Chapter 12
High-Level Synthesis of Loops Using the Polyhedral Model

The MMAlpha Software

Steven Derrien, Sanjay Rajopadhye, Patrice Quinton, and Tanguy Risset

Abstract High-level synthesis (HLS) of loops allows efficient handling of intensive computations of an application, e.g. in signal processing. Unrolling loops, the classical technique used in most HLS tools, cannot produce regular parallel architectures which are often needed. In this Chapter, we present, through the example of the MMAlpha testbed, basic techniques which are at the heart of loop analysis and parallelization. We present here the point of view of the *polyhedral model* of loops, where iterative calculations are represented as recurrence equations on integral polyhedra. Illustrated from an example of string alignment, we describe the various transformations allowing HLS and we explain how these transformations can be merged in a synthesis flow.

Keywords: Polyhedral model, Recurrence equations, Regular parallel arrays, Loop transformations, Space–time mapping, Partitioning.

12.1 Introduction

One of the main problems that High Level Synthesis (HLS) tools have not solved yet is the efficient handling of nested loops. Highly computational programs occurring for example in signal processing and multimedia applications make extensive use of deeply nested loops. The vast majority of HLS tools either provide loop unrolling to take advantage of parallelism, or treat loops as sequential when unrolling is not possible. Because of the increasing complexity of embedded code, complete unrolling of loops is often impossible. Partial unrolling coupled with software pipelining techniques has been successfully used, in the Pico tool [29] for instance, but a lot of other loop transformations, such as loop tiling, loop fusion or loop interchange, can be used to optimize the hardware implementation of nested loops. A tool able to propose such loop transformations in the source code before performing HLS should necessarily have an internal representation in which the loop nest structure

P. Coussy and A. Morawiec (eds.) *High-Level Synthesis*.

is kept. This is a serious problem and this is why, for instance, source level loop transformations are still not available is commercial compilers, whereas the loop transformation theory is quite mature.

The work presented in this chapter proposes to perform HLS from the source language ALPHA. The ALPHA language is based on the so-called *polyhedral model* and is dedicated to the manipulation of *recurrence equations* rather than loops. The MMAlpha programming environment allows a user to transform ALPHA programs in order to refine the ALPHA initial description until it can be translated down to VHDL. The target architecture of MMAlpha is currently limited to regular parallel architectures described in a register transfer level (RTL) formalism. This paradigm, as opposed to the control+datapath formalism, is useful for describing highly pipelined architectures where computations of several successive samples are overlapped.

This chapter gives an overview of the possibilities of the MMAlpha design environment focusing on its use for HLS. The concepts presented in this chapter are not limited to the context were a specification is described using an applicative language such as ALPHA: they can also be used in a compiler environment as it has been done for example in the WraPit project [3].

The chapter is organized as follows. In Sect. 12.2, we present an overview of this system by describing the ALPHA language, its relationship with loop nests, and the design-flow of the MMAlpha tool. Section 12.3 is devoted to the front-end which transforms an ALPHA software specification into a virtual parallel architecture. Section 12.4 shows how synthesizable VHDL code can be generated. All these first sections are illustrated on a simple example of string alignment, so that the main concepts are apparent. In Sect. 12.5, we explain how the virtual architecture can be further transformed in order to be adapted to resource constraints. Implementations of the string alignment application are shown and discussed in Sect. 12.6. Section 12.7 is a short review of other works in the field of hardware generation for loop nests. Finally, Sect. 12.8 concludes the chapter.

12.2 An Overview of the MMAlpha Project

Throughout this chapter, we shall consider the running example of a string matching algorithm for genetic sequence comparison, as shown in Fig. 12.1. This algorithm is expressed using the single-assignment language ALPHA. Such a program is called a `system`. Its name is `sequence`, and it makes use of integral parameters `X` and `Y`. These parameters are constrained (line 1) to satisfy the linear inequalities $3 \leq X$ and $X \leq Y-1$. This system has two inputs: a sequence `QS` (for *Query Sequence*) of size `X` and a sequence `DB` (for *Data Base* sequence) of size `Y`. It returns a sequence `res` of integers. The calculation described by this system is expressed by *equations* defining local variables `M` and `MatchQ` as well as result `res`. Each ALPHA variable is defined on the set of integral points of a convex polyhedron called its *domain*. For example, `M` is defined on the set $\{i, j | 0 \leq i \leq X \wedge 0 \leq j \leq Y\}$. The definition of `M`

```
system sequence :{X,Y | 3<=X<=Y-1}
                        (QS : {i | 1<=i<=X} of integer;
                         DB : {j | 1<=j<=Y} of integer)
          returns  (res : {j | 1<=j<=Y} of integer);
var
  M : {i,j | 0<=i<=X; 0<=j<=Y} of integer;
  MatchQ : {i,j | 1<=i<=X; 1<=j<=Y} of integer;
let
  M[i,j] =
      case
        {| i=0} | {| 1<=i; j=0} : 0;
        {| 1<=i; 1<=j} : Max4(0, M[i,j-1] - 8,
                  M[i-1,j] - 8, M[i-1,j-1] + MatchQ[i,j]);
      esac;
  MatchQ[i,j] = if (QS[i] = DB[j]) then 15 else -12;
  res[j] = M[X,j];
tel;
```

