Libraries of Schedule-Free Operators in Alpha

Florent de Dinechin
IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France
fdupont@irisa.fr

Abstract

This paper presents a method, based on the formalism of affine recurrence equations, for the synthesis of digital circuits exploiting parallelism at the bit-level. In the initial specification of a numerical algorithm, the arithmetic operators are replaced with their yet unscheduled (schedule-free) binary implementation as recurrence equations. This allows a bit-level dependency analysis yielding a bit-parallel array. The method is demonstrated on the example of the matrix-vector product, and discuted.

1: Introduction

Consumer market applications tend to require more and more processing power, a typical example of this trend being the need for real-time data compression or decompression in applications like mobile phones or high-definition television. These computational needs may only be matched by application-specific integrated systems, which have to be designed and tested in a very short time because of economic pressure. To achieve the design of these systems at a sensible cost, formal methodologies are needed to ensure that an architecture matches its specification. In this paper we consider such a methodology, based on the formalism of recurrence equations.

Affine recurrence equations (AREs) allow, on the one side, to express an algorithm at a very high level of abstraction which doesn't restrict its potential parallelism, and on the other side to express implementations of this algorithm as any combination of regular software (loop nests) and regular parallel hardware (processor arrays). Besides, there exists a set of formal techniques allowing one to transform the specification into an implementation in an automatic or semi-automatic manner, while ensuring that the functionality is preserved.

In this paper we address the synthesis, using ARE-based formalisms, of *digital* circuits, that is circuits computing numerical values. Usual design flows for digital circuits yield parallel architectures whose granularity is that of a numerical value. However, these numbers are implemented as fields of bits (words), and there is often some potential parallelism at the granularity of the bit which remains unexploited. Exploiting this bit-level parallelism may lead to architectures with bit-level pipelines, which may be either more efficient for a given silicon cost (bit-parallel architectures), or of much lower silicon cost with roughly the same computational power (bit-serial architectures).

There are technological reasons which make such architectures difficult to synthesize, such as in some cases a higher clock frequency for the same computing power. We will address these questions in the course of this paper. We believe, however, that the main obstacle to bit-parallel architecture synthesis is simply the added degree of complexity of

managing this parallelism: one has to rely on *libraries* of arithmetic operators to lower development time (e.g. datapath libraries), and these libraries impose a word granularity.

Therefore, we introduce the concept of *schedule-free operator libraries:* a schedule-free operator is simply a system of affine recurrence equations (SARE) describing an arithmetic operation at the most abstract level. For example a schedule-free multiplier may describe a multiplication as performed "by hand" (see Fig.1), but without imposing an order on the various bit-level computation involved.

The design process we propose is the following: a specification is written, simulated and validated at the word level, *i.e.* using abstract datatypes (real or integer). Then an automatic program transformation refines it into a functionally equivalent bit-level SARE, using a library of schedule-free arithmetic operators. This bit-level SARE then undergoes the classical synthesis process known as *dependency projection*, yielding a regular array of bit-level processors.

When this design process is successful, it has several strong points.

- From an abstract specification on abstract datatypes, it leads to an efficient bit-level design in an almost automatic manner.
- The word width is still a parameter at this stage, which allows one to simulate the design for several values of this word width very easily, until it matches the precision requirements of the specification.
- The design is highly portable, as it only consists of logical gates and flip-flops.

Unfortunately, this methodology does not always work: dependencies between or within schedule-free operators may prevent the existence of an affine schedule for the whole application. A solution, using non-standard representations of the numbers, is exposed.

This paper is organized as follows: in the first section, we introduce schedule-free operators by giving the recurrence equations for the addition and multiplication of real numbers expressed in the usual fixed point binary representation. We also introduce the synthesis methodology using AREs thanks to these examples, leading to bit-serial operators. In the second section we detail the program transformation which turns a specification into its bit-level version. We demonstrate the methodology on the simple algorithm of the matrix-vector product, and discuss some issues raised by our approach.

2: Arithmetic operators as affine recurrence equations

This section introduces the basic concepts of the SARE formalism in the Alpha language. The algorithms taken as examples are the binary addition and multiplication, as the corresponding SAREs will be the simplest of our schedule-free operators. The design methodology is then introduced by synthesizing a bit-serial multiplier from these SAREs. The interested reader is referred to [9, 15, 4] for a more extensive presentation of Alpha.

