A LUT based Approach for High Level Synthesis on FPGAs

Loic Lagadec, Bernard Pottier, Oscar Villellas, Erwan Fabiani, Catherine Dezan

▶ **To cite this version:**

Loic Lagadec, Bernard Pottier, Oscar Villellas, Erwan Fabiani, Catherine Dezan. A LUT based Approach for High Level Synthesis on FPGAs. International Workshop on Logic and Synthesis (IWLS), Jun 2002, New Orleans, United States. hal-01862801

HAL Id: hal-01862801
https://hal.univ-brest.fr/hal-01862801

Submitted on 27 Aug 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A LUT based approach for high level synthesis on FPGAs

Loïc Lagadec, Bernard Pottier, Oscar Villellas,
Erwan Fabiani and Catherine Dezan
Architectures et Systèmes – Université de Bretagne Occidentale
UFR Sciences, 20 av. LeGorgeu, Brest 29285, France
{fabiani,loic.lagadec,dezan,pottier,villella}@univ-brest.fr

1. Introduction

Development methods for reconfigurable architectures (RA) are dominated by industrial tools largely inherited from VLSI CAD. Each new architecture is delivered with its own set of libraries and tools for floorplanning, place and route. These tools are generally addressed directly or indirectly by hardware description languages (HDL).

Some initiatives such as the public architecture of Xilinx 6200, or Xilinx Jbits API have provided open environments for circuit design. Academic research efforts are resulting in generic low level tools such as VPR[1], circuit components generators, compilers built on the top of HDL and high level logic generators. However, important points such as source portability, reuse, open access to architecture resources, debugging, are far to be reached. This prohibits a more general use of RA, and freeze progresses in software engineering.

MADEO is a medium term project that uses the object paradigm to provide development flexibility, portability and openness on RAs.

The project structure has three parts that interact closely (bottom-up):

1. **reconfigurable architecture model and its associated generic tools.** The representation of practical architectures using a generic model provides immediately operational tools[5]. Mapping logic description to a particular technology is achieved using the algorithms packed into SIS[9], or a hierarchical and parallel variant from Lemarchand's PPart[3].

   With the basic functionalities provided in this framework, circuits can be placed, routed, assembled, drawn, and practically loaded on a platform. Specific "atomic" resources can be merged with logic, and the framework is extensible.

2. **high level function compiler.** With given restrictions, this framework allows to manage library-independent compilation to RAs. This includes data binary presentation, and combinational logic generation. An abstract intermediate level within this framework is a graph of lookup-tables carrying high level values, that could be interpreted or translated into a logic graph. When used in conjunction with the first framework, this part provides a service comparable to a specific logic function generator, or to a meta-compiler for specific arithmetics and data.

3. **System and architecture modeling.** The computation architecture in its static or dynamic aspects is described in this framework. For instance, these are generic regular architectures with their associated tools, processes, platform management and system activity.

   The compiler can make use of logic generation to produce RA configurations, bind them to registers or memories, and produce a configured application. The ability to control placing and routing given by the first part, and synthesis from the second part, yet allow to build complex networks of fine or medium grain elements.

This paper focuses on the HLL compiler supporting logic generation. Synthesis is based on *enumerated symbols* rather than on *interval of values* giving the possibility to decrease automatically the logic complexity in the cases of sparse set of operands. Ideas from a Lin, Whitcomb and Newton paper[4] have been reused in the context of object oriented programming targeting specifically FPGAs. An initial work has described a synthesis technique based on a translation from object blocks of code to one PLA which is mapped to FPGA look-up tables[6]. This approach was limited by the minimization and mapping problem handled at the logic level. The major improvement described in

this paper is that the programmer can split its description in procedure calls allowing the compiler to remove a lot of the complexity by working on a graph of symbol indexes. Logic production occurs at manageable procedure calls, with a direct effect on the synthesis time (minutes rather than hours, and synthesis succeeds). Furthermore, there is now a full set of low level tools and models that will permit to allocate a variety of hardware resources from the semantic level or from compiler decisions.

The text describes the general principles used for specification and logic production, then the transformations achieved by a compiler. An illustration is given with the examples of a small floating point multiplier and of a coder/decoder family for RAID systems (with quantitative results).

2. HLL to FPGAs using network of LUTs

2.1. FPGA modeling

Reconfigurable architectures can mix different grain of hardware resources: logic elements, operators, communication lines, buses, switches, memories, processors...