Fig. 12.1 ALPHA program for the string alignment algorithm

is given by a case statement, each branch of which covers a subset of its domain. If $i = 0$ or if $j = 0$, then its value is 0. Otherwise, it is the maximum of four quantities: 0, `M[i,j-1]` − 8, `M[i-1,j]` − 8, and `M[i-1,j-1]`+`MatchQ[i,j]`. This definition represents a recurrence equation. Its last term depends on whether the query character `QS[i]` is equal to the data base sequence character `DB[j]`. Such a set of recurrences is often represented as a dependence graph as shown in Fig. 12.2. It should be noted, however that the ALPHA language allows one to represent arbitrary linear recurrences, which in general, cannot be represented graphically as easily. ALPHA allows structured systems to be described: a given system can be instantiated inside another one, by using a use statement which operated as a higher order *map* operator. For example

```
use {k | 1<=k<=10} sequence[X,Y] (a, b) returns (res)
```

would allow ten instances of the above sequence program to be instantiated. For the sake of conciseness, we do not detail in this chapter structured systems and refer the reader to [12].

Figure 12.3 shows the typical design flow of MMAlpha. MMAlpha allows ALPHA programs to be transformed, under some conditions, into a VHDL synthesizable program. The input is nested loops which, in the current tools, are described as an ALPHA program, but could be generated from loop nests in an imperative language (see [16] for example). After parsing, we get an internal representation of the program as a set of recurrence equations. Scheduling, localization and space–time mapping are then performed to obtain the description of a virtual architecture also described using ALPHA: all these transformations form the *front-end* of MMAlpha. Several steps allow the virtual architecture to be transformed to synthesizable VHDL

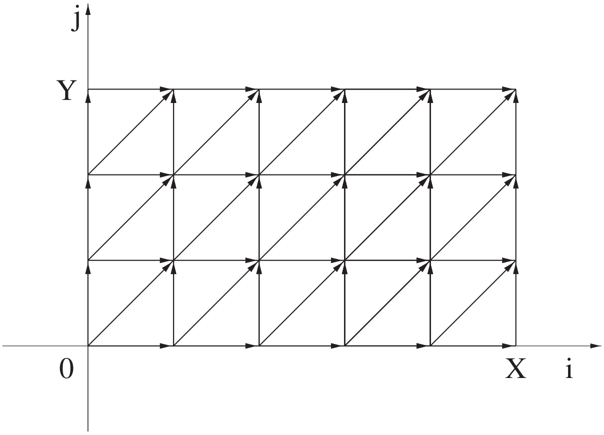

Fig. 12.2 Graphical representation of the string alignment. Each point in the graph represents a calculation `M[i,j]` and the arcs show dependences between the calculations

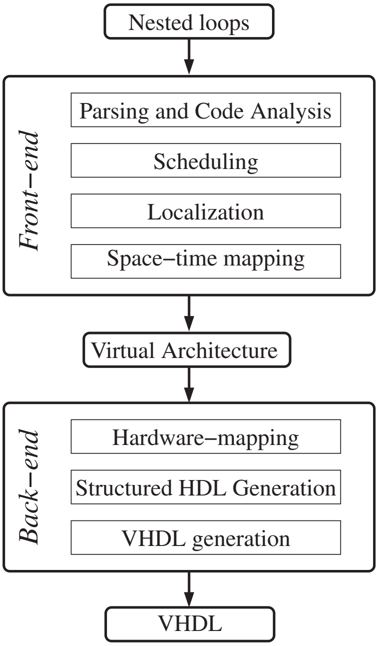

Fig. 12.3 Design flow of MMAlpha

code: hardware-mapping identifies ALPHA constructs with basic hardware elements such as registers, multiplexers, and generates boolean signal control instead of linear inequalities constraints. Then a structured HDL description incorporating a controller and data-path cells is produced. Finally, VHDL is generated.

In Sect. 12.3, we shall survey the front-end transformations whereas back-end will be presented in Sect. 12.4.

12.3 The MMAlpha Front-End: From Initial Specifications to a Virtual Architecture

The front-end of MMAlpha contains several tools to perform code analysis and transformations.

Code analysis and verification: The initial specification of the program, called here a *loop nest*, is translated into an internal representation in form of recurrence equations. Thanks to the polyhedral model, some properties of the loop nest can be checked by analysis: one can check for example that all elements of an *array* (represented by an ALPHA variable) are defined and used in a system, by means of calculations on domains. More complex properties of code can also be checked using verification techniques [8].

