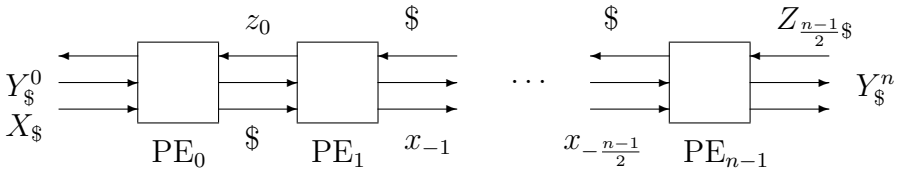


Figure 5.26: Bidirectional array with two directional sparse input. Initial state (n is odd).

The first result (which involves the application of the transition function, f, n times) will appear at PE_{n-1} after n time steps, and because of the sparse input we will get a new result at every second time step:

$$Y_{n\$}^n = \vec{f}[X_{\$}, Z_{(n-1)\$}, \vec{f}[X_{\$}, Z_{(n-2)\$}, \vec{f}[\ldots, \vec{f}[X_{\$}, Z_{1\$}, \vec{f}[X_{\$}, Z_{\$}, Y_{\$}^0]]\ldots] \ .$$

Or, if we consider the list of results, that is $Y_n^n = (Y_{n\$}^n)_{\{2\}}$:

$$Y_n^n = \vec{f}[X, Z_{n-1}, \vec{f}[X, Z_{n-2}, \vec{f}[\ldots, \vec{f}[X, Z_1, \vec{f}[X, Z, (y^{init})^\infty]]\ldots] \ .$$

Let F be the list function that computes Y_n^n, then F can be defined in a functional way as follows:

$$F[n, X, Z] = \vec{f}[X, Z_{n-1}, F[n-1, X, Z]] \quad (5.101)$$
$$F[1, X, Z] = \vec{f}[X, Z, Y^0] \ , \quad (5.102)$$

where $\vec{f}$ is an online transitive function.

"Mirrored" Array:

We can combine, as shown on Fig. 5.27, two arrays of the same type as the array from Fig. 5.26.

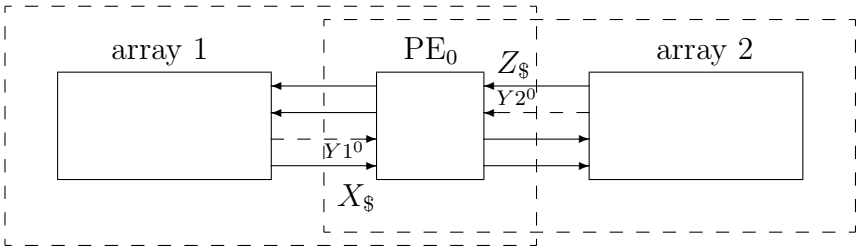


Figure 5.27: Mirrored array

Array1 has n processing elements: PE$_0$, PE$_1$, ..., PE$_{n-1}$, while array2 has m PEs, denoted by: PE$_0$, PE$_{-1}$, ..., PE$_{-(m-1)}$. Such an array computes simultaneously a problem of the form (5.101)-(5.102) and the symmetrical problem (obtained by interchanging the inputs X and Z), thus it has two lists of results.

Note that the composed array of Fig. 5.27 is not a regular systolic array in the sense that the elements of the list of initial values $Y1_\$^0$ for array2 should remain unchanged while passing the PEs of array1, they will, however, contribute to the computation of the results computed by array2. In the same way, the elements of $Y2_\$^0$ arrive unchanged to PE$_0$ after passing the PEs of array2 and will cotribute to the computations only in the PEs of array1 (see the channels denoted by the arrows with dashed line on Fig. 5.27).

Let us consider now a regular systolic array, which has the same interconnection pattern and initial state as the array shown on Fig. 5.27 and

the computations for the two output values $y1$ and $y2$, performed in PE_i $(\forall i, -(m-1) \leq i \leq n-1)$ are:

$$\begin{aligned} y1_{t+1}^{i+1} &= f1[x_t^i, z_t^{i+1}, y1_t^i, y2_t^{i+1}] \\ y2_{t+1}^{i} &= f2[x_t^i, z_t^{i+1}, y1_t^i, y2_t^{i+1}] \ . \end{aligned}$$

Fig. 5.28 presents the data-flow of the two input streams. In the initial phase $X_\$$ is input to PE_0 from left to right, while $Z_\$$ is input to PE_0 from right to left.

One can easily verify that at time step $t = i+j+1$ the input list to PE_{j-i} are $X_{i\$}$ respectively $Z_{j\$}$:

Indeed, at time step 1 $X_{0\$}$ respectively $Z_{0\$}$ is input to PE_0, hence $T_{-k}[X_\$]$ respectively $T_k[Z_\$]$ is the input list to PE_j. Both input streams are advancing with a velocity of one position pro time step, which means that at time step t the two input lists to PE_j will be $T_{t-1}T_{-k}X_\$ = T_{t-1-k}X_\$$ respectively $T_{t-1+k}Z_\$$. Now, with $t = i+j+1$ and $k = j-i$ we get
$T_{i+j+1-1-(j-i)}X_\$ = T_{2i}X_\$ = X_{i\$}$ respectively $T_{2j}Z_\$ = Z_{j\$}$ as input lists.

We assume that the two functions, which compute the partial results are the same, that is $f1 = f2 = f$. We denote by $C^{i,j}$ the list of results computed by PE_{j-i} starting with time step $t = i+j+1$ (and performing a useful computation at each second time step), when $X_{i\$}$ and $Z_{j\$}$ is input to the PE, as already shown. That is $C^{i,j} = Y1_{i+j+1}^{j-i} = Y2_{i+j+1}^{j-i}$.

The connection between the computations is represented on Fig. 5.29. The list of partial results computed at PE_{j-i} starting wit the time step $t = i+j+1$ can be expressed recursively in the following way:

$$C^{i,j} = \begin{cases} f[x_i, z_j, y1_i, y2_j] \overset{\cdot}{\smile} TC^{i,j} & \text{, if } i+j = 0 \\ \vec{f}[X_i, Z_j, C^{i,j-1}, C^{i-1,j}] & \text{, if } (-(m-1) \leq j-i \leq n-1) \wedge \\ & \quad (i+j+1 \geq 1) \\ \$ \overset{\cdot}{\smile} TC^{i,j} & \text{, otherwise} \end{cases}$$

where $\vec{f}$ is an online transitive function.

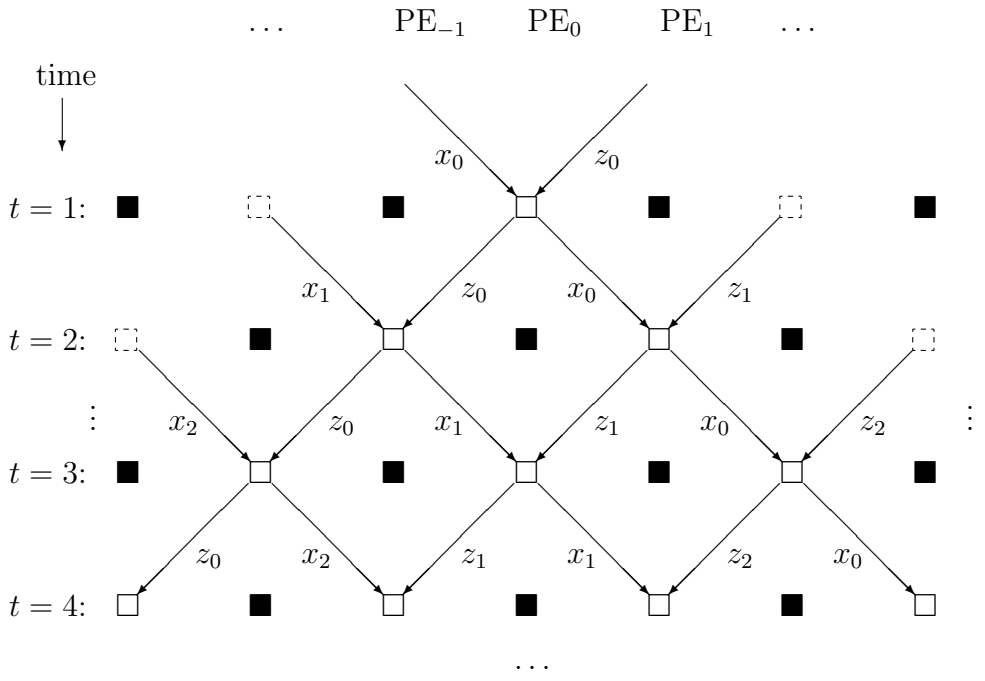
Note that $TC^{i,j}$ is the list of partial results computed a PE_{j-i} starting with time step $i+j+1+2$, that is:

$$TC^{i,j} = C^{i+1,j+1} \ .$$

...

Example 5.14. See the problem of sequences comparison from Sect. 4.3.

...



□ - active PE

⬚ - PE, where at least one of the inputs is a blank value

■ - inactive PE

Figure 5.28: Bidirectional array with bidirectional input and list of results. Data-flow.