2.1: SAREs in Alpha

Program 1 is a simple Alpha program which describes a classical binary addition. Alpha variables (here `A`, `B`, `S`, `X`, `C`) denote data arrays defined over a domain which is a convex polyhedron of some integer vector space $\mathbb{Z}^n$. Here the domain of `C` is `{b| 0<=b<=W}`, where `W` is a size parameter defined in the header of the system: `{W| W>1}`. The values of a variable are defined on each point of this domain through recurrence equations involving the

Program 1 Binary addition in ALPHA

```
1    system Plus: {W| W>1}
2                 (A,B: {b| 0<=b<W} of boolean)
3          returns (S: {b| 0<=b<=W} of boolean);
4      var
5        X: {b| 0<=b<W} of boolean;
6        C: {b| 0<=b<=W} of boolean;
7      let
8        X[b] = A[b] xor B[b] xor C[b];
9        C[b] = case
10                {| b=0}  : 0[];
11                {| b>0} : A[b-1] and B[b-1]
12                            or A[b-1] and C[b-1]
13                            or B[b-1] and C[b-1];
14               esac;
15       S[b] = case
16                {| b<W} : X[b];
17                {| b=W} : C[W];
18               esac;
19      tel;
```

other variables, arithmetic or logical operators, and affine dependencies allowing to access the value of a variable at a different point (e.g. `C[b]` is defined as a function of `A[b-1]`, `B[b-1]` and `C[b-1]`). The `case` operator allows us to have several different expressions defining the values of a variable over distinct sub-domains (see the equation defining `C`, lines 9-14).

Program 2 describes the multiplication of two binary-coded fixed-point reals. A real number $x \in [0 \dots 1[$ is coded as a bit string $b_{W-1} \dots b_1 b_0$ such that $x = \Sigma_{i=0}^{W-1} b_i.2^{i-W}$. The product is performed as by hand (Fig.1), and only the W most significant bits are kept (which means that the result is rounded). Each line of this figure is the product (logical *and*) of one bit of the second operand by all the bits of the first one. These lines are accumulated thanks to "calls" to the system `Plus` of Prog.1. Note that the local variables (defined by the `var` keyword) are two-dimensional: the multiplication involves a two-dimensional array of computations (see Fig.1).

This example introduces the `use` statement [4], which instantiates a regular collection of subsystems. Line 9 should be read as follows:
"use a collection of instances of the system `Plus`, indexed by {`m| 0<=m<W`}, the value of the size parameter being `W` for each instance. The inputs to the `m`-th instance are the `m`-th lines of arrays `Si` and `P` (that is, `Si[.,m]` and `P[.,m]`), and the output goes to the `m`-th

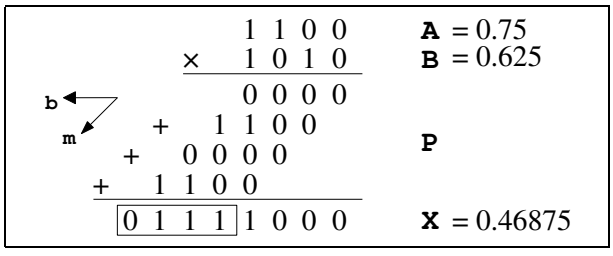


Figure 1. Product of two binary-coded fixed-point reals

Program 2 Binary multiplication in ALPHA

```
1   system Times: {W|W>1}
2                 (A,B: {b| 0<=b<W} of boolean)
3          returns (X  : {b| 0<=b<W} of boolean);
4      var
5        P, Si: {b,m| 0<=b,m<W} of boolean;
6        So: {b,m| 0<=b<=W; 0<=m<W} of boolean;
7      let
8        P[b,m] = A[b] and B[m];
9        use {m| 0<=m<W} Plus[W] (Si,P) returns (So);
11       Si[b,m] = case
13                   {| m=0} : 0[];
14                   {| m>0} : So[b+1,m-1];
15                 esac;
16       X[b] = So[b+1,W-1];
17     tel;
```

line of array `So`". The `use` statement will be crucial to the new program transformation introduced in this paper.