Scheduling: This is the central step of MMAlpha. It consists in analyzing the dependences between the variables, and deriving for each variable, say `V[i,j]` a timing-function $t_{\mathrm{V}}(i,j)$ which gives the time instant at which this variable can be computed. Timing-functions are usually *affine*, of the form $t_{\mathrm{V}}(i,j) = \alpha_{\mathrm{V}} i + \beta_{\mathrm{V}} j + \gamma_{\mathrm{V}}$ with coefficients depending on variable `V`. Finding out a schedule is performed by solving an integer linear problem using parameterized integer programming and is described in [17]. More complex schedules can be found: multi-dimensional timing functions, for example, allow some forms of loop tiling to be represented, but code generation is still not available for such functions.

Localization: It is an optional transformation (also sometimes referred to as uniformization or pipelining) that helps removing long interconnections [28]. It is inherited from the theory of systolic arrays where data which are re-used in a calculation should be read only once from memory, thus saving input–outputs. MMAlpha performs automatically many such localization transformations described in the literature.

Space–time mapping: Once a schedule is found, the system of recurrence equations is rewritten by transforming indexes of each variable, say `V[i,j]`, in a new reference index set `V[t,p]` where t is the schedule of the variable instance and p is the processor where it can be executed. The space–time mapping amounts formally to a *change of basis* of the domain of each variable. Finding out the basis is done by algebraic methods described in the literature (unimodular completion). Simple heuristics are incorporated in MMAlpha to discover quickly reasonable, if not always optimal, changes of basis.

After front-end processing, the initial ALPHA specification becomes a *virtual architecture* where each equation can be interpreted in term of hardware. To illustrate this, consider a sketch of the virtual architecture produced by the front-end from the string alignment specification as shown in Fig. 12.4. In this program, only

```
system sequence :{X,Y | 3<=X<=Y-1}
                (QS : {i | 1<=i<=X} of integer;
                 DB : {j | 1<=j<=Y} of integer)
       returns  (res : {j | 1<=j<=Y} of integer);
var
  QQS_In : {t,p | 2p-X+1<=t<=p+1; 1<=p} of integer;
    ...
  M : {t,p | p<=t<=p+Y; 0<=p<=X} of integer;
  ...
let
...
  M[t,p] =
      case
        { | p=0} : 0;
        { | t=p; 1<=p} : 0;
        { | p+1<=t; 1<=p} :
            Max4( 0[],
                  M[t-1,p] - 8,
                  M[t-1,p-1] - 8,
                  M[t-2,p-1] + MatchQ[t,p] );
      esac;
  QQS[t,p] =
      case
        { | t=p+1} : QQS_In;
        { | p+2<=t} : QQS[t-1,p];
      esac;

...
tel;
```

Fig. 12.4 Sketch of the virtual parallel architecture produced by the front-end of MMAlpha. Only variables `M` and `QQS` are represented. Variable `QQS` was produced by localization to propagate the query sequence to each cell of this array

the declaration and the definition of variable `M` (present in the initial program) and of a new `QQS` variable are kept. In the declaration of `M`, we can see that the domain of this variable in now indexed by t and p. The constraints on these indexes let us infer that the calculation of this variable is going to be done on a linear array of $X+1$ processors. The definition of `M` reveals several informations. It shows that the calculation of `M[t,p]` is the maximum of four quantities: the constant 0, the previous value `M[t-1,p]` which can be interpreted as a register in processor p, the previous value `M[t-1,p-1]` which was held in neighboring processor $p-1$, and value `M[t-2,p-1]`, also held in processor $p-1$. All these informations can be directly interpreted in term of hardware elements. However, the linear inequalities guarding the branches of this definition are much less straightforward to translate into hardware. Moreover, the number of processors of this architecture is directly linked to the size parameter `X`, which may not be appropriate for the requirements of a practical application: this is the rôle of the back-end of MMAlpha to transform this virtual architecture into a real one. The `QQS` variable requires some more explanations, as it is not present in the initial specification. It is produced by the localization transformation, in order to propagate the query value `QS` from processor to processor. A careful examination of its declaration and its definition reveals that this variable is present only in processors 1 to `X` and initialized by reading the value of another variable `QQS_In` when $t = p+1$, otherwise, it is kept in a register of processor p. As for `M`, the guards of this equation must be translated into simpler hardware elements.

12.4 The Back-End Process: Generating VHDL

The back-end of MMAlpha comprises a set of transformations allowing a virtual parallel architecture to be transformed into a synthesizable VHDL description. These transformations can be regrouped into three parts (see Fig. 12.3): hardware-mapping, structured HDL Generation, and VHDL generation.

In this section, we review these back-end transformations as they are implemented in MMAlpha by highlighting the concepts underlying them rather than the implementation details.

12.4.1 Hardware-Mapping

The virtual architecture is essentially an *operational parallel* description of the initial specification: each computation occurs at a particular date on a particular processor. The two main transformations needed to obtain an architectural description are: *control signal generation* and *simple expression generation*. They are implemented in the hardware-mapping component which produces a subset of ALPHA traditionally referred to as ALPHA0.

