Placing, Routing and Editing Virtual FPGAs

L. Lagadec[1], D. Lavenier[2], E. Fabiani[2], B. Pottier[1]

[1]Université de Bretagne Occidentale - Brest
[2]IRISA / CNRS - Rennes
France
{lagadec,pottier}@univ-brest.fr - {lavenier,efabiani}@irisa.fr

Abstract. This paper presents the benefits of using a generic FPGA tool set developed at the university of Brest for programming virtual FPGA structures. From a high level FPGA description, the basic FPGA tools such a placer, a router or an editor are automatically generated. The FPGA description is not constrained by any model, so that abstract FPGA structures, such as virtual FPGAs, can directly exploit the tool set as their basic programming tools.

1 Introduction

Reconfigurable Computing (RC) aims to use the flexibility of the configurable logic proposed by FPGA components to enhance computation performance. The common idea is that RC fills the gap between a general purpose Von Neumann architecture (microprocessor) and a highly specific full custom architecture (ASIC: Application Specific Integrated Circuit) by *programming* an appropriate architecture. Microprocessor performance is limited by the sequential behavior while ASICs suffer the definitive silicon implementation. FPGA components appear as a tradeoff between these two alternatives: the same physical support can be re-programmed (or reconfigured) to support any architecture. Hence, reconfigurable computing claims to add both the flexibility of programming machines and the speed of specific architectures.

The reality is not so obvious. First, an FPGA architecture is much slower than its ASIC counterpart. This is mainly due to its programmable nature, which requires signals to pass along many programmable electronic switches, compared to a direct silicon implementation. Second, the synthesis of a specific architecture onto a FPGA component is still a long and error-prone process that is far to be completely automated. The architecture to implement is often specified in VHDL, and requires a lot of simulation steps to be validated. In addition, a design targeted for a particular reconfigurable platform is usually impossible to re-use: the portability between different reconfigurable platforms is not ensured due to the absence of a *programming model*. Thus, if it is theoreticaly possible to implement any kind of architecture, the non-portability, the time and the efforts devoted to implement an architecture tend to weaken the notion of flexibility.

One way for increasing this flexibility is to define a **virtual FPGA structure**. In that case, an architecture is not defined relative to a specific FPGA component, but relative to a virtual FPGA structure which will be implemented across the different existing components and hopefully across future generations of FPGA components.

As with any virtual machine, such as the Java Virtual Machine, we must deal with the loss of performance introduced by the virtual layer. In our case a virtual structure will have a limited amount of resources (virtual logic blocks) and the clock speed will be slowed down compared to a real FPGA component. To keep performance reasonable we propose to **specialize** the virtual structure towards a specific field of applications. In that case, the slowness may be partially compensated by integrating into the virtual level fast specific functions inherent to the application domain.

If designing a range of specialized virtual FPGA structures targeted to various application domains will guarantee a minimum of performance, one has to wonder how these architectures will be programmed. More precisely, the problem is to develop appropriate utility programming tools, such as the placer or the router, for each specialized FPGA. Of course, we cannot imagine rewriting such tools from scratch each time a new FPGA structure is proposed.

This paper addresses this problem. It presents a generic FPGA tool set which, from a high level specification of a FPGA structure, automatically provides the basic tools required to configure it. This generic tool has not been developed especially for programming virtual specialized FPGA structures. Any kind of FPGA structures can take advantage of it. The purpose, here, is to focus on the benefit of using this approach in the context of virtual FPGAs.

The rest of the paper is organized as follows: Section 2 briefly describes the concept of specialized virtual FPGA structures. Section 3 presents the generic FPGA programming tool set. Section 4 exemplifies a virtual FPGA structure dedicated to linear systolic arrays and shows how the tool set works. Section 5 concludes this paper.

2 Specialized Virtual FPGA Structures

The concept of virtual FPGA structure is identical to the concept of virtual machine. Our view of a virtual FPGA is different of the definition given in [6] where it is seen as an extension of the physical FPGA device: the applications have a virtual view of the FPGA that is mapped on the available physical device by the operating system, in a way similar to virtual memory. In other words, the FPGA is virtualized by multiplexing its physical components. The same idea [10] has also been followed for increasing the number of FPGA I/O pins, leading to the concept of virtual wires. In both cases the goal is to extend the capabilities of the FPGA devices.

As opposed, we see a virtual FPGA as a simplified version of a physical FPGA. The idea is not to enhance the FPGA features, in terms of a higher number of logic blocks or a better frequency, but to set a stable and portable structure like a virtual machine. Transposed to the hardware domain, a virtual FPGA is a regular pattern of virtual configurable logic blocks, each of them

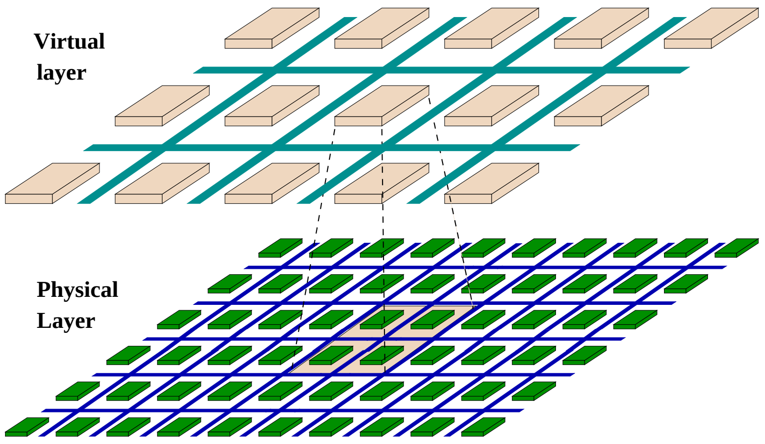


Fig. 1. Physical layer and Virtual layer: A virtual configurable logic block is made of several physical ones.

made up of several physical ones (cf figure 1). Compared to a virtual machine, it has nearly the same advantages and drawbacks. On the positive side it provides:

– A **portable** structure: Architectures targeted to a virtual FPGA structure can be implemented on any physical FPGA supporting the virtual layer.
– An **open** structure: The details of the structure are not constrained by confidentiality. They can be made freely available, allowing groups of people to develop and test their own tools.
– A way to investigate **new** FPGA structures: Today, available FPGA components are not well suited for reconfigurable computing. They are mainly designed to meet the market requirement (ASIC substitution) and don't support advanced functionalities such as fast dynamic or partial reconfiguration.