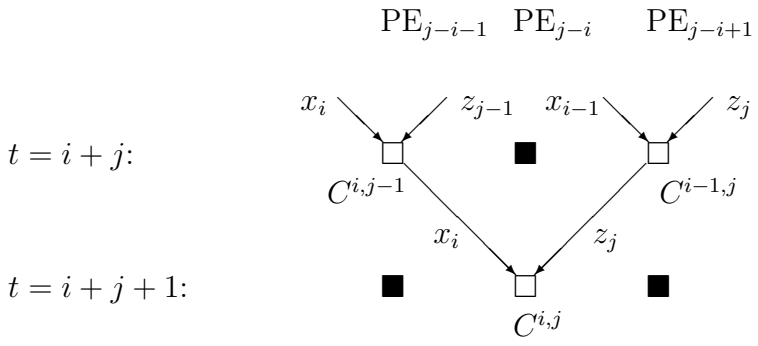


Figure 5.29: Data-flow. Fragment.

5.5.4 Transformations

We have studied in the previous sections arrays with bidirectional data-flow where the data from the two streams advances with the same velocity, but in opposite directions. We have seen that the elements of one of the data streams will meet only each second element of the other one, thus the array computes two independent problems (see the case of the arrays with one directional "pass-through" input without internal state in Sect. 5.5.1, respectively with constant internal state registers, Sect. 5.5.2).

We can use sparse input to ensure that a data will meet each of the elements of the data-flow advancing in the opposite direction, however these kind of arrays are working inefficiently, namely the PEs are idle in each second time step.

A commonly used idea is to merge two PEs which are working alternately (that is to map the computations of two different PEs which are working alternately onto one single PE) in order to transform the array into a more efficient one: if the array with sparse input had n PEs, then the same problem can be solved using only $n/2$ PEs.

One possibility is to merge two neighbouring PEs (PE$_i$ and PE$_{i+1}$) into one single PE. This is possible because two neighbouring PEs are active in alternating time steps, thus the merged PE will do the computations of PE$_i$respectively PE$_{i+1}$ in the even respectively odd time steps. In the sequel we present in detail a different solution based on the same idea. We assume that the number of PEs, n, is even (otherwise an additional "dummy" PE should introduced before merging two PEs).

"Folded" Array

Another interesting solution is to map the computations of PE_{n-1-i} onto PE_i, for $i = \frac{n}{2}, \ldots n-1$. We call the resulted array "folded" array, because the mapping process is like folding the inefficient array in the middle and merging the functionality of the overlapping PEs as shown on Fig 5.30.

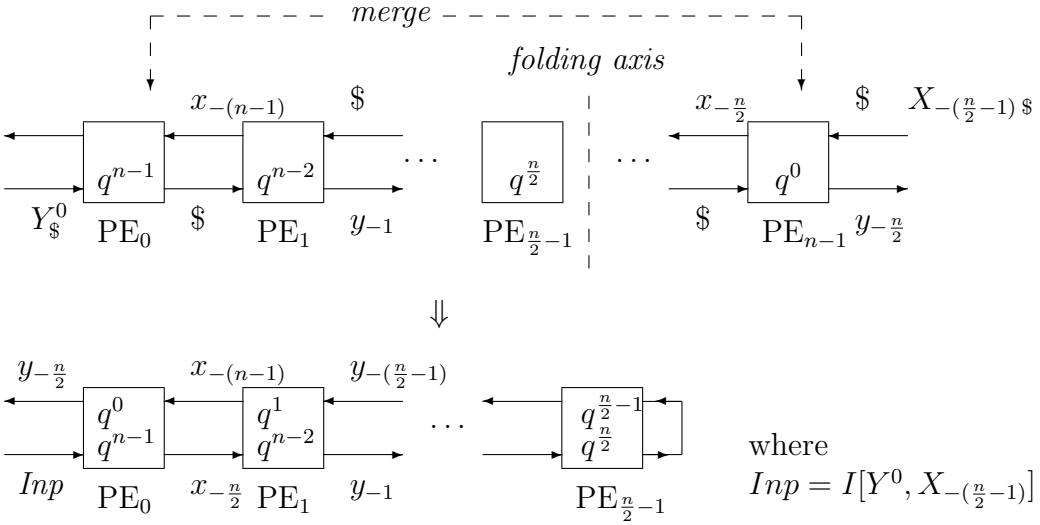


Figure 5.30: Bidirectional "folded" array.

Now each PE has to perform different computations in the successive time steps. The PEs are not aware of the time, but this problem can be solved with either the introduction of a control signal with alternating values, $\langle 0, 1, \rangle^\infty$, or with the help of an additional internal state register s. The s^i state register of PE_i, $0 \leq i \leq \frac{n}{2} - 1$ is initialised with the value s^i_0 such that:

$$s^i_0 = \begin{cases} 0 & \text{, if } i \text{ is even} \\ 1 & \text{, otherwise .} \end{cases}$$

The computations performed by a PE are shown on Fig. 5.31. The last PE, that is $\mathrm{PE}_{\frac{n}{2}-1}$ is slightly different from the other PEs of the array.
Let us consider again the function I which interleaves two lists:

$$I[a \overset{\bullet}{\smile} A, b \overset{\bullet}{\smile} B] = \langle a, b \rangle \smile I[A, B] \ .$$

The input list Inp can be defined as $Inp = I[Y^0, X_{-(\frac{n}{2}-1)}]$. The initial values of the input channels are:

$$\begin{aligned} a^k_0 &= H_{-k}[Inp], 0 \leq k \leq \tfrac{n}{2} \\ b^k_0 &= H_{-(n-k)}[Inp], 0 \leq k \leq \tfrac{n}{2} \end{aligned}$$

Computations:

	(different computations for $i = \frac{n}{2} - 1$)
$s_{t+1}^i = 1 - s_t^i$	
case $s_t^i = 0$	
$a_{t+1}^{i+1} = f[a_t^i, q^{n-1-i}, b_t^{i+1}]$	
$b_{t+1}^i = b_t^{i+1}$	$b_{t+1}^{\frac{n}{2}-1} = a_t^{\frac{n}{2}}$
case $s_t^i = 1$	
$b_{t+1}^i = f[b_t^{i+1}, q^i, a_t^i]$	$b_{t+1}^{\frac{n}{2}-1} = f[b_t^{\frac{n}{2}}, q^{\frac{n}{2}-1}, a_t^{\frac{n}{2}-1}]$
$a_{t+1}^{i+1} = a_t^i$	

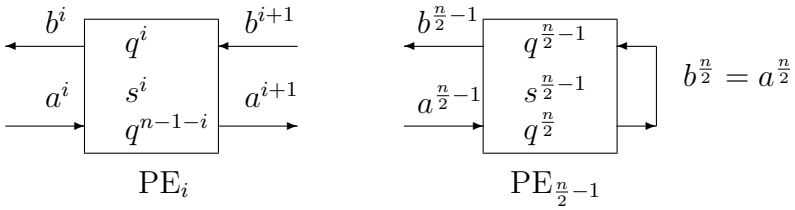


Figure 5.31: Computation of a PE in a bidirectional "folded" array.

Note that

$$H_i[I[A,B]] = \begin{cases} H_i[A_\$] = HT_iA_\$ = HA_{\frac{i}{2}} & \text{, if } i \text{ is even} \\ H_{i-1}[B_\$] = HT_{i-1}B_\$ = HB_{\frac{i-1}{2}} & \text{, if } i \text{ is odd .} \end{cases}$$

We consider, however, that the notation used on Fig. 5.32 is more expressive and it describes the functioning of a PE clearer. By convention we denoted an x value by $\bar{x}$, when it is input from right to left (see the upper input channel on Fig. 5.32 a)) and by $\underline{x}$, when it is input from left to right (Fig. 5.32 b)). In the same way, we use the notation $\bar{y}$ and $\underline{y}$ for the y values.