FPGAs are often built with small lookup memories (LUT) addressed by a set of signals. As seen from the logic synthesis tools, an n-bit wide LUT is the most general way to produce any logic function of n boolean variables. There are known algorithms and tools for partitioning large logic tables or networks to target a particular LUT-based architecture.

LUTs are effectively interconnected during the configuration phases to form logic. This is achieved using various configurable devices such as programmable interconnect points, switches, or shared lines. Some commercial architectures also group several LUTs and registers into cells called configurable logic block (CLB).

Our model for the organization of these architectures is a hierarchy of geometric patterns of hardware resources. The model is addressed to describe concrete architectures via a specific grammar[5]. Given this FPGA description, generic tools operate for technology mapping, placing and routing logic modules. Circuits such as operators or computing networks are described by programs realizing the geometric assembly of such modules and their connection.

2.2. Programming considerations

Applications for fine grain reconfigurable architectures can be specialized without compromise, and they should be optimized in terms of space and performance.

In our view, there is an abusive advantage given to the local performance of arithmetic units in the synthesis tools and also in the specification language.

A first aspect of this advantage is the small range of basic types coming from the capabilities of ALU/FPUs or memory address mechanisms. Control structures strictly oriented toward sequentiality are another aspect that can be criticized. As example, programming SIMD multimedia accelerators remains procedural despite all the past experience in the domain of data parallelism. Hardware description languages have rich descriptive capabilities, however the necessity to use libraries led the language designers to restrict their primitives to a level similar to **C**.

Our aim is to produce a more flexible specification level for the programmers with direct and efficient coupling to logic. This implies allowing easy creation of specific arithmetics representing the algorithmic needs, letting the compilers automatically tune data width, and modeling computations with properties similar to object-oriented modularity.

To obtain this, specifications which are symbolic and functional are used with separate definition of data on which the program will operate (data are objects). Later, sequential computations can be structured in various ways by observing the read and write transactions to memories or registers, either explicitly in the case of architecture description, or implicitly under architecture synthesis control. Reference [6] shows an early work on state machine synthesis compatible with the present method.

In this paper we will consider the case of methods without side effect, operating on a set of objects. For sake of simplicity we will rename these methods '*functions*', and the set of objects, '*values*'. Interaction with external variables is not discussed in this paper. The input language is Smalltalk-80, variant Visualwoks, also used to build the tools and to describe the application architectures.

The example below is the code for a family of multipliers suitable for synthesis, provided that their part are of limited size. The multipliers take two operands known by their sign, exponents, and fractional parts. Elementary operations are described by additional methods (not shown), and the use of local variables. The result is an aggregation of the variables `sign, exp` and `mant` in a new object.

sign: signA significand: significandA
exponent: exponentA
sign: signB significand: significandB
exponent: exponentB

| sign exp mant normalize |

```
sign := self computeSignFor: signA and: signB.
exp := self computeExponentFor: exponentA
and: exponentB.
mant := self computeSignificandsFor: significandA
and: significandB.
normalize := self normalize: mant.
exp := exp + normalize.
mant := mant / (10 raisedTo: normalize).
```

This code must be complemented by types for the six parameters. Types consists of the specification of possible values for sign, exponent and significand. It also defines the precision of the number, that can be used with the same status as standard floating points due to the object oriented language.

2.3. Execution model

The execution model used by the compiler is a high level replication of LUT-based FPGAs. We define a '*program*' as a function that needs to be executed on a set of input values. Thus the notion of *program* groups at once the algorithm and the data description. Our *program* can be embedded in higher level computations of various kind, implying variables or memories. Data descriptions are inferred from these levels.

An execution is the traversal of a hierarchical network of lookup tables in which values are forwarded. A value change in the input of a table implies a possible change in its output that in turn induces other changes downstream. These networks reflect the effective function structure at the procedure call grain and they have an algorithmic meaning. Among the different possibilities offered for execution, there are cascaded hash table accesses, and use of general purpose arithmetic units where they can fit.

Translation to FPGAs can take advantage of the staticity of the tables by exchanging *values* appearing in the input and output for *indexes* in the enumeration of values. Figure 1 shows fan-in and fan-out cases with the aggregation of indexes in the input, and the *next index* selection from the table.

There are some important results or observations from this exchange:

1. data paths inside the network do not depend anymore on data width but on *the number of different values present on the edges*.

2. depending on the requirements at the higher level using the program, it will be needed to insert nodes in the input and output of the network to handle the exchanges between values and indexes.