This description is very high-level, in the sense that it doesn't contain any information about the order of execution of the instructions (see equation defining `P`), and even less architectural information. An architecture will be a *refinement* of this specification, as described below.

2.2: Synthesis methodology using SAREs

We may now give a sketch of the synthesis flow using SAREs [9], as implemented in the Mathematica-based program transformation environment MMALPHA.

A program like that of Prog.2 is first *uniformized* [12, 16] to remove the data broadcasts and non-local communications. It is then *scheduled*, i.e. each computation of the program is assigned an affine time function consistent with the data dependencies [3]. Finally an affine *change of basis* is performed on the index space of each variable, so that one of the indices represents the *time* at which this variable is computed, and the other indices specify the *processor* on which the computation is performed, in some processor array whose shape is given by the resulting domains of the variables.

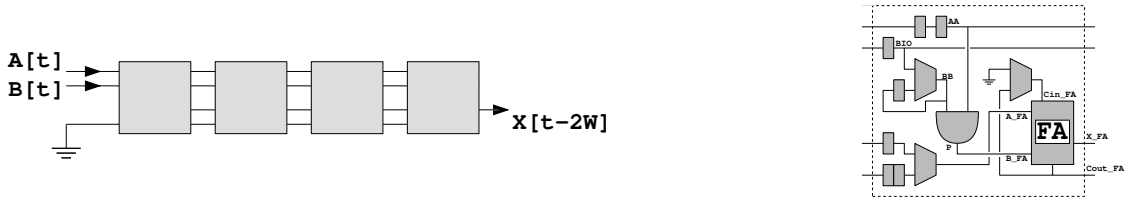
When this process is carried on Prog.2, we get an abstract description of the architecture described in Fig.2, whose core is partially given as Prog.3 (due to lack of space, some lines are deleted).

In this program, the data arrays are still two-dimensional as in Prog.2, but now the index `t` represents the time and the index `m` is the processor index in the linear array of Fig.2. The declaration domains of the variables show that there are `W` processors (indexed by `m` such that `0<=m<W`), and that the `m`-th processor computes for `2m<=t<2m+W`. Line 7 shows how `A[t]` is input at time `t` on the first processor (`m=0`), and line 8 shows how it is then propagated from processor `m-1` to processor `m` through two registers (implied by the `t-2` dependency). The computation equations of lines 11-14 are interpreted as hardware operators, whereas the data translation equations are interpreted as registers. Finally the last equation shows that the `b`-th bit of the result is output by the last (`W-1`-th) processor at time `b+2W-1`. The complete program thus describes a virtual linear array (Fig.2a).

This design is still very abstract. Additional lower-level program transformations are needed to turn control information present in the domains into systolic control variables [13]. In our example we get a bit-serial multiplier similar to Lyon's [10]. The resulting ALPHA program may then be translated [8] into structural VHDL for synthesis by commercial VLSI CAD tools like Compass or Synopsys. We will come back to these transformations in section 3.4.

Program 3 Fixed-point bit-serial multiplier

```
1  system times ...
2      var
3        AA,BB,So,A_FA,B_FA,in_FA,S_FA,Cout_FA:
4          {t,m| 0<=m<W; 2m<=t<2m+W} of boolean;
5      let
6        AA[t,m] = case
7                    {| m=0} : A[t];
8                    {| m>0}: AA[t-2,m-1];
9                  esac;
10       BB[t,m] = ... ;
11       A_FA[t,m] = ... ;
12       B_FA[t,m] = AA[t,m] and BB[t,m];
13       S_FA[t,m] = A_FA[t,m] xor B_FA[t,m] xor Cin_FA[t,m];
14       Cout_FA[t,m] = ... ;
15       So[t,m] = case
16                   {| t<2m+W} : S_FA[t,m];
17                   {| t=2m+W} : Cout_FA[t-1,m];
18                 esac;
19       X[b] = So[b+2W-1,W-1];
20     tel
```



a: Interpretation of Prog.3 **b: One cell of the final design**

Figure 2. Fixed-point bit-serial multiplier

3: Using schedule-free operator libraries

The ALPHA language also allows the user to express data arrays with `integer` or `real` abstract datatypes. For example the classical matrix-vector product is defined by the usual recurrence equation below:

$$\forall i \in \{1 \dots N\} \quad R_i = \sum_{j=1}^{N} M_{ij} V_j$$

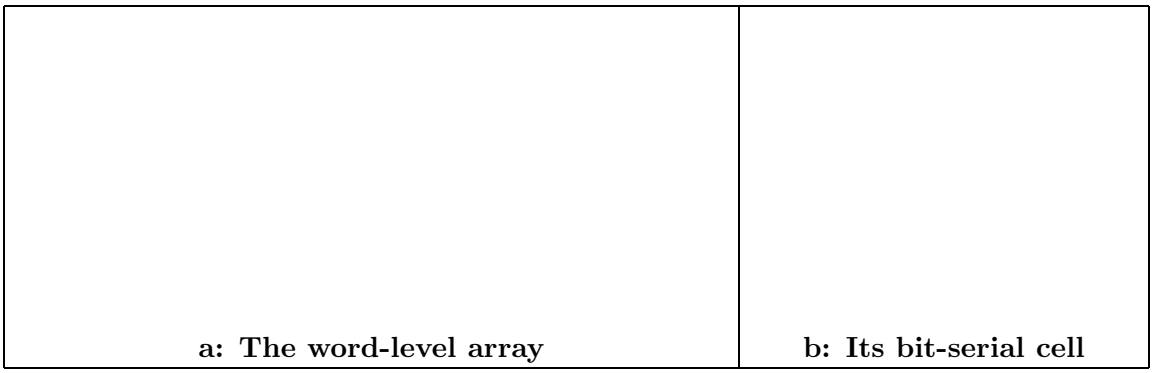
This equation is implemented straightforwardly as the ALPHA SARE of Prog.4 where we have serialized the summation by accumulating the partial results in C. The rest of this paper discusses the bit-level synthesis of such a word-level specification.

Program 4 Matrix-vector product on abstract real data

```
1       system matvect:{N | 1<=N}
2                      ( M: {i,j| 1<=i,j<=N} of real;
3                        V: {j| 1<=j<=N} of real )
4              returns ( R: {i| 1<=i<=N} of real );
5       var
6         C: {i,j| 1<=i<=N; 0<=j<=N} of real;
7       let
8         C[i,j] = case
9                    {| j=0} : 0[];
10                   {| j>0} : C[i,j-1] + M[i,j]*V[j];
11                esac;
12        R[i] = C[i,N];
13      tel;
```

3.1: Separate synthesis of the word-level array and of the operators

The first idea is to synthesize a word-level array for the matrix-vector product (we get the very classical systolic array depicted by Fig.3a), to synthesize bit-serial operators as we did in the previous section (a bit-serial adder consists of one full adder and two flip-flops), then combine them to get a bit-level circuit as shown by Fig.3b.



a: The word-level array | **b: Its bit-serial cell**

Figure 3. Bit-serial systolic array for the matrix-vector product

We won't elaborate on this approach: it needs multidimensional scheduling techniques [6] which haven't been implemented in ALPHA yet. Note that it boils down to defining libraries of bit-serial, that is *scheduled*, operators. The interested reader will find in [7] a survey of the parameters to take into account in this case.

3.2: Bit-level refinement transformation

We introduce here an automatic program transformation that replaces, in a word-level specification like Prog.4, the operators + and * with instances of the corresponding ALPHA

SAREs. The `real` variables are replaced with `boolean` ones, with one more dimension holding the bit representation.

For simplicity we present only the fixed-point representation of the reals, but this work may be straightforwardly adapted to integers. We haven't considered floating point representations yet: floating-point computations are based on fixed-point ones, but dynamic (i.e. data-dependent) shifting is usually needed for the normalization of the representation. This makes this kind of computations much less regular than fixed-point ones.

Our *bit-level refinement* transformation is performed in three steps.