The connection between the two notations can be expressed in the following way:

$$\begin{aligned} a_t^i &= \begin{cases} \underline{y}_t^i & \text{, if } t \text{ is even } (s^i = 1) \\ \underline{x}_t^i & \text{, if } t \text{ is odd } (s^i = 0) \end{cases} \\ b_t^i &= \begin{cases} \bar{x}_t^i & \text{, if } t \text{ is even } (s^i = 1) \\ \bar{y}_t^i & \text{, if } t \text{ is odd } (s^i = 0) \end{cases} \end{aligned} \tag{5.103}$$

The first result appears after n time steps at PE_0:

$$\begin{aligned} \bar{y}_n^0 &= f[\bar{y}_{n-1}^1, q^0, \underline{x}_{n-1}^0] = f[f[\bar{y}_{n-2}^2, q^1, \underline{x}_{n-2}^1], q^0, \underline{x}_{n-1}^0] = \dots \\ &= \underbrace{f[f[\dots f}_{\frac{n}{2} \text{ times}}[\underline{y}_{\frac{n}{2}}^{\frac{n}{2}}, q^{\frac{n}{2}-1}, \underline{x}_{\frac{n}{2}}^{\frac{n}{2}-1}], q^{\frac{n}{2}-2}, \underline{x}_{\frac{n}{2}}^{\frac{n}{2}-1}] \dots] q^0, \underline{x}_{n-1}^0] = \\ &= \underbrace{f[f[\dots f}_{\frac{n}{2} \text{ times}}[f[\underline{y}_{\frac{n}{2}-1}^{\frac{n}{2}-1}, q^{n-1-(\frac{n}{2}-1)}, \bar{x}_{\frac{n}{2}-1}^{\frac{n}{2}}], q^{\frac{n}{2}-1}, \underline{x}_{\frac{n}{2}}^{\frac{n}{2}-1}] \dots] q^0, \underline{x}_{n-1}^0] = \dots \\ &= \underbrace{f[f[\dots f}_{\frac{n}{2} \text{ times}}[\underbrace{f[\dots f}_{\frac{n}{2} \text{ times}}[\underline{y}_0^0, q^{n-1}, \bar{x}_0^1], q^{n-2}, \bar{x}_1^2] \dots \\ &\quad \dots], q^{\frac{n}{2}}, \bar{x}_{\frac{n}{2}-1}^{\frac{n}{2}}], q^{\frac{n}{2}-1}, \underline{x}_{\frac{n}{2}}^{\frac{n}{2}-1}], \dots], q^0, \underline{x}_{n-1}^0] \overset{(5.103)}{=} \\ &= \underbrace{f[f[\dots f}_{\frac{n}{2} \text{ times}}[\underbrace{f[\dots f}_{\frac{n}{2} \text{ times}}[y^0, q_{n-1}, x_{-(n-1)}], q_{n-2}, x_{-(n-2)}] \dots \\ &\quad \dots], q_{\frac{n}{2}}, x_{-\frac{n}{2}}], q_{\frac{n}{2}-1}, x_{-(\frac{n}{2}-1)}] \dots] q_0, x_0] \end{aligned}$$

A new result appears after each second time step. The list of results is:

$$\bar{Y}_{n\,\{2\}}^0 = \underbrace{f[f[\dots f}_{n \text{ times}}[Y^0, q_{n-1}, X_{-(n-1)}], q_{n-2}, X_{-(n-2)}] \dots], q_1, X_{-1}], q_0, X] \ .$$

If we denote the reversed list of parameters $\langle q_{n-1}, q_{n-2}, \dots, q_1, q_0 \rangle$ by $W_{0,n-1}$, $Z = X_{-(n-1)}$, and the function computed by the array by $F'[n, X]$ then F' can be expressed in a functional way:

Computations:

$$\underline{y}_{t+1}^{i+1} = f[\underline{y}_t^i, q^{n-1-i}, \bar{x}_t^{i+1}]$$

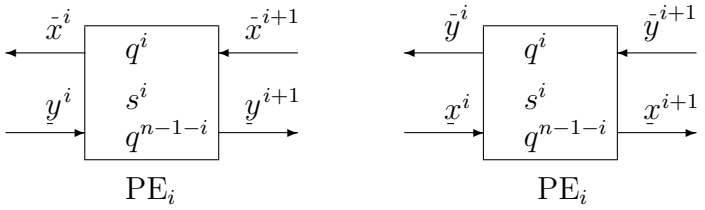
$$\bar{x}_{t+1}^{i} = \begin{cases} \bar{x}_t^{i+1}, \text{if } 0 \le i \le \frac{n}{2} - 2 \\ \underline{x}_t^{i+1}, \text{if } i = \frac{n}{2} - 1 \end{cases}$$

$$\bar{y}_{t+1}^{i} = \begin{cases} f[\bar{y}_t^{i+1}, q^i, \underline{x}_t^i], \text{if } 0 \le i \le \frac{n}{2} - 2 \\ f[\underline{y}_t^{i+1}, q^i, \underline{x}_t^i], \text{if } i = \frac{n}{2} - 1 \end{cases}$$

$$\underline{x}_{t+1}^{i+1} = \underline{x}_t^i$$

common computations:

$$\begin{array}{lllll} s_{t+1}^i & = & 1 - s_t^i & & \\ q_{t+1}^i & = & q_t^i & = q^i & = q_i \\ q_{t+1}^{n-1-i} & = & q_t^{n-1-i} & = q^{n-1-i} & = q_{n-1-i} \end{array}$$



PE_i PE_i

a) $s^i = 0$ **b)** $s^i = 1$

Figure 5.32: Computation of a PE in a bidirectional "folded" array. A different view.

$$F'_{W_{0,n-1}}[n,Z] = \vec{f}[w_{n-1}, Z_{n-1}, F'_{W_{0,n-2}}[n-1,Z]] \quad (5.104)$$

$$F'_{w_0}[1,Z] = \vec{f}[w_0, Z, Y^0] \ , \quad (5.105)$$

where $\vec{f}_w[\langle Z,Y\rangle]$ has property (5.16).

Thus we have shown that the "folded" array with constant internal state registers defined in this section computes the same list of results as a corresponding bidirectional array with sparse input and constant internal state registers.

Note that we had to rename (more exactly to shift) the input list in order to obtain the same result, which is obvious because in the case of the array on Fig. 5.30 we have chosen for the sake of convenience $x_{-(n-1)}$ to be the first input from right to left to PE_0. Thus the initial state of the array corresponds to that of the inefficient array with global input $T_{-1}X_{-(\frac{n}{2}-1)}\,\$$.

There is no speed-up in the computation of the results but the number of the PEs used for the computation of the same result was reduced to the half which increases the efficiency of the array. The PEs of the folded array are active at each time step.

5.6 Online Systolic Arrays

Online arrays are in fact a class of bidirectional arrays, but we will study them in a distinct section because of their importance. They are characterised by the fact that they begin to provide the first result after a constant number of time steps. Thus the efficiency of such arrays does not depend on the number of processing elements. This feature make them very useful for solving real time problems, where the response time is a critical factor.

Informally, we can see an *online systolic array* as a device consisting of one *head-processor* connected to a *tail-array*, which is an identical systolic array (Fig. 5.33).

Note that this is actually a simplified version of Fig. 5.1, namely in this case we are not interested in the number of PEs, because the computations do not depend on it. The only aspect related to the number of PEs that is important is to have a "sufficient" number of PEs to be able to perform the computations. Otherwise the problem of partitioning should be taken into account, which is not tackled in the thesis.

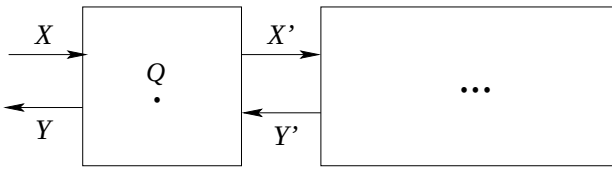


Figure 5.33: Informal view of an online systolic array

The array receives as input a list $X = \langle x_0, x_1, \ldots \rangle$ and outputs a list $Y = \langle y_0, y_1, \ldots \rangle$, while the list of the values of the internal state of the head-processor is $Q = \langle q_0, q_1, \ldots \rangle$. The communication with the tail-array is substantiated in the lists $X' = \langle x'_0, x'_1, \ldots \rangle$ and $Y' = \langle y'_0, y'_1, \ldots \rangle$. X and X', as well as Y and Y' are built upon the same scalar types.

At each discrete time step t (starting at 0), the array receives a value x_t and outputs a value y_t through the head-processor. Additionally the head processor outputs the value x'_t and receives the value y'_t (which are the input and the output, respectively, of the tail array), and updates its internal state q_t into q_{t+1}. The update is performed by the scalar function f (the transition function), and uses the currently received values x_t and y'_t, as well as the current value of q_t. The outputs y_t and x'_t are parts of the same current value q_t, obtained by the projection functions p_x and p_y. More exactly:

$$q_{t+1} = f[x_t, q_t, y'_t], \tag{5.106}$$

$$y_t = p_y[q_t], \quad x'_t = p_x[q_t].$$

In more practical terms, the internal state is composed by a certain number of internal variables. We further convene that some of these variables represent the input X, some represent the output Y, and some represent the output X' towards the tail-array. The transition function f will be represented as a parallel assignment for all variables except the ones corresponding to X. Note that the array is completely specified by the list of these variables, their association to X, Y, X', and the assignment describing f.

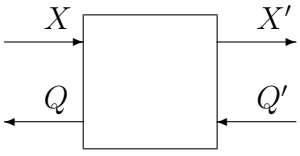
The initial configuration of the array consists in blanks (as values of the internal states of all the processors), and we convene that f must have the property $f[\$, \$, \$] = \$$. Therefore $q_0 = y_0 = x'_0 = \$$, thus only $T[Y]$ and $T[Q]$ contain "interesting" values. We will say that the array computes the function $F[X] = T[Y]$. Usually, the infinite lists representing the input and the output will have "interesting" values only for a finite number of elements at the beginning, and the rest will be blanks.

The synthesis problem consists in finding f when F is known. When F is given as a recursive functional program, we will show in the sequel that f can be obtained by syntactic transformations of this program.