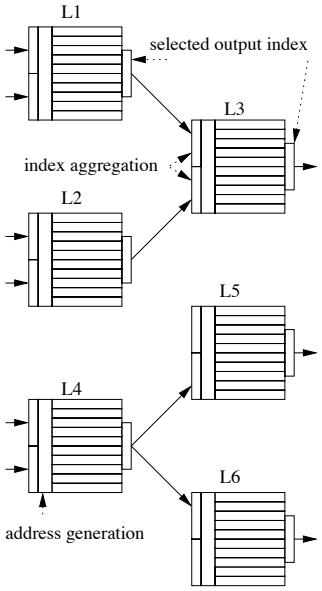

Figure 1. Fan-in: index are aggregated to form an address in the table. Fan-out: the same index is presented to each table downstream.

3. a support is needed to produce the addressing mechanism for index tables. As we are willing to map on FPGAs, it is sufficient to consider the tables as a logic table producing the output indexes from the aggregation of the input indexes. Such a table can be handled by logic mapping algorithms mentioned section 1.

4. logic synthesis capability is limited to medium grain problems. To allow compilation to FPGAs, algorithms must progressively *decrease the number of values* down to nodes that can be easily handled by the bottom layer (SIS partitioning for LUT-n). Today, this grain is similar to algorithms coded for 8-bit microprocessors.

5. *decreasing the number of values* is the natural way in which functions operates, since the size of a Cartesian product on a function input values is the maximum number of values produced in the output.

2.4 Type system

Language types appear to the programmers as annotations for checking code consistency and binding to architecture resources. The type system we are using does not restrict to this kind of binding. It is only intended to specify any possible set of values appearing

in the program input or inside the computation network. In the object environment, it is supported by a set of classes supporting operations.

Implicit or explicit collections of values are denoted by intervals or sets. *Class-based classes* are associated either to classes having a finite number of instances (booleans, bytes, small integers), or to user defined new functionalities, including arithmetics. *Unions* are resulting from operations on the two previous types.

3. Compiler flow

3.1. Flat expressions

Let us consider a program where the number of values appearing in the input of each function call is compatible with an efficient logic synthesis (partitioning for LUT-n). Each node being directly synthesized, we have a *flat* expression in opposition with *hierarchical* expressions that will need to build new compilation contexts for some function calls.

As a Smalltalk development environment is used, there is an obvious interest to use the same syntax for '*programs*' targeting FPGAs. An immediate benefit is the reuse of the standard compiler front-end.

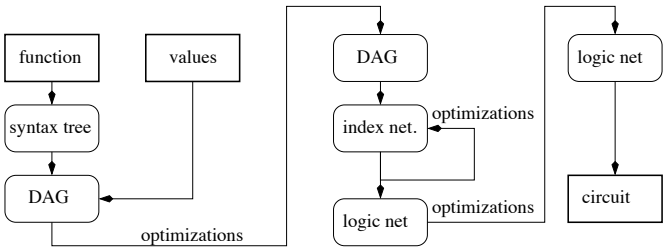

Figure 2. Compiler flow

1. **Building the value network**

   The first compilation stage consists in building an acyclic flat graph which nodes are lookup tables based on values and which edges allow to pass values downstream.

   As stated, the syntax tree is produced by the standard compiler. The directed acyclic graph (DAG) is built by analyzing the syntax tree and variable use. Local variable references are eliminated. At this stage nodes are still holding function calls receiving edges from the function parameter list, or other nodes.

   To replace these nodes by lookup tables, the values are propagated progressively from the function parameter list. A graph traversal is achieved mostly breadth-first, building a table for each ready node.

   During this transformation care must be taken of dependent variables in fan-out to fan-in subnets. As $h(f(x,y),g(x,z))$ has a smaller output than $h(f(x,y),g(t,z))$, a number of inputs in the fan-in node and upstream are not useful and can be deleted.

   Another focus of interest is the special meaning of some nodes such as the conditional instructions. For these cases the input values in the true and false branches must reject the values selected or discarded by the conditional expression.

2. **High level optimization and building the index network**

   After this first stage we have a situation similar to a compiler having a language semantic knowledge because the tables have inferred equivalent properties from message execution. It is time to apply usual high level optimizations such as elimination of constant nodes and dead code or subexpression factorization. This imply backward and forward processing on the DAG.

   The next transformation is the translation of the DAG by deducing index based tables from associations of value tables. This is achieved by generating index for values. Care must be taken of class based types to preserve their special encoding.