12.4.1.1 Control Signal Generation

It consists in replacing complex, linear inequalities by the propagation of simple control signals and is better explained here on an example. Consider for instance the definition of the `QQS` variable in the program of Fig. 12.4. It can be interpreted as a multiplexer controlled by a signal which is true at step `t=p` in processor number `p` (Fig. 12.5a). It is easy to see intuitively that this control can be implemented by a signal initialized in the first processor (i.e., value 1 at step 0 in processor 0) and then transmitted to the neighboring processor with a one cycle delay (i.e., value 1 at step 1 in processor 1, and so on). This is illustrated on Fig. 12.5b: the control signal `QQS_ctl` is inferred and is pipelined through the array. This is what the control signal generation achieves: to produce a particular cell (the *controller*) at the boundary of the regular array and to pipeline (or broadcast) this control signal through the array.

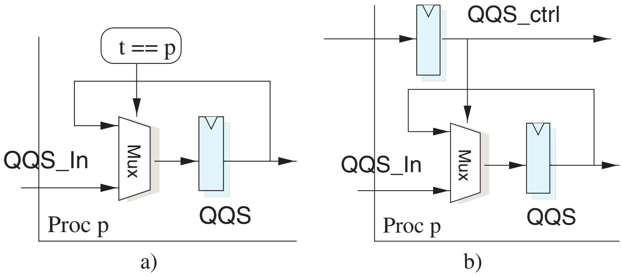

Fig. 12.5 Control signal inference for QQS updating

```
1  QQSReg6[t,p] = QQS[t-1,p];
2  QQS_In[t,p] = QQSReg6[t,p-1];
3  QQS[t,p] =
4      case
5        { |  1<=p<=X;} : if (QQSXctl1) then
6               case
7                 { | t=p+1;} : QQS_In;
8                 { | p+2<=t<=p+Y;} : 0[];
9               esac else
10           case
11             { | t=p+1; } : 0[];
12             { | p+2<=t<=p+Y; } : QQSReg6;
13           esac;
14     esac;
```

Fig. 12.6 Description in ALPHA0 of the hardware of Fig. 12.5b

12.4.1.2 Generation of Simple Expressions

This transformation deals with splitting complex equations in several simpler equations so that each one corresponds to a single hardware component: a register, an operator or a simple wire.

In the ALPHA0 subset of ALPHA, the RTL architecture can be very easily deduced from the code. For instance Fig. 12.6 shows three equations which represent: a register (line 1), a connexion between two processors (line 2) and a multiplexer (lines 3–14). They are interconnected to produce the hardware shown in Fig. 12.5b.

12.4.2 Structured HDL *Generation*

The second step of the back-end deals with generating a structured hardware description from the ALPHA0 format so that the re-use of identical cells explicitly appears in the structuration of the program and provision is made to include other components in the description. The subset of ALPHA which is used at this level is called ALPHARD and is illustrated in Fig. 12.7. Here, we have a module including

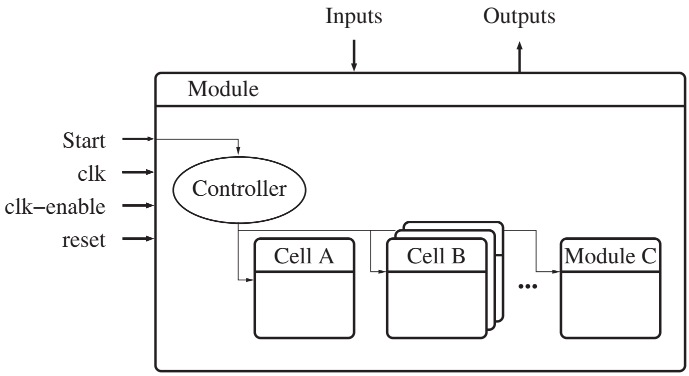

Fig. 12.7 An ALPHARD program is a complex module containing a controller and various instantiations of cells or modules

a local controller, a single instance of a `A` cell, several instances of a `B` cell and an instance of another module. Cells are simple data-paths whereas modules include controllers and can instantiate other cells and modules. Thanks to the hierarchical structure of ALPHA, it is easy to represent such a system in our language while keeping its semantics.

In the case of the string alignment application, the hardware structure contains, in addition to the controller, an instance of a particular cell representing processor $p = 0$, and $X - 1$ instances of another cell representing processors 1 to X. It is depicted in Fig. 12.8. (for the sake of clarity the controller and the control signal are not represented).

The main difficulty of this step is to uncover, in the set of recurrence equations of ALPHA0, the least number of common cells. To this end, the polyhedral domains of all equations are projected on the space indexes and combined to form space maximal regions sharing the same behavior. Each such region defines a cell of the architecture. This operation is made possible thanks to the polyhedral model which allows projection, intersection, unions, etc. of domains to be computed easily.