First step: the operators are isolated by adding extra auxilliary variables, until they only appear in equations containing one and only one operator, without any dependency nor `case` statement. In our example the equation defining `C`, lines 8-11 of Prog.4, becomes:

```
aux1[i,j] = V[j];
aux2[i,j] = M[i,j] * aux1[i,j];     -- operator *
aux3[i,j] = C[i,j-1];
aux4[i,j] = aux3[i,j] + aux2[i,j];  -- operator +
C[i,j] = case
           {| j=0} : 0[];
           {| j>0} : aux4[i,j];
         esac;
```

The program transformations involved are not as trivial as it may seem, as we need to define the domain of the extra variables. In our example this is rather straightforward: the domain of each of the auxilliary variables has to be that of `M`. In the general case, the domain of the extra variable depends on the arguments of the operation, but also on the *context* of this operation. Consider the following equation:

```
C[i,j] = case
           ...
           {| i=j} : M[i,j] + C[i,j] ;
         esac;
```

In this case the domain of the extra variable `aux[i,j] = M[i,j] + C[i,j];` would not be the domain of `M` (a full matrix) but only its diagonal defined as `{i,j| 1<=i,j<=N; i=j}`. The purpose of this first step is therefore to compute the domain of the operators: in the last example the `+` describes a linear collection of additions placed on the diagonal of a square matrix, and the domain of `aux` must be defined as this domain.

The rules allowing us to compute the domains of the extra variables are deduced from the denotational semantics of Alpha, and computed thanks to operation on polyhedra [14] which are well known and already widely used in the MmAlpha environment [5].

Second step: a parameter, `W`, is added to the system, representing the word size, and all the domains of the variables are augmented by one dimension (indexed by `{b| 0<=b<W}`) denoting the bit representation. Then all the expressions of the program are modified to take into account the extra dimension: a recursive search over the expression trees extends the dimensions of the `case` domains and affine dependencies.

Third step: an operator equation generated in the first step is replaced with a `use` statements describing the same collection of operations, using subsystems from a schedule-free operator library.

In our example, the variable `aux4`, extended at the second step:

```
aux4[i,j,b] = aux3[i,j,b] + aux2[i,j,b];  -- operator +
```

is replaced with:

```
use {i,j| 1<=i,j<=N} Plus[W] (aux3, aux2) returns (aux4);
```

In this `use` line, there is a domain which describes the shape of the collection of instances of the system `Plus`. This domain is taken as the domain of `aux4` defined in the first step: remember that this domain exactly described the shape of the collection of additions to be performed. Thus this `use` line describes the same collection of additions as in the initial program, only at the bit level, and thus the functionality of the program is preserved.

3.3: Synthesis of the bit-level architecture

Now we have automatically derived a SARE specification of the bit-level array, which may undergo the synthesis process. For this the operator subsystems (`Plus` and `Times` in our example) need to be *inlined* [4] (currently the MmAlpha tools don't allow the scheduling and space/time projection of structured SAREs). Uniformization of the dependencies may be performed before or after inlining. Finally the resulting system is scheduled and space/time transformed, yielding a systolic array. This scheduling takes into account the dependencies at the bit level, therefore the operators are pipelined at the bit level. Besides, we still have at this point a parameterized design where we may vary the bit size and the parameters of the problem (here `W` and `N`).

In the case of the matrix-vector product, we get a 3-dimensional array (three processor indices, one time index). We do not give the Alpha SARE due to lack of space. For synthesis, the three processor indices need projecting on a 2-dimensional space (the chip) as represented in Fig.4. Note that some local data transfers in the 3-dimensional array lead to long (but still regular) wires in the actual circuit. The figure shows the bit arrays for each matrix or vector element, input and output in a *parallel skewed* manner (the isochrone lines are also shown). As previously, we read from the Alpha program that, if the first bit of the matrix is input at time 0, the b-th bit of output vector element R_i is output from the `X` output of the last line of full adders at time $T_R(b,i) = b + i + N + 2W + o(1)$.

It is interesting to note that this array has a latency linearly better than that of the bit-serial array, while its silicon cost is linearly worse. One may also check that, with only a little more control, this circuit may be used to perform several matrix-vector products in a pipelined manner.