5.6.1 Online "Pass-Through" Arrays without Internal State

We study in this section online arrays with:

- uniform initialization,
- no internal state,
- inputs "pass-through"



$$\begin{array}{rcl} x_t, q_t & \xrightarrow{f} & q_{t+1} \\ x_t & \longrightarrow & x'_{t+1} \end{array}$$

Figure 5.34: Computations of a PE

Figure 5.34 presents the computations that take place in a PE of such an array. We denote by q_t^n the output of the n^{th} PE at time step t, respectively with x_t^n the input of the n^{th} PE at time step t. Figure 5.35 presents the initial state of the array. The inputs *pass through* the array, that means $x_t^n = x_{t-1}^{n-1}$. The array is *uniformly initialised*, so let us denote with x^0 the initial input values $x_0', x_0'', \ldots$ of PE_j, $j \geq 1$, and with q^0 the initial value for the outputs $q_0, q_0', q_0'', \ldots$ of all PEs.

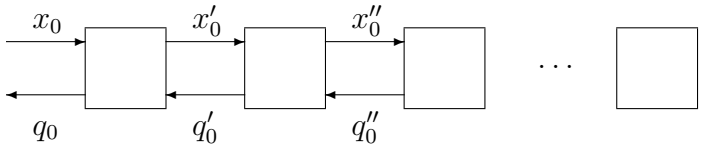


Figure 5.35: Initialization of the array

The output q of the first PE at time step $t+1$ depends on the input x and the output q' of the next PE at time step t. That is:

$$q_{t+1} = f[x_t, q_t']$$

Let us compute a few of the first elements of the result:

$$\begin{array}{lllllllll}
q_0 & = & q^0 & & & & & & \\
q_1 & = & f[x_0, q_0'] & = & f[x_0, q^0] & & & & \\
q_2 & = & f[x_1, q_1'] & = & f[x_1, f[x_0', q_0'']] & = & f[x_1, q^0] & & \\
q_3 & = & f[x_2, q_2'] & = & f[x_2, f[x_1', q_1'']] & = & f[x_2, f[x_0, f[x_0'', q_0''']]] & = & \\
 & = & f[x_2, f[x_0, q^0]] & = & f[x_2, q_1] & & & & \\
q_4 & = & f[x_3, q_3'] & = & \ldots & = & f[x_3, q_2] & & \\
\ldots & & & & & & & &
\end{array}$$

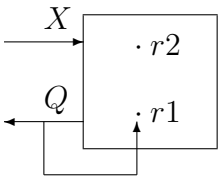
Note that we assumed that $f[x^0, q^0] = q^0$ holds, which is a natural requirement. In other words we suppose that the function f computed by a PE will change the initialization values only for the relevant input values (starting with x_0).

One can conclude that in general

$$q_k = f[x_{k-1}, q_{k-2}], k \geq 2 \tag{5.107}$$

Equation (5.107) can be easily verified by mathematical induction. This means actually that the computation performed by such an array can be computed by a single PE having two registers, such that the result computed two time steps before can be stored. Thus, such an array can be reduced to a single processor.

The functioning of such a PE is presented in Fig. 5.36



Computations:

$$\begin{aligned} x_t, r2_t &\xrightarrow{f} q_{t+1} \\ r1_t &\rightarrow r2_{t+1} \\ q_{t+1} &\rightarrow r1_{t+1} \end{aligned}$$

Initialization:

$$r1 = r2 = q^0$$

Figure 5.36: PE which performs the same computation as the array of Fig. 5.35

Also note that both, the systolic array, respectively the processor of Fig. 5.36 are computing two independent results on even/odd order elements of X:

$$\begin{aligned} q_{t+1} &= f[x_t, f[x_{t-2}, f[x_{t-4}, f[x_{t-6}, \ldots] \ldots] \\ q_{t+2} &= f[x_{t+1}, f[x_{t-1}, f[x_{t-3}, f[x_{t-5}, \ldots] \ldots] \end{aligned}$$

5.6.2 Arrays with Delayed Input "Pass-Through"

We study in this section a simplified version of online arrays, in which the output from the head processor towards the tail array consists in a copy of the main input, except for the first 2 elements: thus the tail array receives the second tail of the input. The synthesis method is based on two main properties, which are detected by equational reasoning based on the functional view of the array and of the computed functions [JS05]:

- The list function computed by the array has the property that the 4-th tail of the result can be expressed recursively using the same function applied to the 2-nd tail of the input (and some other head and tail components which are easy to synthesise).

- The scalar expressions (involving individual numeric variables and functions upon them) describing the transition function of the head processor generate the list expressions (involving also list variables and list

functions) describing the function realised by the array according to certain simple rewrite rules, which are also "reversible".

These two facts lead to the following synthesis method:

- Unfold the list expression of the target list function until it has the required property stated above. Unfolding consists in extracting repetitively the scalar expression of the head and the list expression of the tail, by using the functional definitions of the list functions and a few simple unfolding rules.
- Project the final list expression into the scalar space, by using the "reversed" rules mentioned above, and thus obtain the expression of the transition function.

The input X' of the tail-array is $T_k[X]$ for some fixed k, thus the computation of $F[X]$ will use $F[T_k[X]]$ computed by the tail-array. The data flow is illustrated in Fig. 5.37 for the case $k = 2$.

This behaviour can be realised by including into the internal state a "state variable" s with values from {0 = \$, 1, 2, ..., k+2}, and the following assignments for s and x':

$$s := \begin{cases} s, \text{ if } x = \$ \text{ or } s = k+2 \\ s+1, \text{ if } x \neq \$ \text{ and } s < k+2 \end{cases}$$

$$x' := \begin{cases} \$, \text{ if } s < k \\ x, \text{ if } s \geq k \end{cases}$$

When s is $k + 2$, then the first "interesting" value computed by the tail array becomes available.

Characterization of the Array

We will derive now the general recursive equation for the function F computed by such an array. Let us denote by G the list function which gives $T[Q]$ from X: $GX = TQ$. (In the sequel we will sometimes omit the brackets denoting function application, when this does not lead to ambiguities.) Furthermore let us consider the list extensions of the scalar functions characterising the array:

$$\vec{f}[u \overset{\cdot}{\smile} U, v \overset{\cdot}{\smile} V, w \overset{\cdot}{\smile} W] = f[u, v, w] \overset{\cdot}{\smile} \vec{f}[U, V, W],$$

$$P_x[u \overset{\cdot}{\smile} U] = p_x[u] \overset{\cdot}{\smile} P_x[U],$$

$$P_y[u \overset{\cdot}{\smile} U] = p_y[u] \overset{\cdot}{\smile} P_y[U].$$

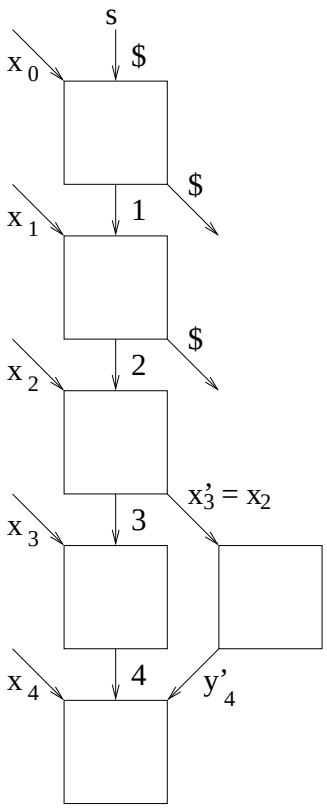


Figure 5.37: Data flow in an array with input pass-through delayed by 2.

Note that they all commute with T:

$$\vec{f}[TU, TV, TW] = T\vec{f}[U, V, W],$$

$$P_x[TU] = TP_x[U], \quad P_y[TU] = TP_y[U],$$

and also that (5.106) extends to the list equations:

$$TQ = \vec{f}[X, Q, Y'] = \vec{f}[X, Q, P_yQ'],$$

where Q' denotes the list of internal states of the first processor of the tail array.

In order to simplify the presentation, we will develop the expressions for the case $k = 2$ (but the generalisation is straightforward).

Clearly we need to express the function G, and then $F = P_yG$. We know that $GX = TQ$, whose first 4 values are: $q_i = f[x_{i-1}, q_{i-1}, \$],\ i = 1, 4$, thus they do not use any result from the tail array. The behaviour after this moment allows the derivation of a recursive expression for T_4G:

$$T_4GX = T_5Q = \vec{f}[T_4X,\ T_4Q,\ T_4P_yQ'].$$

On the right-hand side, $T_4Q = T_3GX$, because $TQ = GX$. Also: $T_4P_yQ' = P_yT_4Q'$ (commutativity of P_y with T), and $T_4Q' = GT_3X'$ (the tail–array computes the same function G), and $T_3X' = T_2X$ (input pass–through). Thus one obtains the characteristic equation for G:

$$T_4GX = \vec{f}[T_4X,\ T_3GX,\ P_yGT_2X].$$

Additionally one may use $F = P_yG$ in order to transform this into:

$$T_4FX = P_y\vec{f}[T_4X,\ T_3GX,\ FT_2X], \tag{5.108}$$

which shows that a function F that can be unfolded until the expression of T_4F contains FT_2 is a good candidate for implementation on an online systolic array.

