A LUT based high level synthesis framework for reconfigurable architectures

Loïc Lagadec, Bernard Pottier,
Oscar Villelas-Guillen
Université de Bretagne Occidentale,
UFR Sciences, 20 av. LeGorgeu, Brest 29285, France

September 18, 2003

Abstract

We describe a method for producing combinational FPGA circuits from symbolic specifications completed by context dependent type declarations. Types represent the set of values that will appear in the circuit input. The program is automatically specialized for the application data domain, shortening development time.

The principle is to represent the whole computation as a network of lookup-tables equivalent to the function calls. When these tables become small enough, they are translated into logic tables and mapped to FPGAs. A compiler is developed according to this method. Results are shown on the practical case of a Reed-Solomon RAID corder-decoder generator.

1 Introduction

1.1 General context

It is a fact that integration technology is providing hardware resources at an exponential rate while the development methods in industry are only progressing at a linear rate. This can be seen as the repetition of a common situation where a mature technical knowledge is providing useful possibilities in excess of current method capabilities. An answer to this situation is to change the development process in order to avoid work repetition and to provide more productivity by secure assembly of standard components.

This situation is also known from computer scientists since it has been encountered earlier in the programming language story[DC90]. The first ages of this story are: (1) symbolic expression of computations (Fortran), (2) structured programs (Algol), (3) modularity, and code abstraction via interfaces and hiding (Modula2, object oriented programming).

Modularity came at the age where efficient engineering of large programs was the main concern, and when the task of programming was overtaken by the organization problems. System development can be considered as a new age for computer architecture design, with hardware description languages needing to be transcended by higher level of descriptions to increase productivity. Companies developing applications have their specific methods for design and production management in which they can represent their products, tools and hardware or software components. The method that ensure the feasibility of a product is leading technical choices and developments. It also change some of the rules in design organization, since most of the application is achieved in a top-down fashion using object models or code generators reaching the functional requirements.

1.2 Reconfigurable architectures

FPGAs are one of the driving forces for integration technology progresses, due to their increasing field of applications. Like software and hardware programming languages, reconfigurable architectures are sensitive to scale mutations. As the chip size is increasing, the characteristics of the
application architecture change, with new needs for structured communications, more efficiency on arithmetic operators, and partial reconfigurability.

The software follows slowly, migrating from HDL to HLL. Preserving the developments and providing a same support for production tools is, a major issue. Reconfigurable architectures can take benefits from the top-down design style of section 1.1, by a high level of specialization in the applications.

1.3 Madeo

Madeo is a medium term project that makes use of open object modeling to provide a portable access to hardware resources and tools on reconfigurable architectures.

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 on a generic model enables sharing of basic tools such as place and route, allocation, circuit edition[LP00]. Mapping a logic description to a particular technology is achieved using generic algorithms from SIS[Sa92], or PPart[Lem99]. Specific atomic resources such as memories, sensors or operators, can be merged with logic, and the framework is extensible.

2. **high level logic compiler.** This compiler produces circuits associated to high level functionalities on a characterization of the above model. Object oriented programming is not restricted to a particular set of operators or types, and then provides the capability to produce primitives for arbitrary arithmetics or symbolic computing.

The compiler handles an intermediate level which is graph of look-up-tables carrying high level values (objects). Then this graph is translated into a logic graph that will be mapped on hardware resources. The translator makes use of values produced in the high level environment what allows to implement a lot of classical optimizations without attaching semantics to operations at the language level.

3. **System and architecture modeling.** The computation architecture in its static or dynamic aspect 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 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, allows to build complex networks of fine or medium grain elements.

The chapter focuses on the logic compiler. Historically, this work has taken ideas from symbolic translation to logic as described in [LWN91] and knowledge on automatic generation of primitives in interpreters[BWB86]. Relation with the object oriented environment has been described in [LP96] with limited synthesis capabilities that are removed in current work. System modeling and program synthesis has been demonstrated on the case study of a smart sensor camera[FLLP99] based on the same specification syntax as the one used in the current work.

The chapter describes the general principles used for specification and logic production, then it gives more details on the transformations that are achieved. An illustration is given with the example of a decoder/decoder family for RAID systems with quantitative results (section 5).

2 A framework for logic synthesis

2.1 Architecture modeling

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

Most FPGAs provide logic functions using 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 phase 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 via a specific grammar [LP00] allowing the description of concrete architectures. Given this description, generic tools operate for technology mapping, placing and routing logic modules. See figure 9 for a view of the generic editor. Circuits such as operators or computing networks are described by programs realizing the geometric assembly of such modules and their connections.