12.4.3 Generating VHDL

The VHDL generation is basically a syntax-directed translation of the ALPHARD program as each ALPHA construct corresponds to a VHDL construct. For instance, the VHDL code that corresponds to the ALPHA0 code shown in Fig. 12.6 is given in Fig. 12.9. Line 1 is a simple connexion, line 3 represents a multiplexer and lines 5–8 model a register. One can notice that the time index `t` disappears (except in the controller) as it is implemented by the `clk` and a clock enable signal.

If the variable sizes are not specified in the ALPHA program, the translator assumes 16-bit fixed-point arithmetics (using `std_logic_vector` VHDL type) but other signal types can be specified. VHDL test benches are also generated to ease the testing of the resulting VHDL.

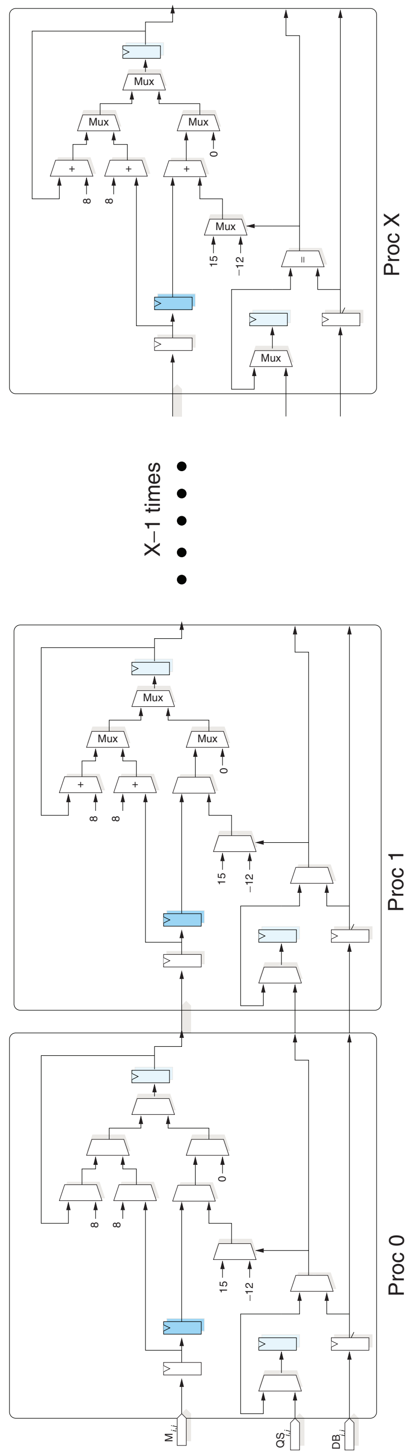

Fig. 12.8 Architecture of the string matching application

```
QQS_In <= QQSReg6_In;

QQS <= QQS_In WHEN QQSXctl1 = ‘1’ ELSE QQSReg6;

PROCESS(clk) BEGIN IF (clk = ‘1’ AND clk’EVENT) THEN
  IF CE=‘1’ THEN QQSReg6 <= QQS; END IF;
            END IF;
END PROCESS;
```

Fig. 12.9 VHDL code corresponding to the ALPHA0 code shown in Fig. 12.6

12.5 Partitioning for Resource Management

In MMAlpha, the choice of the various scheduling and/or space–time mappings can be seen as a design space exploration step. However there are practical situations in which none of the *virtual architectures* obtained through the flow matches the user requirements. This is often the case when iteration domains involved in the loop nests are very wide: in such situations, the mapping may result in an architecture with a very large number of processing elements, which often exceeds the allowed silicon budget. As an example, assuming a string alignment program with a query size $X = 10^3$, the architecture corresponding to the mapping proposed in Sect. 12.3 and shown in Fig. 12.4 would result in 10^3 processing elements, which represents a huge cost in term of hardware resources.

Many methods can be used to overcome such a difficulty. In the context of regular parallel architectures, *partitioning* transformations are the method of choice. Here, we consider a processor array partitioning transformation, which can be applied directly on the virtual architecture (i.e., at the RTL level).

Partitioning is a well studied problem [14, 25] and it is essentially based on the combination of two techniques. *Locally Sequential Globally Parallel* (LSGP) partitioning consists in merging several virtual PE into a single PE with modified time-sliced schedule. *Locally Parallel Globally Sequential* (LPGS) partitioning consists in *tiling* the virtual processor array into a set of virtual sub-arrays, and in executing the whole computations as a sequence of passes on the sub-array.

In the following, we present an LSGP technique based on *serialization* [13]: serialization merges σ virtual processors along a given processor axis into a single physical processor. One can show that a complete LSGP partitioning can be obtained through the use of successive serializations along the processor space axis.