Expressions vs. Functions

If E is a scalar expression with variables from the internal state, then the list of values of E depends on X. We will say that E realises the function F_E, and also that it realises the list $F_E[X]$. For instance, the expression s realises the list $F_s[X] = \langle 0, 1, \ldots, k+1, k+2, k+2, k+2, \ldots\rangle$, and also $F_x[X] = X$.

Let f be a scalar function in two variables and the list function $\vec{f}$ defined as $\vec{f}[u \overset{\cdot}{\smile} U, v \overset{\cdot}{\smile} V] = f[u, v] \overset{\cdot}{\smile} \vec{f}[U, V]$. Then the expression $f[E1, E2]$

realises the function $\vec{f}[F_{E1}, F_{E2}]$. This relation allows us to transform a scalar expression into its list function and also the other way around, by recursive projection of the list expressions into the scalar space.

Of particular interest are the head and tail functions H_i and T_i mentioned in the previous section. They can be realised by adding some suitable variables to the internal state.

The list having (almost) all elements equal to H_i is realised by a "static" variable h_i having the assignment:

$$h_i := \begin{cases} x, \text{ if } s = i \\ h_i, \text{ if } s \neq i \end{cases}.$$

Let us also consider the "transition" variables z_0, z_1, z_2, z_3 having the assignments:

$$z_0 = z_1,\ z_1 = z_2,\ z_2 = z_3,\ z_3 = x.$$

In the expression of T_4FX, the subexpression T_4X will be realised by the expression x, and each T_iX will be realised by the expression z_i (for $0 \leq i \leq 3$).

5.6.3 Synthesis Method for Arrays with Delayed Input "Pass-Through"

The considerations presented allow us to construct the systolic array in a systematic manner by transformations and projections of the target function F.

First one unfolds the recursive expression of F until one obtains an expression $\mathcal{E}$ for T_4FX which contains FT_2X. (The unfolding principles are common knowledge in the program transformation theory and are illustrated in the next section.) By unfolding one also obtains the scalar expressions for the first 4 values of FX, which can be directly used in the expression of the transition function f.

Second one adds the necessary static and transition variables to the internal state, according to the occurrences of H_i and T_i in the expression $\mathcal{E}$.

Third one projects the expression $\mathcal{E}$ into the scalar space, in order to obtain the assignment for y:

- The subexpression FT_2X is projected into y', because it corresponds to the output of the tail-array.
- T_4X is projected to x, because it is the current input of the array.
- H_i and T_i are projected to the corresponding static and transit variables.

- Recursively, each subexpression of the form $\vec{f}[\mathcal{E}_1, \mathcal{E}_2]$ is projected into the corresponding $f[E_1, E_2]$.

The expressions corresponding to the first 4 values of the output contain only scalar functions and subexpressions of the form $H_i[X]$, which are projected into the variables h_i, with the exception that if $H_i[X]$ occurs in the expression of $H_i[F[X]]$, then it is projected into x.

All these transformations are readily specified as rewrite rules and are used in order to generate the online array in a completely automatic manner, as illustrated in the examples below.

Of course the automatic generation succeeds only if the unfolded version of F has the appropriate shape and contains only functions which are also "projectable", but this is absolutely natural, since we cannot expect that every recursive function is realizable by an online array.

5.6.4 Functional-Based Synthesis of Systolic Online Multipliers - Two Case Studies

Polynomial Multiplication

We demonstrate now the method by synthesising the online multiplier of univariate polynomials.

An univariate polynomial (like e. g. $a_0 + a_1x + a_2x^2 + \ldots$) is represented by the list of its coefficients (lowest degree first): $A = \langle a_i \rangle_{i=0}^{\infty}$, with an infinite number of redundant zeroes at the end. The type of the coefficients is not important, we just assume it is some scalar type having ring properties.

We also assume as known the scalar operations " $\dot{+}$ " and " $\dot{*}$ " in the ring of the coefficients, as well as the following functional definitions of the operations on polynomials:

- addition of a scalar with a polynomial:

$$a \dot{+} (b \dot{\smile} B) = (a \dot{+} b) \dot{\smile} B$$

- addition of polynomials:

$$(a \dot{\smile} A) + (b \dot{\smile} B) = (a \dot{+} b) \dot{\smile} (A + B)$$

- multiplication of a scalar with a polynomial:

$$a \dot{*} (b \dot{\smile} B) = (a \dot{*} b) \dot{\smile} (a \dot{*} B)$$

- multiplication of polynomials:

$$(a \dot{\smile} A) * (b \dot{\smile} B) = (a \ddot{*} b) \dot{\smile} ((a \dot{*} B) + (b \dot{*} A) + (0 \dot{\smile} (A * B)))$$

Using these definitions and the unfolding rules presented previously, we unfold the expression "$A * B$", extracting repetitively the scalar expression representing the first element of the result, until the list expression representing the tail of the result contains $T_2[A] * T_2[B]$. (For brevity we denote $H_i[A]$ by a_i, $T_i[A]$ by A_i, and similarly for B.)

$$\begin{aligned}
& A * B = \\
& = (a_0 \dot{\smile} A_1) * (b_0 \dot{\smile} B_1) \\
& = \langle a_0 \ddot{*} b_0 \rangle \smile + \left\{ \begin{array}{l} a_0 \dot{*} B_1 \\ b_0 \dot{*} A_1 \\ 0 \dot{\smile} (A_1 * B_1) \end{array} \right. \\
& = \langle a_0 \ddot{*} b_0 \rangle \smile + \left\{ \begin{array}{l} a_0 \dot{*} (b_1 \dot{\smile} B_2) \\ b_0 \dot{*} (a_1 \dot{\smile} A_2) \\ 0 \dot{\smile} A_1 * B_1 \end{array} \right. \\
& = \langle a_0 \ddot{*} b_0,\ a_0 \ddot{*} b_1 \ddot{+} b_0 \ddot{*} a_1 \rangle \smile + \left\{ \begin{array}{l} a_0 \dot{*} B_2 \\ b_0 \dot{*} A_2 \\ A_1 * B_1 \end{array} \right. \\
& = \ldots \\
& = \langle \ a_0 \ddot{*} b_0, \\
& \qquad a_0 \ddot{*} b_1 \ddot{+} b_0 \ddot{*} a_1, \\
& \qquad a_2 \ddot{*} b_0 \ddot{+} a_1 \ddot{*} b_1 \ddot{+} a_0 \ddot{*} b_2), \\
& \qquad a_3 \ddot{*} b_0 \ddot{+} a_2 \ddot{*} b_1 \ddot{+} a_1 \ddot{*} b_2 \ddot{+} a_0 \ddot{*} b_3 \ \rangle \smile \\
& \smile ((a_0 \dot{*} B_4) + (b_0 \dot{*} A_4) + \\
& \qquad + (a_1 \dot{*} B_3) + (b_1 \dot{*} A_3) + (A_2 * B_2))
\end{aligned}$$

Thus we obtain an expression of the form required by (5.108):

$$T_4[A * B] = + \left\{ \begin{array}{l} H_0[A] \dot{*} T_4[B] \\ H_0[B] \dot{*} T_4[A] \\ H_1[A] \dot{*} T_3[B] \\ H_1[B] \dot{*} T_3[A] \\ T_2[A] * T_2[B] \end{array} \right.$$

as well as the first 4 values of the output.

Let us denote the input by xa and xb and the corresponding static and transition variables by ha_i, hb_i, za_i, zb_i. According to the rules presented in the previous section, the expression on the right–hand side is projected into:

$$\begin{array}{l}(ha_0 \stackrel{\cdot\cdot}{*} xb) \stackrel{\cdot\cdot}{+} (hb_0 \stackrel{\cdot\cdot}{*} xa) \stackrel{\cdot\cdot}{+} \\ \stackrel{\cdot\cdot}{+} (ha_1 \stackrel{\cdot\cdot}{*} zb_3) \stackrel{\cdot\cdot}{+} (hb_1 \stackrel{\cdot\cdot}{*} za_3) \stackrel{\cdot\cdot}{+} y'\end{array}$$

The first 4 elements are projected into:

$$\begin{array}{l}\langle\ xa \stackrel{\cdot\cdot}{*} xb, \\ \ hb_0 \stackrel{\cdot\cdot}{*} xa \stackrel{\cdot\cdot}{+} ha_0 \stackrel{\cdot\cdot}{*} xb, \\ \ ha_1 \stackrel{\cdot\cdot}{*} hb_1 \stackrel{\cdot\cdot}{+} hb_0 \stackrel{\cdot\cdot}{*} xa \stackrel{\cdot\cdot}{+} ha_0 \stackrel{\cdot\cdot}{*} xb, \\ \ hb_1 \stackrel{\cdot\cdot}{*} za_3 \stackrel{\cdot\cdot}{+} ha_1 \stackrel{\cdot\cdot}{*} zb_3 \stackrel{\cdot\cdot}{+} hb_0 \stackrel{\cdot\cdot}{*} xa \stackrel{\cdot\cdot}{+} ha_0 \stackrel{\cdot\cdot}{*} xb\ \rangle\end{array}$$