Using this framework, few days of work are sufficient to bring up the set of tools on a new architecture, with the possibility to port application components. On a concrete platform, it is then necessary to build the bit-stream generation software by rewriting the application descriptions to the native tools. Two practical examples are the xc6200 that has a public architecture and has been addressed directly, the Virtex 1 is addressed through the JBits API, and other implementations include industrial prototype architectures.

If behavioral generators are known to offer numerous benefits over HDL synthesis, including ease of specifying a specialized design and the ability to perform partial evaluation [MC98], they generally remain dependent of some libraries of modules appearing as primitives [FM01, BT94]. Our approach draws attention to itself by relying on a generic back-end tool in charge of the modules production. There are no commercial tool nor library involved in the flow.

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, too much emphasis is placed on the local performance of standard arithmetic units in the synthesis tools and also in the specification languages.

A first consequence of this advantage is the restricted range of basic types coming from the capabilities of ALU/FPU or memory address mechanisms. Control structures strictly oriented toward sequentiality are another aspect that can be criticized. As an example, programming for multimedia processor accelerators remains procedural in spite of all the experience available from the domain of data parallel languages. Hardware description languages have rich descriptive capabilities, however the necessity to use libraries has led the language designers to restrict their primitives to a level similar to C.

![Figure 1: The modules can be either flat or hierarchical; the modules can be composed in order to produce pipelines or can be instantiated during architecture synthesis.](image_url)

Our aim is to produce a more flexible specification level with direct and efficient coupling to logic. This implies allowing easy creation of specific arithmetics representing the algorithm needs, letting the compilers automatically tune data width, and modeling computations based on well understood object classes. The expected effect is an easy production of dedicated support for processes that need a high level of availability, or could waste processor resources in an integrated system. To reach this goal, we use specifications with symbolic and functional characteristics, jointly with separate definition of data on which the program is to operate.
Sequential computations can be structured in various ways by splitting programs on register transfers, either explicitly in the case of an architecture description, or implicitly during the compilation. Figure 1 shows these two aspects, with a circuit module assembled in a pipeline and in a data-path. In the case of simple control loops or state machines, high level variables can be used to retain the initial state with known values, the compiler retrieving progressively the other states by enumeration [LP96]. Figure 2 shows a diagram where registers are provided to hold state values associated to high level variables that could be instance variables in an object.

![Diagram](image)

Figure 2: State machines can be obtained by methods operating on private variables having known initial values.

At this stage, 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 there. The input language is Smalltalk-80, variant VisualWorks, also used to build the tools and to describe the application architectures.

### 2.3 Execution model

The execution model targeted by the compiler is currently 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. The resulting circuit is highly dependent from the data it is intended to process.

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 a strong algorithmic meaning. Among the different possibilities offered for practical execution, there are cascaded hash table accesses, and use of general purpose arithmetic units where they are detected to fit.

Translation to FPGAs need binary representation for objects, as shown figure 6. This is achieved in two ways, by using a specific encoding known to be efficient, or by exchanging object values appearing in the input and output for indexes in the enumeration of values. Figure 3 shows a fan-in case with an aggregation of indexes in the input of function $h()$. Basically the low level representation of a node such as $h()$ is a Programmable Logic Array (PLA) having in its input the Cartesian product of the set of incoming indexes ($fout \times goat$), and in its output the set of indexes for downstream.

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 interfacing requirements, it will be needed to insert nodes in the input and output of the network to handle the exchanges between values and indexes.

3. logic synthesis tool capabilities are limited to medium grain problems. To allow compilation to FPGAs, algorithms must 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.

4. 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. The number of values carried by edges is decreasing either in the hierarchy structure or in a graph flow. There is no possible divergence and the efficiency of an algorithm can be stated to be its ability to quickly decrease the data amplitude on which the logic complexity depends.

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 programming 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 types 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

In a first stage, 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 for a LUT-n FPGA architecture. As each node can be directly synthesized, we have a flat expression in opposition with hierarchical expressions that will need additional compilation contexts for some of the function calls.

As a Smalltalk development environment is used, there were an obvious interest to use this language syntax for ’programs’ targeting FPGAs. Immediate benefits are the reuse of the standard compiler front-end and use of the existing classes.