To explain the principles of serialization, consider the processor datapath of the string alignment architecture shown in Fig. 12.10. We distinguish *temporal* registers (shown in grey) which have both their source and sink in the same processor, and *spatial* registers, the source and sink of which are in distinct processors. (We assume that registers have always a single sink, which is easy to ensure by transformation if needed.) Besides we assume that the communications between processing elements are unidirectional and pipelined.

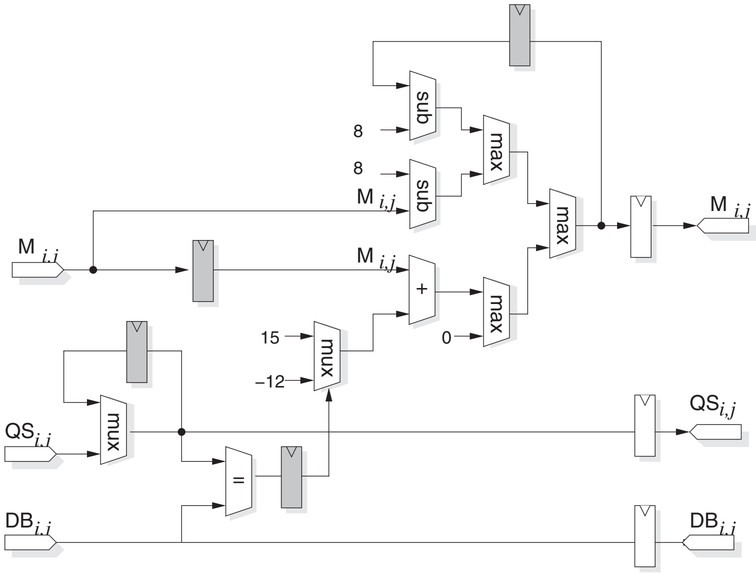

Fig. 12.10 Original datapath of the string alignment processor

Under these assumptions, serialization can be done in two steps:

– Any temporal register is transformed into a shift register line of depth σ.
– A one cycle delay feedback loop is associated to each *spatial register*; this feedback loop is controlled (through an additional multiplexer) by a signal activated every σ cycles.

Obviously, a serialization by a factor σ replaces an array of X processors by a *partitioned* array containing $\lceil X/\sigma \rceil$ processors. Figure 12.11 shows the effect of a serialization with $\sigma = 3$. This kind of transformation can be used to adjust the number of processors to the needs of the application. It can also be combined with various other transformations to cover a large set of potential hardware configurations. An example of hardware resource exploration for a bioinformatics application is presented in [11].

12.6 Implementation and Performance

To illustrate the typical performance of a parallel implementation of an application, we implemented on a Xilinx Virtex-4 device several configurations of string alignment with or without partitioning. The results are shown in Table 12.1. For each configuration, the number X of processors, the total resources of the device, – look-up tables, flip-flops and number of slices – the clock frequency and the performance, in Giga Cell Update per second (GCUps) are given. The last four lines present partitioned versions. As a reference, we show the typical performance of a software implementation of the string aligment on a desktop computer which

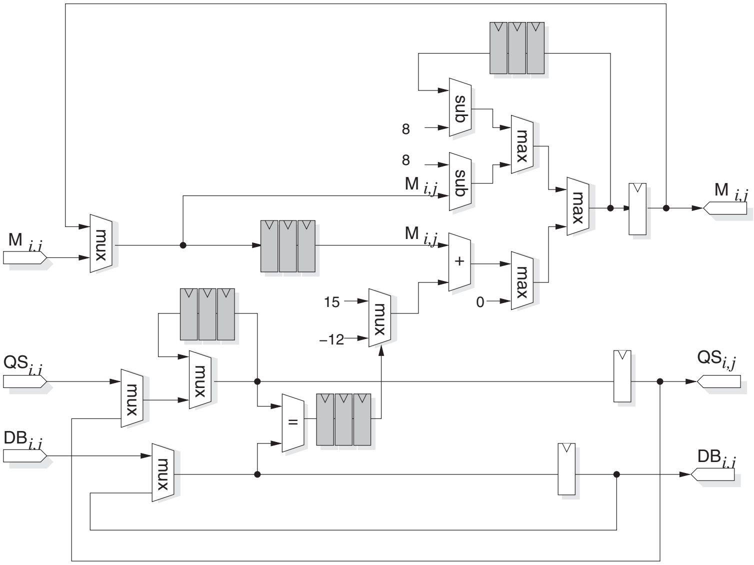

Fig. 12.11 The string alignment processor datapath after serialization by $\sigma = 3$

Table 12.1 Performance of various string alignment hardware configurations measured in Giga Cell Updates per seconds

Description	LUT/DFF/Slices	Clock (MHz)	Perf. (GCUps)
Software	–	–	0,1
$X = 10$	1,047/1,619/1,047	110	1.1
$X = 50$	8,088/4,130/4,771	110	5.5
$X = 100$	16,300/8,233/9,542	110	11
$X = 100, \sigma = 2$	10.8 K/4,308/6,628	95	5.5
$X = 100, \sigma = 10$	2 K/1 K/1,543	102	≈1.0
$X = 100, \sigma = 20$	1.2 K/550/931	93	≈0.45
$X = 100, \sigma = 50$	5.6 K/231/517	98	≈0.2

LUT is the number of look-up tables, *DFF* is the number of data flip-flops, and *Slices* is the number of Virtex-4 FPGA slices used by the designs

achieves 100 MCUps. The speed-up factor reaches up to two orders of magnitude depending on the number of processors. It is also noteworthy that the derived architecture is scalable: the achievable clock period does not suffer from an increase in the number of processing elements, and the hardware resource cost grows linearly with that number.