3. **LUT based optimizations and architecture mapping**

   Index path optimizations involves the detection of subnets with particular topologies. As an example, linear cascade of tables can be grouped in a single table. For logic translation, each index-based table is given to logic synthesis tools to produce an equivalent binary description. At this stage we must take into account the size of LUT memories in the target architecture. The result is a two level hierarchical logic description which is a binary equivalent for the high level program.

   The last stage is to place and route the logic graph using the generic tools in the RA framework, producing a hardware module.

3.2. Hierarchical aspects

In section 3.1 the program was supposed to be directly synthesizable at each function call. We are now considering the more general case where calls must be developed to reach this condition.

The logic needed to implement a particular function call depends on the expressed algorithm, the number of parameters, the number of possible values for parameters and the original encoding of values in the higher level environment. A valuable property of an algorithm is its ability to quickly decrease the number of values present on graph edges. This gradual decrease comes from function calls that are processed in the same way as their root function, for every node showing an excessive complexity related to synthesis.

When the compiler reaches such a condition, it creates a new compilation context and process recursively the call. The context will return a logic description that will be installed as part of the current level production.

The general form of a logic description associated to a compiled program is a hierarchical description that can be partially flattened for further logic optimization, and partially placed under control of a floor planner.

A more speculative compiler built-in function is type partitioning. When a data set appears to be too much large, the compiler decides to partition the type in order to reach a synthesizable grain. Automatic type division by the compiler should be considered only as a quick approximation, since the *function* algorithms are normally written to manage synthesis complexity at high level.

A similar situation is the knowledge of a 'best encoding' for values. As an example, the order of elements in a Galois field has an influence on the logic complexity of basic operators. If these operations are dominant in the code, rules must be attached to the compiler to prevent new type generation in node outputs.

4. Examples

4.1. Floating point multiplier

Here we comment synthesis of the multiplier described in section 2.2. This operator is designed and checked in the high level environment, as a normal method used in numeric algorithms. By adding types, we allow the compiler to produce a logic circuit equivalent to the software behavior.

The DAG is built and the lookup tables associated with each node are computed. Depending on the local complexity, an additional development of a call can be executed.

A range of multipliers up to 12 bits $\times$ 12 bits have been synthesized in times not exceeding 5 minutes (1Ghz PIII). All these multipliers have been post-optimized using standard algorithms from SIS. This is achieved in a maximum delay of 1h 30 minutes with a logic reduction of 25% for 500 luts of 4 variables. We think that post-optimization can be executed more efficiently by local algorithms rather than flattening the circuit (work in progress).

Remember that this multiplier can be optimized when it is used in expressions involving constants or set of values.

4.2. Results for RAID corrector

A first example on which the basic approach has been applied successfully is a generator for a family of Reed-Solomon decoders for a RAID system. These decoders are basically linear system solver used with Galois field arithmetic to reserve the reversibility. The generator produces corrector for each possible failures. A corrector is a large mathematical expression that can be computed on different kind of numbers. By presenting data types and expressions to the compiler, we are able to produce a synthesizable program as long as the elementary operations in GF are synthesizable. Our compiler has processed automatically each of the possible decoders, some of them being placed and routed on abstract FPGAs similar to Xilinx architectures to provide area and delay characteristics.

The high level part of the development has been achieved in the order of two days based on Plank's tutorial[8]. The application is interesting for FPGA implementation since each possible disk failure can have its specific configuration correction circuit.

The RAID system has n data disk and p redundancy disks. A valid example is $n = 8, p = 2$, and a particular decoder takes its input in the 10 disk array producing the output for a particular disk.

The table 1 shows the average numbers of nodes obtained for all the possible decoders associated to failures with $n = 4, p = 3$. Upper part of the table is related to high level value optimizations. The lower part are optimizations obtained after encoding (values exchanged for indexes). There is an important gain on the number of nodes and critical path, due to the large number of constant coefficients produced by the solver.

To give an idea of the hardware resources involved in a decoder, an instance with $n = 8$ and $p = 2$ involving 90 operations on a 15 stage critical path has been reduced to a 14 operation network with an 8 stage critical path after operator fusion. This decoder was translated to a 31 LUT-4 network with a 15 LUT-4 critical path, and a 79 LUT-2 network with a 19 LUT-2 critical path. In the first case the area is 72 LUT-4, in the second case 121 LUT-2. Notice that the area is mostly conditioned by the number of I/Os in the rectangular shape.