1. Building the value network

The first compilation stage consists in building an acyclic flat graph which nodes are lookup tables based on objects 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, building a table for each node having defined inputs.
During this transformation care must be taken of dependencies in variable used in fan-out to fan-in subsets. As an example, the composition $h(f(x, y), g(x, z))$ has a smaller output than $h(f(x, y), g(t, z))$ because of the dependency on variable $x$. A number of inputs in the fan-in node $h$ and upstream are not useful and can be deleted by constraining the Cartesian products in $f$ and $g$ tables.
Lot of conditional computations fall in this case.

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 stronger properties from the message executions. It is time to apply 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 subsets with particular topologies. As an example, linear cascade of tables can be collapsed 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 also take into account the size of LUT memories in the target architecture. The result is a 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 framework, producing a hardware module for further system handling and binding.

### 3.2 Hierarchical aspects
In section 3.1 we supposed that the program can to be directly synthesized 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 than their root function, for every node showing an excessive complexity related to synthesis.

When the compiler reaches a condition where logic tools will be inefficient, it creates a new compilation context and process recursively the call. The context will return a structured logic description that will be installed as part of the current level production.
The technical form of a logic description associated to a compiled program is a hierarchical BLIF (Berkeley Logic Interchange Format) description that can be partially flattened for further logic optimization, and partially placed under control of a floor planner. In this case each developed function call has its corresponding circuit component assembled in the global hierarchy.

A more speculative compiler built-in function is type partitioning. When a data set appears to be too much large, the compiler can divide the type in order to reach a grain suitable with synthesis. 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, type-based rules must be managed by the compiler to prevent new type generation in node outputs.

4 Example of small FP multipliers

To explain the programming interface related to the compiler work, let us comment the case of a multiplier operating on small floating point numbers. The numbers are represented as a sign, an exponent and a fractional part. The multiplication is described in a class supporting secondary methods for adding exponents, multiplying fractional parts, and normalising the result. The algorithm for the multiplication is taken from [PHB8]. It is developed and verified in the software environment before synthesis. Below is a Smalltalk-80 code for this.

```smalltalk
sign: signA significand: significandA exponent: exponentA
sign: signB significand: significandB exponent: exponentB
| sign exp mant shift |
sign := self computeSignFor: signA and: signB.
mant := self computeSignificandsFor: significandA and: significandB.
shift := self computeMantOffset: mant.
exp := (self computeExponentFor: exponentA and: exponentB)
+ shift.
mant := self normalizeMant: mant with: shift.
"Array with: sign with: mant with: exp"
```

To produce logic for this program, it is necessary to provide a characterization of the objects present in the various fields as boolean or integer intervals, thus tailoring the arithmetic. Depending on the data amplitude of the inputs of each node, the compiler either develops them hierarchically or not. In the first case a new graph is produced which is reachable from the node. In the second case a table, equivalent to the node, will directly be computed and associated to the node.

![Diagram](image)

Figure 5: Data-flow graph for the multiplication after removing temporaries and complexity evaluation.
Figure 5 shows the multiplier directed acyclic graph (DAG). Light grey boxes are associated to hierarchical nodes expanding their own DAG, dark grey boxes are hierarchical nodes flattened due to their small size. Medium grey boxes are associated to terminal primitives translated directly into lookup tables.

Figure 6 shows two different aspects of a particular node. The first table is holding high level objects, such as rational numbers and the right view displays the equivalent index table with symbol numbers.

Figure 6: Values and index lookup tables

Figure 7 displays a 6 node hierarchical circuit and a flat globally optimized circuit for the multiplier. In the first case, there was no attempt to compact the design nor to achieve a flat logic optimization, to help readability. The technology is LUT4 and the circuits have respectively 161 and 138 nodes. The second circuit had 24 nets unrouted due to a choice of low routability in the FPGA architecture.

Figure 7: Hierarchical and flat circuit for the multiplier

5 A RAID error correction case study

The procedure for flat expressions is illustrated by the example of a RAID system correction system. In RAID system redundancy for error correction is kept using Reed-Solomon (RS) coding over Galois Fields. Here we have made the choice of a field with $2^k$ elements. We will concentrate on the implementation of the encoder/decoder parts. We will talk of $n : m$ RS indicating a RS schema where $m$ checksum disks are used as redundancy for $n$ disks of data.
Basically, the encoder part will take the n streams to be stored in the data disks to generate the m streams to be stored in the redundant disks. This makes for a unique reconfiguration for a given n : m schema.

On the other hand, the decoder part will take the streams stored on disks and return the original data streams. When all stored streams are available, the decoders simply return the data from the data disks. However, if one (or up to m) disks fails, the original data may need to be reconstructed from the checksums.