12.7 Other Works: The Polyhedral Model

The polyhedral model has been used for memory modeling [9, 15], communication modeling [33], cache misses [24], but its most important use was done in parallelizing compilers and HLS tools.

There is an important trend in commercial high-level synthesis tools to perform hardware synthesis from C programs: CatapultC (Mentor Graphics), Pico (Synfora) [30], Cynthesizer (Forte Design System) [18], and Cascade (Critical Blue) [4]. However all these tools suffer from inefficient handling of arbitrary nested loops algorithms.

Academic HLS tools are numerous and reflect the focus of recent researches on efficient synthesis of application-specific algorithms. Among the most important tools: Spark [19], Compaan/Laura [32], ESPAM [27], MMAlpha [26], Paro [6], Gaut [31], UGH [2], Streamroller [22], xPilot [7]. Compaan, Paro and MMAlpha have focused of the efficient compilation of loops, and they use the polyhedral model to perform loop analysis and/or transformations. Another formalism, called *Array-OL*, has been used for multidimensional signal processing [10] and revisited recently [5].

Parallelizing compiler prototypes have also provided a lot of research results on loop transformations [23]: Tiny [34], LooPo [20], Suif [1] or Pips [21]. Recently, WraPit [3], integrated in the `Open64` compiler, proposed an explicit polyhedral internal representation for loop nest, very close to the representation used by MMAlpha.

12.8 Conclusion

We have shown the main principles of high-level synthesis for loops targeting parallel architectures. Our presentation has used the MMAlpha tools as an example to explain the polyhedral model, the basic loops transformations, and the way these transformations may be arranged in order to produce parallel hardware. MMAlpha uses the ALPHA single-assignment language to represent the architecture, from its initial specification to its practical, synthesizable hardware implementation.

The polyhedral model, which underlies the representation and transformation of loops, is a very powerful vehicle to express the variety of transformations that can be used to extract parallelism et take benefit of it for hardware implementations. Future SoC architectures will increasingly need such techniques to exploit available multi-core architectures. We therefore believe that it is a good basis for carrying research on HLS whenever parallelism is considered.

References

1. S. Amarasinghe et al. Suif: An Infrastructure for Research on Parallelizing and Optimizing Compilers. Technical report, Stanford University, May 1994.
2. I. Augé, F. Pétrot, F. Donnet, and P. Gomez. Platform-Based Design From Parallel C Specifications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(12):1811–1826, 2005.
3. C. Bastoul, A. Cohen, S. Girbal, S. Sharma, and O. Temam. Putting Polyhedral Loop Transformations to Work. In *LCPC*, pages 209–225, 2003.