Thus each processor of the array has the input variables xa, xb, and the output variable y and communicates with the next processor through the variables xa', xb', and y'. The internal variables are s, ha_0, hb_0, ha_1, hb_1, za_3, zb_3, and y, from which y is the output variable. The transition function is described by the assignments presented in the previous section for the variables $s, ha_i, hb_i, za_3, zb_3, xa', xb'$, and by the following assignment for y:

$$\left\{\begin{array}{ll} \$ & \text{, if } s = \$ = xa \\ xa \stackrel{\cdot\cdot}{*} xb & \text{, if } s = \$ \neq xa \\ hb_0 \stackrel{\cdot\cdot}{*} xa \stackrel{\cdot\cdot}{+} ha_0 \stackrel{\cdot\cdot}{*} xb & \text{, if } s = 1 \\ ha_1 \stackrel{\cdot\cdot}{*} hb_1 \stackrel{\cdot\cdot}{+} & \\ \quad \stackrel{\cdot\cdot}{+} hb_0 \stackrel{\cdot\cdot}{*} xa \stackrel{\cdot\cdot}{+} ha_0 \stackrel{\cdot\cdot}{*} xb & \text{, if } s = 2 \\ hb_1 \stackrel{\cdot\cdot}{*} za_3 \stackrel{\cdot\cdot}{+} ha_1 \stackrel{\cdot\cdot}{*} zb_3 \stackrel{\cdot\cdot}{+} & \\ \quad \stackrel{\cdot\cdot}{+} hb_0 \stackrel{\cdot\cdot}{*} xa \stackrel{\cdot\cdot}{+} ha_0 \stackrel{\cdot\cdot}{*} xb & \text{, if } s = 3 \\ hb_1 \stackrel{\cdot\cdot}{*} za_3 \stackrel{\cdot\cdot}{+} ha_1 \stackrel{\cdot\cdot}{*} zb_3 \stackrel{\cdot\cdot}{+} & \\ \quad \stackrel{\cdot\cdot}{+} hb_0 \stackrel{\cdot\cdot}{*} xa \stackrel{\cdot\cdot}{+} ha_0 \stackrel{\cdot\cdot}{*} xb \stackrel{\cdot\cdot}{+} y' & \text{, if } s = 4 \end{array}\right.$$

Integer Multiplication

The multiplication of two numbers is similar to the polynomial multiplication problem, the only significant difference is the carry propagation.

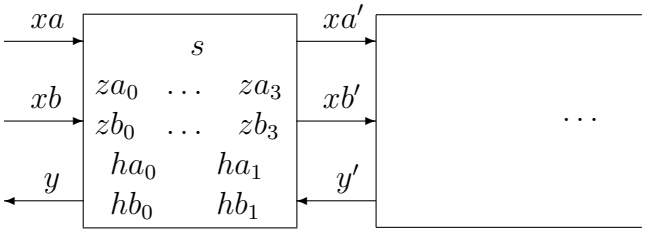


Figure 5.38: Online systolic array for polynomial multiplication

Let β be the radix for integer representation ($\beta > 1$). We consider that integer numbers are represented by the list of their digits, least significant digits first:

$$\langle a_0, a_1, a_2, \ldots \rangle \text{ represents } a_0 + a_1\beta + a_2\beta^2 + \ldots$$

We will use a scalar type for *digits* (positive integers less than β) and the corresponding list type for arbitrary precision integers, but also a scalar type for *large integers* – but not arbitrary large. The later is determined by the fixed number of operations performed by the transition function, and for sufficiently large β will be equivalent to three digits. Unless otherwise specified, in the sequel we will use "scalar" for large integer.

On large integers we define the operation "d" of *carry decomposition*

$$d[a] \;=\; \langle a \mod \beta, \lfloor \frac{a}{\beta} \rfloor \rangle,$$

which generates a pair consisting in the least significant digit of a and the "carry". Note that the carry is not necessarily a digit.

We consider as known the scalar operations "$\dot{\dot{+}}$" and "$\dot{\dot{*}}$" acting on large integers (thus also on digits).

The operation "$\dot{+}$" of addition between a scalar and a list is defined as:

$$a \mathbin{\dot{+}} (b \mathbin{\dot{\smile}} B) = \text{Let}\{\langle y, r\rangle = d[a \mathbin{\dot{\dot{+}}} b];\; y \mathbin{\dot{\smile}} (r \mathbin{\dot{+}} B)\}$$

Here the construct "Let" contains a local assignment and a final expression which is the result of the construct. This construct is necessary in order to avoid repeated computations of the same expression, in particular when the result occurs both in the head and in the tail of the resulting list (as above).

The projection of this construct (when translating from list expressions into scalar expressions) is performed by inserting (if necessary) local variables into the internal state and by adding the respective assignment to the transition function, at the time step corresponding to the stage of decomposition of the main function (see the example below).

In order to treat carry propagation in a functional way, we use the operation "$\dot{\rightarrow}$" (prepend with carry addition):

$$a \dot{\rightarrow} A \;=\; Let\{\langle y, r\rangle = d[a];\; y \dot{\smile} (r \dot{+} A)\}$$

Obviously

$$a \dot{\smile} A = a \dot{\rightarrow} A$$

when a is a digit, and also:

$$a \dot{+} (b \dot{\smile} B) = (a \ddot{+} b) \dot{\rightarrow} B$$
$$a \dot{+} (b \dot{\rightarrow} B) = (a \ddot{+} b) \dot{\rightarrow} B$$

The operations with lists are defined by:

$$(a \dot{\smile} A) + (b \dot{\smile} B) = (a \ddot{+} b) \dot{\rightarrow} (A + B)$$
$$a \dot{*} (b \dot{\smile} B) = (a \ddot{*} b) \dot{\rightarrow} (a \dot{*} B)$$
$$(a \dot{\smile} A) * (b \dot{\smile} B) =$$
$$= (a \ddot{*} b) \dot{\rightarrow} ((a \dot{*} B) + (b \dot{*} A) + (0 \dot{\smile} A * B))$$

and note that:

$$(a \dot{\rightarrow} A) + (b \dot{\rightarrow} B) = (a \ddot{+} b) \dot{\rightarrow} (A + B).$$

Similarly to what happened by polynomial multiplication, we unfold the expression of the multiplication:

$$\begin{aligned}
&A * B = \\
&= (a_0 \dot{\smile} A_1) * (b_0 \dot{\smile} B_1) \\
&= \; (a_0 \ddot{*} b_0) \dot{\rightarrow} + \begin{cases} a_0 \dot{*} B_1 \\ b_0 \dot{*} A_1 \\ 0 \dot{\smile} (A_1 * B_1) \end{cases} \\
&= \mathrm{Let}\{\langle y, r\rangle = d[a_0 \ddot{*} b_0]; \\
&\qquad\qquad y \dot{\smile} (r \dot{+} + \begin{cases} a_0 \dot{*} B_1 \\ b_0 \dot{*} A_1 \\ 0 \dot{\smile} (A_1 * B_1) \end{cases})\},
\end{aligned}$$

which gives the expression for the first value of the output as y, while the tail of the output is further unfolded and transformed using the properties of "$\dot{\rightarrow}$":

$$
\begin{aligned}
T_1[A*B] &= r \mathbin{\dot{+}} + \begin{cases} a_0 \mathbin{\dot{*}} (b_1 \mathbin{\dot{\smile}} B_2) \\ b_0 \mathbin{\dot{*}} (a_1 \mathbin{\dot{\smile}} A_2) \\ 0 \mathbin{\dot{\smile}} A_1 * B_1 \end{cases} \\
&= r \mathbin{\dot{+}} + \begin{cases} (a_0 \mathbin{\ddot{*}} b_1) \mathbin{\dot{\rightarrow}} (a_0 \mathbin{\dot{*}} B_2) \\ (b_0 \mathbin{\ddot{*}} a_1) \mathbin{\dot{\rightarrow}} (b_0 \mathbin{\dot{*}} A_2) \\ 0 \mathbin{\dot{\rightarrow}} A_1 * B_1 \end{cases} \\
&= r \mathbin{\dot{+}} ((a_0 \mathbin{\ddot{*}} b_1 \mathbin{\ddot{+}} b_0 \mathbin{\ddot{*}} a_1 \mathbin{\ddot{+}} 0) \mathbin{\dot{\rightarrow}} + \begin{cases} a_0 \mathbin{\dot{*}} B_2 \\ b_0 \mathbin{\dot{*}} A_2 \\ A_1 * B_1 \end{cases}) \\
&= (r \mathbin{\ddot{+}} a_0 \mathbin{\ddot{*}} b_1 \mathbin{\ddot{+}} b_0 \mathbin{\ddot{*}} a_1) \mathbin{\dot{\rightarrow}} + \begin{cases} a_0 \mathbin{\dot{*}} B_2 \\ b_0 \mathbin{\dot{*}} A_2 \\ A_1 * B_1 \end{cases} \\
&= \mathrm{Let}\{\langle y, r\rangle = d[r \mathbin{\ddot{+}} a_0 \mathbin{\ddot{*}} b_1 \mathbin{\ddot{+}} b_0 \mathbin{\ddot{*}} a_1]; \\
&\qquad y \mathbin{\dot{\smile}} (r \mathbin{\dot{+}} + \begin{cases} a_0 \mathbin{\dot{*}} B_2 \\ b_0 \mathbin{\dot{*}} A_2 \\ A_1 * B_1 \end{cases})\},
\end{aligned}
$$