The use of FPGA for encoding/decoding seems appropriate mainly for two reasons:

- There may be little or no performance loss due to error correction. The cost is paid only when a disk failure happens (transition from disk working to disk non-working).
- It will provide the system with added flexibility. The ability to mutate the circuits will allow the same hardware to be used for different failure schemas.

We have based our case study in the encoding/decoding in a word-by-word basis. This means that, as we are talking of Galois Fields 2^8, we obtain circuits that take data from the streams in groups of 4 bits. As the operations for different words are independent, it is possible to replicate the circuits to work on multiple data words simultaneously (at the expense of logic space) to meet desired performance.

### 5.1 Reed-Solomon coding

Reed-Solomon (RS) coding allows correction of up to m errors using m checksum words. For a system with n data words allowing error recovery for m failures, a total of n + m words should be stored.

The basic idea of RS coding is to build a system with n + m rows and n columns. All rows are built to be independent. Recovery from up to m errors is possible as we could always take the available words and build a system that is solvable. Solving that system by any technique (like Gaussian Elimination) will provide the original data words.

**Encoding:** at this point we need to build m additional independent equations. To achieve that, we will use the Vandermonde matrix and compute its associated independent terms that will serve as checksums. This can be done by performing the following operation (where d_1 ... d_n are the data words to encode and c_1 ... c_m its associated checksums):

\[
\begin{bmatrix}
1 & 1 & 1 & \cdots & 1 \\
1 & 2 & 3 & \cdots & n \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
1 & 2^{m-1} & 3^{m-1} & \cdots & n^{m-1}
\end{bmatrix}
\begin{bmatrix}
d_1 \\
d_2 \\
\vdots \\
d_n
\end{bmatrix} =
\begin{bmatrix}
c_1 \\
c_2 \\
\vdots \\
c_m
\end{bmatrix}
\]  

(1)

**Decoding:** when retrieving data we know that the following equation must apply:

\[
\begin{bmatrix}
1 & 0 & 0 & \cdots & 0 \\
0 & 1 & 0 & \cdots & 0 \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
1 & 2 & 3 & \cdots & n \\
1 & 2^{m-1} & 3^{m-1} & \cdots & n^{m-1}
\end{bmatrix}
\begin{bmatrix}
d_1 \\
d_2 \\
\vdots \\
d_n
\end{bmatrix} =
\begin{bmatrix}
d_1 \\
d_2 \\
\vdots \\
d_n
\end{bmatrix}
\]  

(2)

So in case of an error on a word of data, we can compute its value by solving a system involving n rows of the equation. This will be possible as long as we have n valid values in the independent term vector, that is, there are less than m errors.

**Arithmetic over Galois Fields** is used as the algebra needed to solve the system, as is closed over a field of finite size.

For a more detailed description of RS coding using arithmetic over Galois Fields, the reader may refer to Plank's tutorial [Pla99], in which we have based our work. Other interesting bibliography comes from C. Paar et al., as example [PR97] provides information on GF operator complexity and implementation on FPGAs.
5.2 Encoder/decoder specification

Our objective is to obtain configurations for the encoding/decoding using RS over Galois Fields. We are targeting a system that has a reconfigurable part to do both, encoding and decoding. We will need a configuration for the encoder and several configurations for decoding, one for each working condition (set of words missing). This can be applicable to RAID systems, where the working condition (disk failure) can be considered rare, and the cost of reconfiguration in such a case will have little impact. See figure 8.

![Diagram of encoder/decoder specification](image)

Figure 8: Encoding $n$ data slices to $n + m$ disks with one redundant encoder shown. Decoding from $n + m$ disks to $n$ data slices with one decoder for a broken disk shown.

The specification has been developed in three steps:

**Reed Solomon specification.** The specification generates the equation 2. This is specified as a Smalltalk class that has methods for encoding/decoding data. The class has been built in order to fix the number of data and checksum words at instance creation, so it can be used for any encoding size. The specification has been tested by exhaustively performing error correction using conventional arithmetic.

**Development of Galois Field arithmetic class.** A Galois Field $2^4$ for Smalltalk has been developed (GF16). The operations implemented are those needed in our problem, following the guidelines shown in [Fla98]. Using Smalltalk’s powerful polymorphic nature, we can apply the Reed Solomon class using the new arithmetic. We have performed an exhaustive test of the RS coding using the Galois Field arithmetic in order to test correctness.

**Extraction of error correction expressions.** Building an arithmetic like class that records the operations performed, we can build the expressions for the encoding of checksums, as well as for the decoders in a given working condition. Those expressions can be packed into method code that will be compiled to build the configurations.