4. Critical Blue. Boosting Software Processing Performance With Coprocessor Synthesis, 2005. http://www.criticalblue.com.
5. P. Boulet. Array-OL Revisited, Multidimensional Intensive Signal Processing Specification. Research Report 6113, INRIA, February 2007.
6. M. Bednara and J. Teich. Automatic Synthesis of FPGA Processor Arrays from Loop Algorithms. *Journal of Supercomputer*, 26(2):149–165, 2003.
7. J. Cong, Y. Fan, G. Han, W. Jiang, and Z. Zhang. Platform-Based Behavior-Level and System-Level Synthesis. In *International SOC Conference*, pages 199–202. IEEE, 2006.
8. D. Cachera and K. Morin-Allory. Verification of Safety Properties for Parameterized Regular Systems. *Transaction on Embedded Computing Systems*, 4(2):228–266, May 2005.
9. F. Catthoor, S. Wuytack, E. De Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle. *Custom Memory Management Methodology*. Kluwer Academic Publishers, 1998.
10. A. Demeure and Y. Del Gallo. An Array Approach for Signal Processing Design. In *SAME 98*, October 1998.
11. S. Derrien and P. Quinton. Parallezing HMMER for Hardware Acceleration on FPGAs. In *ASAP07*, pages 10–17, Montreal, Quebec, July 2007.
12. F. Dupont de Dinechin, P. Quinton, and T. Risset. Structuration of the Alpha Language. In *Int. Conf. on Massively Parallel Programming Models*, Berlin, Germany, October 1995.
13. S. Derrien, S. V. Rajopadhye, and S. Sur-Kolay. Combined Instruction and Loop Parallelism in Array Synthesis for FPGAs. In *ISSS'01 : Proceedings of the International Symposium on System Synthesis*, pages 165–170, 2001.
14. A. Darte, R. Schreiber, B. R. Rau, and F. Vivien. Constructing and Exploiting Linear Schedules with Prescribed Parallelism. *ACM Trans. Des. Autom. Electron. Syst.*, 7(1):159–172, 2002.
15. A. Darte, R. Schreiber, and G. Villard. Lattice-Based Memory Allocation. *IEEE Transactions on Computers*, 54(10):1242–1257, 2005.
16. P. Feautrier. Dataflow Analysis of Array and Scalar References. *Int. J. Parallel Programming*, 20(1):23–53, February 1991.
17. P. Feautrier. Some Efficient Solutions to the Affine Scheduling Problem, Part I, One Dimensional Time. *Int. J. of Parallel Programming*, 21(5), October 1992.
18. Forte Design Systems. Cynthesizer Closes the ESL-to-Silicon Gap. http://www.forteds.com/products/cynthesizer.asp.
19. S. Gupta, R. Gupta, N. Dutt, and A. Nicolau. *SPARK: A Parallelizing Approach to the High-Level Synthesis of Digital Circuits*. Kluwer Academic, 2004.
20. M. Griebl and C. Lengauer. The Loop Parallelizer LooPo. In M. Gerndt, editor, *Proceedings of Sixth Workshop on Compilers for Parallel Computers*, volume 21 of *Konferenzen des Forschungszentrums Jülich*, pages 311–320. Forschungszentrum Jülich, 1996.
21. F. Irigoin, P. Jouvelot, and R. Triolet. Semantical Interprocedural Parallelization: An Overview of the PIPS Project. In *ACM International Conference on Supercomputing, ICS'91, Cologne*, June 1991.
22. M. Kudlur, K. Fan, and S. Mahlke. Streamroller: Automatic Synthesis of Prescribed Throughput Accelerator Pipelines. In *CODES+ISSS '06: Proceedings of the 4th International Conference on Hardware/Software Codesign and System Synthesis*, pages 270–275, New York, NY, USA, 2006. ACM Press, New York.
23. C. Lengauer. Loop Parallelization in the Polytope Model. In E. Best, editor, *CONCUR'93*, Lecture Notes in Computer Science 715, pages 398–416. Springer, Berlin Heidelberg New York, 1993.
24. V. Loechner, B. Meister, and P. Clauss. Precise Data Locality Optimization of Nested Loops. *The Journal of Supercomputing*, 21(1):37–76, 2002.
25. D. I. Moldovan and J. A. B. Fortes. Partitioning and Mapping Algorithms into Fixed Size Systolic Arrays. *IEEE Transactons on Computers*, 35(1):1–12, 1986.
26. A. Mozipo, D. Massicotte, P. Quinton, and T. Risset. Automatic Synthesis of a Parallel Architecture for Kalman Filtering using MMAlpha. In *International Conference on Parallel Computing in Electrical Engineering (PARELEC 98)*, pages 201–206, Bialystok, Poland, September 1998.

27. H. Nikolov, T. Stefanov, and E. Deprettere. Efficient Automated Synthesis, Programming, and Implementation of Multi-Processor Platforms on FPGA Chips. In *16th International Conference on Field Programmable Logic and Applications (FPL'06)*, pages 323–328, Madrid, Spain, August 2006.
28. P. Quinton and V. Van Dongen. The Mapping of Linear Recurrence Equations on Regular Arrays. *The Journal of VLSI Signal Processing*, 1:95–113, 1989.
29. R. Schreiber et al. PICO-NPA: High-Level Synthesis of Nonprogrammable Hardware Accelerators (HPL-2001-249), October 2001.
30. R. Schreiber, S. Aditya, B. R. Rau, V. Kathail, S. Mahlke, S. Abraham, and G. Snider. High-Level Synthesis of Nonprogrammable Hardware Accelerators. In *ASAP'00: Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures, and Processors*, page 113, Washington, DC, USA, 2000. IEEE Computer Society, Washington, DC.
31. O. Sentieys, J. P. Diguet, and J. L. Philippe. GAUT: A High Level Synthesis Tool Dedicated to Real Time Signal Processing Application. In *European Design Automation Conference*, September 2000. University booth stand.
32. T. Stefanov, C. Zissulescu, A. Turjan, B. Kienhuis, and E. Deprettere. System Design Using Kahn Process Networks: The Compaan/Laura Approach. In *DATE '04: Proceedings of the Conference on Design, Automation and Test in Europe*, page 10340, Washington, DC, USA, 2004. IEEE Computer Society, Washington, DC.
33. A. Turjan, B. Kienhuis, and E. F. Deprettere. Translating Affine Nested-Loop Programs to Process Networks. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 220–229, 2004.
34. M. Wolfe. A Loop Restructuring Research Tool. Technical Report CSE 90-014, Oregon Graduate Institute, August 1990.