which gives the expression of the second value of the input as y and the second tail of the output. The later is further transformed in a similar manner into:

$$
\begin{aligned}
&T_2[A*B] = \\
&\mathrm{Let}\{\langle y, r\rangle = d[r \mathbin{\ddot{+}} a_2 \mathbin{\ddot{*}} b_0 \mathbin{\ddot{+}} a_1 \mathbin{\ddot{*}} b_1 \mathbin{\ddot{+}} a_0 \mathbin{\ddot{*}} b_2]; \\
&\qquad y \mathbin{\dot{\smile}} (r \mathbin{\dot{+}} + \begin{cases} a_0 \mathbin{\dot{*}} B_3 \\ b_0 \mathbin{\dot{*}} A_3 \\ a_1 \mathbin{\dot{*}} B_2 \\ b_1 \mathbin{\dot{*}} A_2 \\ 0 \mathbin{\dot{\smile}} (A_2 * B_2) \end{cases})\},
\end{aligned}
$$

and finally:

$$
\begin{array}{l}
T_3[A * B] = \\
\mathrm{Let}\{\langle y, r\rangle = \\
\qquad d[r \overset{\cdot\cdot}{+} a_3 \overset{\cdot\cdot}{*} b_0 \overset{\cdot\cdot}{+} a_2 \overset{\cdot\cdot}{*} b_1 \overset{\cdot\cdot}{+} a_1 \overset{\cdot\cdot}{*} b_2 \overset{\cdot\cdot}{+} a_0 \overset{\cdot\cdot}{*} b_3]; \\
\qquad y \overset{\cdot}{\smile} (r \overset{\cdot}{+} + \begin{cases} a_0 \overset{\cdot}{*} B_4 \\ b_0 \overset{\cdot}{*} A_4 \\ a_1 \overset{\cdot}{*} B_3 \\ b_1 \overset{\cdot}{*} A_3 \\ A_2 * B_2 \end{cases})\}
\end{array}
$$

gives the expression of the fourth value of the output and the list expression for $T_4[A * B]$ becomes of the same form as in (5.108).

We need to use the same variables as in the case of polynomial multiplication, but also the additional variable r for the carry. The assignments are again all the same, with the exception of the assignment for y, which is replaced by a new assignment for $\langle y, r\rangle$:

$$
\begin{cases}
\langle \$, \$\rangle & , \text{if } s = \$ = xa \\
d[xa \overset{\cdot\cdot}{*} xb] & , \text{if } s = \$ \neq xa \\
d[r \overset{\cdot\cdot}{+} hb_0 \overset{\cdot\cdot}{*} xa \overset{\cdot\cdot}{+} ha_0 \overset{\cdot\cdot}{*} xb] & , \text{if } s = 1 \\
d[r \overset{\cdot\cdot}{+} ha_1 \overset{\cdot\cdot}{*} hb_1 \overset{\cdot\cdot}{+} & \\
\quad \overset{\cdot\cdot}{+} hb_0 \overset{\cdot\cdot}{*} xa \overset{\cdot\cdot}{+} ha_0 \overset{\cdot\cdot}{*} xb] & , \text{if } s = 2 \\
d[r \overset{\cdot\cdot}{+} hb_1 \overset{\cdot\cdot}{*} za_3 \overset{\cdot\cdot}{+} ha_1 \overset{\cdot\cdot}{*} zb_3 \overset{\cdot\cdot}{+} & \\
\quad \overset{\cdot\cdot}{+} hb_0 \overset{\cdot\cdot}{*} xa \overset{\cdot\cdot}{+} ha_0 \overset{\cdot\cdot}{*} xb] & , \text{if } s = 3 \\
d[r \overset{\cdot\cdot}{+} hb_1 \overset{\cdot\cdot}{*} za_3 \overset{\cdot\cdot}{+} ha_1 \overset{\cdot\cdot}{*} zb_3 \overset{\cdot\cdot}{+} & \\
\quad \overset{\cdot\cdot}{+} hb_0 \overset{\cdot\cdot}{*} xa \overset{\cdot\cdot}{+} ha_0 \overset{\cdot\cdot}{*} xb \overset{\cdot\cdot}{+} y'] & , \text{if } s = 4
\end{cases}
$$

The last expression (having 6 terms of 2 digits) indicates that 3 digits suffice as the size of a large integer, if $\beta > 6$. Thus we have also determined this scalar type.

Chapter 6

Conclusions

Achievements

...

Future Work

...

Bibliography

[AFM03] Srinivas Aluru, Natsuhiko Futamura, and Kishan Mehrotra. Parallel biological sequence comparison using prefix computations. *J. Parallel Distrib. Comput.*, 63(3):264–272, 2003.

[BDD90] J. Bu, E. F. Deprettere, and P. Dewilde. A design methodology for fixed-size systolic arrays. In S. Y. Kung and E. E. Swartzlander, editors, *Application Specific Array Processors*, pages pages 591–602. IEEE Computer Society, 1990.

[BK85] R. P. Brent and H. T. Kung. A systolic algorithm for integer gcd computation. In *7th Symp. on Computer Arithmetic*, pages 118–125. IEEE Computer Society Press, 1985.

[BM79] Robert S. Boyer and J. Strother Moore. *A computational logic*. Academic Press, New York, 1979.

[Che83] Marina Chien-mei Chen. *Space-Time Algorithms: Semantics and Methodology*. PhD thesis, California Institute of Technology, Pasadena, California, 1983.

[CK98] Y.-K. Chen and S. Y. Kung. A systolic design methodology with application to full-search block-matching architectures. *Journal of VLSI Signal Processing Systems*, 19(1):51–77, June 1998.

[Dar91] Alain Darte. Regular partitioning for synthesizing fixed-size systolic arrays. *The VLSI Jounal*, 12(3):293–304, Dec 1991.

[DD90] A. Darte and J.M. Delosme. Partitioning for array processors. Technical Report 90-23, LIP, ENS-Lyon, 46 allee d'Italie, 69364 Lyon cedex 07, France, October 1990.

[DI86] Jean-Marc Delosme and Ilse C. F. Ipsen. Systolic array synthesis: computability and time cones. In *Parallel algorithms & architectures (Luminy, 1986)*, pages 295–312, Amsterdam; New York, 1986. North-Holland.

[DLT89] J. Derrick, G. Lajos, and J. V. Tucker. Specification and verification of synchronous concurrent algorithms using the nurpl proof development system. Technical report, Centre for Theoretical Computer Science, The University of Leeds, 1989.

[DQ94] Catherine Dezan and Patrice Quinton. Verification of regular architectures using alpha: a case study. Internal publication 823, IRISA, Campus de Beaulieu, Rennes, France, May 1994.

[DVQS91] C. Dezan, H. Le Verge, P. Quinton, and Y. Saouter. The alpha du centaur environment. In P. Quinton and Y. Roberts, editors, *International Workshop Algorythms and Parallel VLSI Architectures*, volume II, pages 325–334, Bonas, France, June 1991. North-Holland.

[Eme90] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of theoretical Computer Science (vol. B): Formal models ans Semantics*, pages 995–1072. MIT Press, Cambridge, MA, 1990.

[Eve87] Hans Eveking. Verification, synthesis and correctness-preserving transformations cooperative approaches to correct hardware design. In D. Borrionne, editor, *From HDL Descriptions to Guaranted Correct Circuit Designs (IFIP)*. Elsevier Science Publishers BV (North-Holland), 1987.

[Fea92] P. Feautrier. Some efficient solutions to the affine scheduling problem, part I : One-dimensional time. *Int. J. of Parallel Programming*, 21(5):313–348, October 1992.

[FM84] J.A.B. Fortes and D.I. Moldovan. Data broadcasting in linearly scheduled array processors. In *11th Annual Symp. on Computer Architecture*, pages 224–231, 1984.

[FW87] J. A. B. Fortes and B. W. Wah. Systolic arrays: From concepts to implementation. *IEEE Computer*, 20(7):12–17, July 1987.

[GD92] Pascal Gribomont and Vincent Van Dongen. Generic systolic arrays: A methodology for systolic design. *Lecture Notes in Computer Science*, 668:746–761, 1992.

[Gor87] M. Gordon. Hol: a proof generating system for higher order logic. In G.M. Britwistle and P.A. Subrahmaynyam, editors, *VLSI Specification, Verification and Synthesis, Proceedings of*

the Workshop of Hardware Verification, Calgary, North-Holland, Amsterdam, January 1987.

[GQMS88] P. Gachet, P. Quinton, C. Mauras, and Y. Saouter. Alpha du centaur: A prototype environment for the design of parallel regular algorithms. Technical Report 953, IRISA, August 1988.