As a note, the above specification took a few hours to be done, with no initial experience on Reed-Solomon coding.

5.3 Expression compilation

From the specification, we can take the expressions for the encoders/decoders and compile them to logic. The steps below show the most important effect of the implementation as handled in our framework.

**Type inference.** After building the DAG, all edges are typed. In this case, the inputs are data and checksums words, all of type GF16.

**Constant folding.** Due to the generated nature of our expressions, there are plenty of operations over constants. Those are all removed in this step. Also operations yielding a constant result (like multiplication by 0) are removed.
**Dead code removal.** As a result of removing multiplication by 0 operations, and due to the nature of automatically generated expressions, it is possible that there are expressions whose result is never used, so they are removed.

**Code factorization.** That is, common sub-expression elimination.

**Operator LUTification.** This step is the first one towards architecture binding. It transforms the symbolic operations into look up tables suitable for logic synthesis.

**No-op removal.** Unary operators whose output is equivalent to its input are removed.

**Operator fusion.** Unary operators are removed by fusing them with its producer/consumer operators. We assume that this will provide a better implementation.

**Circuit production.** Several circuits have been produced to collect practical information of the results.

### 5.4 Encoders/decoders statistics

Tables 1 and 2 display respectively statistics for the 3 necessary encoders and all the possible decoders for disk failures. The tables has columns showing the decrease in the number of GF16 operators, average number of inputs per operator, and the critical path in the network of operators as a result of each compiler operation. The meaning of rows is the observed value after each optimization operation as described in section 5.3.

Correctness has been checked at the logic level by selecting random random inputs and verifying the output after logic synthesis using SIS simulate command.

<table>
<thead>
<tr>
<th>Compiler operation</th>
<th>operators</th>
<th>average input</th>
<th>critical path</th>
</tr>
</thead>
<tbody>
<tr>
<td>type inference</td>
<td>12</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>constant folding</td>
<td>8</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>dead-code removal</td>
<td>8</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>code factorization</td>
<td>8</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>operator to LUT</td>
<td>8</td>
<td>1.375</td>
<td>5</td>
</tr>
<tr>
<td>no-op removal</td>
<td>5</td>
<td>1.67</td>
<td>3.67</td>
</tr>
<tr>
<td>operator fusion</td>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 1: statistics for encoders-RS4:3.

<table>
<thead>
<tr>
<th>Compiler operation</th>
<th>operators</th>
<th>average input</th>
<th>critical path</th>
</tr>
</thead>
<tbody>
<tr>
<td>type inference</td>
<td>85.08</td>
<td>2</td>
<td>11.24</td>
</tr>
<tr>
<td>constant folding</td>
<td>11.68</td>
<td>2</td>
<td>7.65</td>
</tr>
<tr>
<td>dead-code removal</td>
<td>11.68</td>
<td>2</td>
<td>7.65</td>
</tr>
<tr>
<td>code factorization</td>
<td>10.41</td>
<td>2</td>
<td>7.65</td>
</tr>
<tr>
<td>operator to LUTs</td>
<td>10.41</td>
<td>1.43</td>
<td>7.65</td>
</tr>
<tr>
<td>no-op removal</td>
<td>7.42</td>
<td>1.65</td>
<td>5.75</td>
</tr>
<tr>
<td>operator fusion</td>
<td>4.5</td>
<td>2</td>
<td>3.625</td>
</tr>
</tbody>
</table>

Table 2: statistics for decoders-RS4:3.

### 5.5 Specific 8:2 case with circuit generation

This time, the case of a RAID system with 8 data disks and 2 redundant disks is considered.