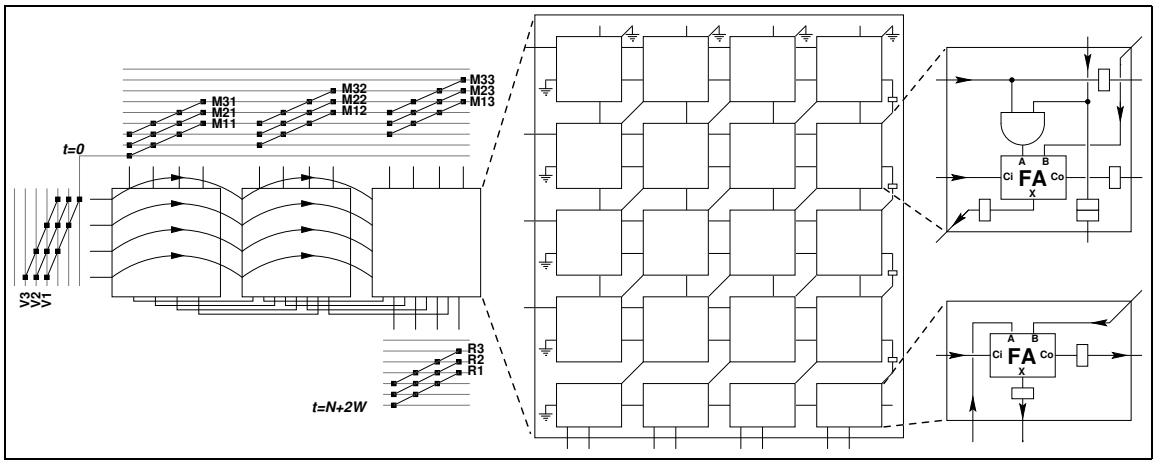


Figure 4. Bit-parallel systolic array for the matrix-vector product

3.4: Complexity and VLSI implementation issues

Let us try and compare the time and space cost of our approach with that of a more standard (typically, datapath) synthesis. The area of a datapath multiply-and-add is always roughly proportional to W^2, just as in our circuit, but our circuit involves more registers and more control (a register is about as big as a full adder). We will therefore need to evaluate this area issue on an experimental basis.

Now let us discuss the clock frequency. Our bit-parallel circuit outputs one value each clock cycle thanks to the bit-level pipeline, just like the datapath version. However, looking at Fig.4, we see that the critical path of our circuit goes only through one *and* gate and one full adder, whatever the word width. This is much shorter than the critical path of any datapath multiply-and-add. Therefore we may hope to clock our circuit much higher than its datapath equivalent, and almost independently from the word width.

The point is that we then have W times more clock nets in our design than in the datapath version (in Fig.3a the clock net is only $W \times N$ big). The design of clock trees for synchronous circuits is an ever-increasing part of the development cost, and our approach obviously makes this task even more complex. Here again, a full experiment is needed. Preleminary results, however, are very encouraging. Besides the regularity of the circuit greatly simplifies the design of the clock tree. In particular, uniformizing the clock skews will be made easy by the fact that all the cells are identical. Automatic tools for synthesising regular clock trees are indeed possible, the main difficulty, for us, being that they don't fit well in the ARE framework (trees are not affine data structures expressible as polyedra).

3.5: Generality issues

Another important question is the range of algorithms which will have interesting bit-parallel implementations thanks to schedule-free operators. Obviously we have to restrict ourselves to algorithms whose word-level version has an affine schedule. Among these algorithms, it is difficult to know in advance those which will benefit from a bit-level approach: this requires a dependency analysis which is no simpler than that performed by the Alpha scheduler. Our experience shows that it works for most algorithms based on sums of products (such as convolutions and basic matrix operations).

It is easier, however, to point out algorithms which surely won't benefit from a bit-level dependency analysis. To start with, the previous schedule-free addition operator involves a carry propagation, which is a data dependency from the least significant digits to the most significant ones. *Division* algorithms produce their results most significant digits first (see the division "by hand"). This means that no efficient bit-pipelined architecture is possible using both of these operators.

A well-known solution is the use of another representation of the numbers: hardware implementations of the division operation, for example, usually rely on redundant digit sets [1, 11] for this reason. We have implemented some operators in binary-signed (BS) arithmetics [2]. In this case the addition may be performed in parallel (there is no carry propagation), and therefore doesn't constrain the schedule. It is also the case of the multiplication (although a schedule most significant bits first requires a diffusion, i.e. a one-to-all communication). Similarly most basic operators may be scheduled most significant bits first. We still have to evaluate the cost and efficiency of circuits designed using a BS schedule-free library.