The only specific information given to the compiler

was the inference rule on types for GF16 numbers. This was necessary to keep a known optimality of encoding during the index network generation. Beside this, the compiler has handled the whole work efficiently up to place and route for two different FPGA architectures (see figure 3). It is difficult to compare the efficiency of these circuits with other works since most of these works focus on operator complexity[7] rather than on whole program optimization, as it is the case there.

stage	operators	critical path
initial expression	85.08	11.24
constant folding	11.68	7.65
factorization	10.41	7.65
no-op removal	7.42	5.75
operator fusion	4.5	3.62

Table 1. Average numbers of nodes and critical path stages for all 4:3 RAID decoders

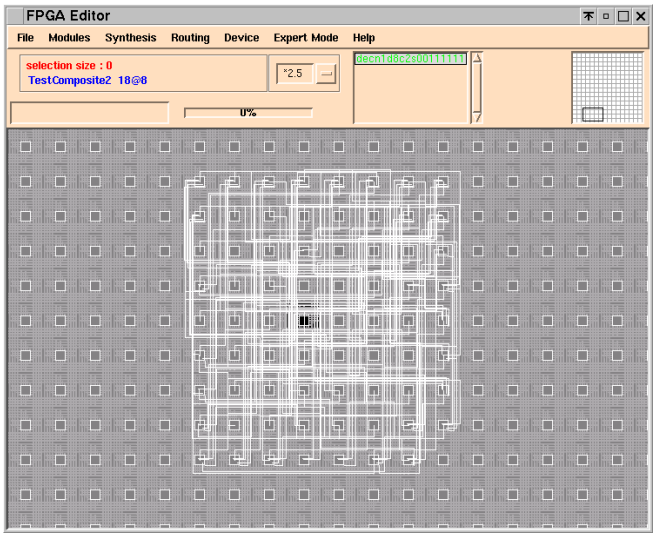

Figure 3. Decoder 8+2 on a Lut-4 architecture

5. Work status and perspectives

Section 1 has presented our project in three parts. RA modeling tools are operational, with a practical implementation on xc6200 and current modeling for the Virtex. There are also investigations on designing and tuning new FPGA architectures with our tools.

The compiler described in this paper is in progress. Procedure call translation is working, but optimization for tests (if), variable dependencies, and class-based type inference are not fully implemented. At the higher level, we are experimenting the compiler on regular architectures. The link with system synthesis can be found in a previous work where a whole application code has been synthesized for an embedded system featuring a smart vision sensor. Code production (C, communications and microcode) was achieved using class-based types[2].

We would like to point out that logic synthesis is not exclusive of architecture synthesis, the type system allowing to detect sparse data path or, at the opposite, opportunities to reuse general purpose operators. Our main conclusion is that this method gives the possibility to create specific logic based on clear algorithm expression. These algorithms are handling the logic exactly needed in the situation, leading to short development times and efficient hardware.

References

[1] V. Betz and J. Rose. Vpr: A new packing, placement and routing tool for fpga research. In *Field Programmable Logic and Application*, LNCS, 1997.

[2] G. Fabregat, G. Leon, O. Le Berre, and B. Pottier. Embedded system modeling and synthesis in oo environments. a smart-sensor case study. In G. Gao and K. Palem, editors, *CASES'99*, Oct 1999.

[3] L. Lemarchand. Parallel performance directed technology mapping for fpga. In *Proceedings of IEEE Southwest Symposium on Mixed-Signal Design, Tucson, USA*, pages 189–194, 1999.

[4] B. Lin, S. Whitcomb, and A. Newton. Symbolic don't care and equivalence in high level synthesis. In I. P. Michel and G. Saucier, editors, *Logic and Architecture Synthesis*. Elsevier, 1991.

[5] L.Lagadec and B.Pottier. Object oriented meta-tools for reconfigurable architectures. In *SPIE, Reconfigurable technology II*, volume 4212, Nov. 2000.

[6] J.-L. Llopis and B. Pottier. Smalltalk blocks revisited, a logic generator for fpgas. In J.-M. Arnold and K. Pocek, editors, *FCCM'96*, Napa, CA, 1996. IEEE press.

[7] C. Paar and M. Rosner. Comparison of arithmetic architectures for reed-solomon decoders in reconfigurable hardware. In *(FCCM'97)*, Napa, CA, 1997. IEEE press.

[8] J. Plank. A tutorial on reed-solomon coding for fault-tolerance in raid-like systems. Technical Report UT-CS-96-332, Department of Computer Science, University of Tennessee, Feb. 1999.

[9] E. Sentovich and al. Sis: A system for sequential cirquit synthesis. Technical Report UCB/ERL M92/41, EECS, Berkeley, May 1992.