These circuits has been optimized and mapped to 2 different architectures having 2-LUT and 4-LUT cells. The table 5.5 shows the compared characteristics of the encoder and decoder on
<table>
<thead>
<tr>
<th></th>
<th>Encoder</th>
<th></th>
<th>Decoder</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LUT 2</td>
<td>LUT 4</td>
<td>LUT 2</td>
<td>LUT 4</td>
</tr>
<tr>
<td>Area (1)</td>
<td>90</td>
<td>56</td>
<td>121</td>
<td>72</td>
</tr>
<tr>
<td>Cells Used (2)</td>
<td>85</td>
<td>53</td>
<td>119</td>
<td>71</td>
</tr>
<tr>
<td>Input cells (3)</td>
<td>32</td>
<td>32</td>
<td>40</td>
<td>40</td>
</tr>
<tr>
<td>Internal cells (4)</td>
<td>53</td>
<td>21</td>
<td>79</td>
<td>31</td>
</tr>
<tr>
<td>Input average (5)</td>
<td>1.62</td>
<td>2.04</td>
<td>1.67</td>
<td>2.23</td>
</tr>
<tr>
<td>Gates Input average (6)</td>
<td>2.0</td>
<td>3.62</td>
<td>2.0</td>
<td>3.81</td>
</tr>
<tr>
<td>Routing Cost (7)</td>
<td>1095</td>
<td>640</td>
<td>1839</td>
<td>1049</td>
</tr>
<tr>
<td>Critical Path (8)</td>
<td>18</td>
<td>14</td>
<td>19</td>
<td>15</td>
</tr>
<tr>
<td>CPU Time (9)</td>
<td>43.14</td>
<td>20.34</td>
<td>98.70</td>
<td>34.89</td>
</tr>
<tr>
<td>Max struct. area (10)</td>
<td>128</td>
<td>88</td>
<td>208</td>
<td>176</td>
</tr>
<tr>
<td>Cells used (11)</td>
<td>109</td>
<td>85</td>
<td>181</td>
<td>170</td>
</tr>
<tr>
<td>Internal cells (12)</td>
<td>53</td>
<td>29</td>
<td>79</td>
<td>58</td>
</tr>
</tbody>
</table>

Table 3: Results from place and route on 2 architectures

These architectures. Each one has a routing channel of size 8 inside the cell patterns, providing a first run success. Some parameters are extracted that can be used at a higher level, as an example for system management of the reconfigurable logic resources, or for making choices in the compiler generation code strategy. Notice that at the end of optimization on LUT, it is easy to generate processor code and table contents equivalent to the network of reconfigurable logic cells.

The table has two parts for post-assembly optimized logic and simple structured assembly as it is used for floor planning. The presented characteristics are:

1. total circuit area in number of cell patterns,
2. gates used in this area,
3. number of inputs for the circuit,
4. effective number of cells used to implement logic,
5. the average of used inputs in module cells including the border
6. the same measure for cells in (4)

Notice that (3)+(4)=(2), with (4) being low. The circuit is I/O dominated. Gates used in (3) disappear when the module is connected to other architecture element.

Routing cost (7) is an estimation on the number of resources allocated for connections. Critical path (8) is the maximum number of cells and other resources allocated in the circuit between an input and an output, with unitary costs. CPU time (9) provides an idea of the delay to place an route the circuit on a PC/750Mhz with the Visualworks environment running on Linux. Figure 9 show a view on a decoder as generated by the tools.

(10) is the maximum area occupied by the assembly of elementary modules without post-assembly optimization, and without the use of the floor planner. (11) is the maximum number of cells used in this area, and (12) is the number of cells used to implement logic in the area. (12) is similar to (4). A good measure of the post-assembly optimization is the respective 46% and 27% logic decreases in the cases of the decoder and encoder. The use of the floor planner will bring (10) and (11) closer to (1) and (2).

6 Conclusion

Madeo tools for reconfigurable architecture modeling are operational, with practical implementations on commercial FPGAs and prospective FPGA prototype architectures. This demonstrates the feasibility of a FPGA hardware/software interface standardisation in a way similar to microprocessors and HLL compilers.

The described compiler is still a work in progress. It is possible to produce an optimized hierarchical logic description suitable for technology mapping, place and route. Dependencies in fan-out to fan-in subgraph, rejecting unused values from the branches of conditional statements,
and recursive specifications are handled. The main effect of this compiler is to achieve optimizations mostly at high level, removing a considerable load on the logic mapping algorithms. The strength in optimization comes from the fine knowledge on values being processed, that allows to simplify computations either at high level or logic level. The underlying execution model is understandable for the programmer who has a direct feedback for the algorithms.

Related to productivity in developments, our method also gives the possibility to create specific logic based on concise behavioral specifications that are reusable in a variety of situations on different kind of data. The compiler will be able to allocate medium grain resources such as memories or arithmetic operators, based on the architectural model and types propagated inside the computation graph.

Finally, we find the object oriented approach very promising either for architecture management and high level synthesis. This encourages to follow in this direction for higher scope work in the context of systems on chips.

acknowledgment
This work has been supported by grants from the MEFI/STSI and STMicroelectronics. The authors would like to thank O. Colavin, T. Ben-Ismail and V. George for all they are sharing. Many thanks to all the students involved in the various aspects of the developments.

References