[Gri88] E. Pascal Gribomont. Proving systolic arrays. In Max Dauchet and Maurice Nivat, editors, *CAAP '88, 13th Colloquium on Trees in Algebra and Programming*, volume 299 of *Lecture Notes in Computer Science*, pages 185–199, Nancy, France, March 21-24 1988. Springer.

[Gup91] Aarti Gupta. Formal hardware verification methods: A survey. Internal report CMU-CS-91-193, Carnegie Mellon University, Pittsburgh, PA, October 1991.

[HH95] Y. T. Hwang and Y. H. Hu. A unified partitioning and scheduling scheme for mapping multi-stage regular iterative algorithms onto processor arrays. *J. of VLSI Signal Processing*, 11(1&2):133–150, 1995.

[HL87] C.-H. Huang and C. Lengauer. The derivation of systolic implementations of programs. *Acta Informatica*, 24(6), Nov 1987.

[Hoa85] C. S. Hoare. *Communicating Sequential Processes*. Prentice Hall International, 1985.

[Jeb93] Tudor Jebelean. A generalization of the binary GCD algorithm. In *ISSAC'93*, pages 111–116. ACM Press, 1993.

[Jeb94a] Tudor Jebelean. Designing systolic arrays for integer gcd computation. In *ASAP 94*, IEEE Computer Society Press, pages 295–301, 1994.

[Jeb94b] Tudor Jebelean. Systolic multiprecision arithmetic, PhD thesis. Technical Report 94-37, RISC-Linz, April 1994.

[JS05] Tudor Jebelean and Laura Szakács. Functional-Based Synthesis of Systolic Online Multipliers. In D. Zaharie, D. Petcu, V. Negru, T. Jebelean, G. Ciobanu, A. Cicortas, A. Abraham, and M. Paprzycki, editors, *SYNASC-05 (International Symposium on Symbolic and Numeric Scientific Computing)*, pages 267–275. IEEE Computer Society, 2005.

[KG99] Christoph Kern and Mark R. Greenstreet. Formal verification in hardware design: a survey. *ACM Transactions on Design Automation of Electronic Systems*, 4(2):123–193, 1999.

[KL78] H. T. Kung and C. E. Leiserson. Systolic Arrays (for VLSI). In I.S. Duff and G. W. Stewart, editors, *SIAM Sparse Matrix Proceedings*, pages 256–282, 1978.

[KL80] H. T. Kung and C. E. Leiserson. Algorithms for VLSI Processor arrays. In N.Ranganathan, editor, *VLSI Algorithms and Architectures -fundamentals*, pages 43–64. 1980.

[KMW76] R. M. Karp, R. E. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *Journal of the ACM*, 14(3):563–590, July 1976.

[KRS94] Ladan Kazerouni, Basant Rajan, and R. K. Shyamasundar. Derivation of systolic programs. In *International Conference on Parallel Processing (3)*, pages 69–73, 1994.

[KRS95] Ladan Kazerouni, Basant Rajan, and R. K. Shyamasundar. Mapping linear recurrences onto systolic arrays. In *IPPS: 10th International Parallel Processing Symposium*. IEEE Computer Society Press, 1995.

[KRS96] Ladan Kazerouni, Basant Rajan, and R. K. Shyamasundar. Mapping linear recurrence equations onto systolic architectures. *International Journal of High Speed Computing (IJHSC)*, 8(3):229–270, 1996.

[Kun82] H. T. Kung. Why systolic architectures? *IEEE Computer*, pages 37–46, 1982.

[Kun87] Sun-Yuan Kung. *VLSI Array Processors*. NJ: Prentice Hall, Englewood Cliffs, 1987.

[LB89] Nam Ling and Magdy A. Bayoumi. Sta: A tool for systolic array reasoning. In *Proceedings of the 1989 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 461–464, Portland, Oregon, USA, May 1989.

[LB99] Nam Ling and Magdy A. Bayoumi. *Specification and Verification of Systolic Arrays*. World Scientific Publishing Company Pte. Ltd., Singapore, 1999.

[LQR99] Dominique Lavenier, Patrice Quinton, and Sanjay Rajopadhy. Advanced systolic design. In Marcel Dekker, editor, *Digital Signal Processing for Multimedia Systems*, Signal Processing, chapter 23, pages 657–692. 1999.

[LW85] G.-J. Li and B. W. Wah. The design of optimal systolic arrays. *IEEE Trans. on Computers*, C-34(1):66–77, Jan. 1985.

[Mar96] Tiziana Margaria. Verification of systolic arrays in M2l(Str). Technical Report MIP-9613, Fakultät für Mathematik und Informatik, Universität Passau, 1996.

[Mel88] Thomas F. Melham. Abstraction mechanisms for hardware verification. In Graham Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 267–291. Kluwer Academic Publishers, 1988.

[MF86] Dan I. Moldovan and J. A. B. Fortes. Partitioning and mapping algorithms into fixed size systolic arrays. *IEEE Trans. Computers*, 35(1):1–12, 1986.

[Mil91] G. J. Milne. The formal description and verification of hardware timing. *IEEE Transaction on Computers*, 40(7), 1991.

[Mol83] Dan I. Moldovan. On the design of algorithms for VLSI systolic array. *Proc. IEEE*, 71(1):113–120, 1983.

[MQRS90] Christophe Mauras, Patrice Quinton, Sanjay Rajopadhye, and Yannick Saouter. Scheduling affine parameterized recurrences by means of variable dependent timing functions. Technical Report 520, IRISA, Campus de Beaulieu 35042 Rennes Cédex, Jan 1990.

[ND88] H. W. Nelis and E. F. Deprettere. Automatic design and partitioning of systolic/wavefront arrays for VLSI. *Circuits systems signal process*, 7(2), 1988.

[NW70] S. Needleman and C. Wunsch. A general method applicable to the search for similarities in the amino acides sequences of two protein. *J. of Molecular Biology*, 48:443–453, July 1970.

[OFW91] M. T. O'Keefe, J. A. B. Fortes, and B. W. Wah. On the relationship between systolic array design methodologies. *IEEE Trans. on Computers*, 41(12):1589–1593, 1991.

[Pet93] Nicolay Petcov. *Systolic parallel processing*. North-Holland, 1993.

[PS89] S. Purushothaman and P. A. Subrahmanyam. Mechanical certification of systolic algorithms. *Journal of Automated Reasoning*, 5(1):67–91, 1989.

[QD89] Patrice Quinton and Vincent Van Dongen. The mapping of linear recurrence equations on regular arrays. *Journal of VLSI Signal Processing*, 1(2):95–113, Oct 1989.

[QR90] P. Quinton and Y. Robert. *Systolic Algorithms and Architectures*. Prentice-Hall, 1990.

[Qui84] P. Quinton. Automatic synthesis of systolic arrays from uniform recurrence equations. In *11th Ann. Int. Symp. on Computer Architecture*, pages 208–214. IEEE Computer Society Press, 1984.

[Qui87] P. Quinton. The systematic design of systolic arrays. In Francoise Fogelman Soulie, Yves Robert, and Maurice Tchuente, editors, *Automata Networks in Computer Science*, chapter 9, pages 229–260. Manchester University Press, 1987.

[RJ06] Laura Ruff and Tudor Jebelean. Functional–based synthesis of a systolic array for GCD computation. In *Proceedings of the* 18^{th} *International Symposium on Implementation and Application of Functional Languages IFL'06*, August 2006.

[SC03] Laura Szakács and Ioana Chiorean. Automatic Derivation of a Systolic Algorithm for Sequences Comparison. In D. Petcu, V. Negru, D. Zaharie, and T. Jebelean, editors, *Proceedings of SYNASC 2003, 5th International Workshop on Symbolic and Numeric Algorithms for Scientific Computing Timisoara*, pages 277–288, Timisoara, Romania, 1-4 October 2003. Mirton Publisher.

[Sch86] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley and Sons, New York, 1986.

[Son94] Siang W. Song. *Systolic algorithms: concepts, synthesis, and evolution*. Temuco, CIMPA School of Parallel Computing, Chile, 1994.

[SQ93] Y. Saouter and P. Quinton. Computability of recurrence equations. *Theoretical Computer Science*, 114, 1993.

[Ste67] J. Stein. Computational problems associated with racah algebra. *J. Comp. Phys.*, 1:397–405, 1967.

[Sza02a] Laura Szakács. Automatic Design of Systolic Arrays: A Short Survey. Technical Report 02-27, RISC Report Series, University of Linz, Austria, December 2002.

[Sza02b] Laura Szakács. Different Approaches to Automatic Systolic Array Design. *Analele Universitatii din Timisoara, Seria Matematica - Informatica*, XL:257–280, 2002. special issue on Computer Science - Proceedings of SYNASC'02.

[VMQ91] H. Le Verge, C. Mauras, and P. Quinton. The alpha language and its use for the design of systolic arrays. *Journal of VLSI Signal Processing*, (3):173–182, 1991.

[Wil93] D. Wilde. A library for doing polyhedral operations. Technical Report 785, IRISA, Rennes, France, 1993.

[WS94] D. K. Wilde and O. Sie. Regular array synthesis using alpha. Technical report, 1994.