4: Conclusions and future work

This paper is a contribution to the the field of computer-aided design of digital circuits, extending a standard methodology based on systems of affine recurrence equations. We provide an automatic tool for translating a high-level algorithmic specification, operating on words (integers or reals), into an equivalent bit-level one. This tool replaces the word-level operators of the initial specification with their bit-level counterpart, expressed as yet unscheduled Alpha SAREs. When synthesizing this bit-level specification, we get bit-level designs which are more portable than datapath implementations, very efficient thanks to bit-level dependency analysis, and fully parameterized.

Our current work on this topic is an evaluation of the quality of the circuits obtained through these techniques. Preliminary results are very positive. Future works include, in the short term, the completion of the synthesis flow down to silicon (that is, structural VHDL plus placement directives), and also the extension of these techniques to more operators and more number representations.

References

[1] A. Avizienis. Signed-digit number representations for fast parallel arithmetic. *IRE Transactions on Electronic Computers*, 10:389–400, 1961.

[2] Jean-Claude Bajard, Jean Duprat, Sylvanus Kla, and Jean-Michel Muller. Some operators for on-line radix-2 computations. *Journal of Parallel and Distributed Computing*, 22(2):336–345, August 1994.

[3] A. Darte and Y. Robert. Constructive methods for scheduling uniform loop nests. *IEEE Trans. Parallel Distributed Systems*, 5:814–822, 1994.

[4] Florent de Dinechin, Patrice Quinton, and Tanguy Risset. Structuration of the Alpha language. In W.K Giloi, S. Jahnichen, and B.D. Shriver, editors, *Massively Parallel Programming Models*, pages 18–24. IEEE Computer Society Press, August 1995.

[5] Florent de Dinechin and Sophie Robert. Hierarchical static analysis of structured systems of affine recurrence equations. In *Application Specific Array Processors*. IEEE Computer Society Press, August 1996.

[6] Paul Feautrier. Some efficient solutions to the affine scheduling problem, part II, multidimensional time. *Int. Journal of Parallel Programming*, 21(6):389–420, December 1992.

[7] P. Frison, P. Gachet, and P. Quinton. Designing systolic arrays with DIASTOL. In S.Y. Kung, R.E. Owen, and J.G. Nash, editors, *VLSI Signal Processing II*, pages 93–105. IEEE Press, November 1986.

[8] Patricia Le Moënner, Laurent Perraudeau, Sanjay Rajopadhye, Tanguy Risset, and Patrice Quinton. Generating regular arithmetic circuits with AlpHard. In *Massively Parallel Computing Systems (MPCS'96)*, pages 429–436, May 1996.

[9] Hervé Le Verge, Christophe Mauras, and Patrice Quinton. The Alpha language and its use for the design of systolic arrays. *Journal of VLSI Signal Processing*, 3:173–182, 1991.

[10] R. F. Lyon. Two's complement pipeline multipliers. *IEEE Trans. Comm.*, 24:418–425, April 1976.

[11] Stuart F. Oberman and Michael J. Flynn. An analysis of division algorithms and implementations. Technical Report CSL-TR-95-675, Stanford University, July 1995.

[12] S. V. Rajopadhye, S. Purushothaman, and R. M. Fujimoto. On synthesizing systolic arrays from recurrence equations with linear dependencies. In *Sixth Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 488–503, New Delhi, India, Dec 1986. Springer Verlag, LNCS No 241.

[13] Jürgen Teich and Lothar Thiele. Control generation in the design of processor arrays. *Journal of VLSI Signal Processing*, 3:77–92, 1991.

[14] D. Wilde. A library for doing polyhedral operations. Publication Interne 785, IRISA, Rennes, France, December 1993. Also published as INRIA Research Report 2157.

[15] Doran K. Wilde. The Alpha language. Publication Interne 827, IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France, January 1994.

[16] Yoav Yaacoby and Peter R. Cappello. Converting affine recurrence equations to quasi-uniform recurrence equations. In *AWOC 1988: Third International Workshop on Parallel Computation and VLSI Theory*. Springer Verlag, June 1988. See also, UCSB Technical Report TRCS87-18, February 1988.