On the negative side we can point out:

– the reduction of available resources: a virtual logic block will be made of several physical ones, leading probably to a reduction of an order of magnitude in terms of useful hardware. This is may be the price to pay for portability.
– the speed: the architectures mapped onto FPGA components are renowned to be slow compared to an ASIC implementation. Adding an intermediate layer will further slow them down!
– the absence of programming tools: the tools provided by the FPGA vendors are useless for mapping an architecture on the virtual layer. New tools are required and must be developed.

The solution we propose to remedy the speed problem is to specialize virtual FPGA structures to target specific applications or well-defined models of architectures. In this case, the low speed may be partially compensated by integrating into the virtual level fast specific functions inherent to the architecture or the application domain. The idea of specialization is illustrated in section 4.

However, from our point of view, the absence of programming tools is a much more serious barrier to the concept of virtual FPGA. The main reason is that developing such tools is a long and complex task. There are no development tools, such as retargetable C compilers [8] available for microprocessors. Everything must be nearly redesigned from scratch. The next section presents the generic FPGA programming tool set developed at the University of Brest as a solution to suppress this stage.

3 The Generic FPGA Programming Tool Set

The software we have developed aims to provide very quickly a set of essential tools for programming a FPGA structure (virtual or not). The first step is to specify the FPGA structure. It is then compiled and a tool set (placer-router, editor, estimator, ...) is immediately generated. This section focuses first on how a FPGA structure is specified, then the different available tools are described. We end by giving a word about the implementation.

3.1 Specification of a FPGA structure

An FPGA structure is always a hierarchical organization of patterns replicated in a regular way. Pattern are an assembly of cells and routing resources. Cells may contain one or several look-up tables (LUT), registers, and specific functions such as, for example, carry propagation logic. Routing includes wires and their interconnection mechanism (transistors, tri-states, multiplexers).

An FPGA structure is specified using a grammar style. For example, the simplified FPGA structure shown in figure 2 is simply specified as follows:

```
   (ARRAY ((DOMAIN 0 99 0 99)
   (COMPOSITE (WIRE wireH) (WIRE wireV)
   (SWITCH switch (Type disjoint))
   (FUNCTION funct (INPUTS in0 in1 in2 in3) (OUTPUTS out))
   (CONNECT
      'funct out connectTo: wireH'
      'wireH connectTo: func in0'
      'wireH connectTo: func in1'
      'wireV connectTo: func in2'
      'wireV connectTo: func in3'
      'wireH connectTo: switch east'
      'wireH connectTo: (self relativeAt: 1@0) switch west'
      'wireV connectTo: switch south'
      'wireV connectTo: (self relativeAt: 0@-1) switch north')
))
```

It specifies an array of 100 × 100 cells. A cell is composed of 2 wires (wireH and wireV), one switch and one 4-input logic function. All the possible connections are specified with the keyword `connectTo`.

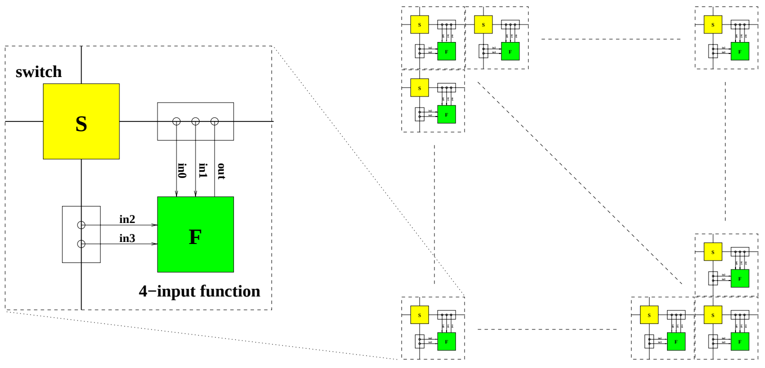


Fig. 2. FPGA structure example. It is a 2D array of identical cells. A cell is composed of one switch and one 4-input boolean function.

A specification may contain different kinds of cells. In that case, the abutment of different cells is resolved using a specific mechanism, called *PortMapper*. It describes the signal interconnection at the cell borders and specifies how the routing is managed. Cells located on the borders of an array are managed by adding specific resources to the elements rather than by defining new kinds of elements.

An important specification possibility is that an FPGA structure can be totally or partially parameterized. It allows the designer of the reconfigurable structure to tune elements such as the granularity of the logic functions, the number of routing resources, etc. This is particularly useful during the definition step for guiding the designer towards the best choices.

3.2 The Tool Set

Once the FPGA structure has been specified, a basic set of tools is available for programming this structure. The next subsections detail the different tools we provide.

Input Format The logic to implement in the FPGA structure must be described as a netlist of logic functions. There must be a direct relation between the available resources and the netlist. For instance, a 5-input function is not allowed in the netlist if the FPGA structure accepts only 4-input functions. The LUT partitioning has to be done before. We currently accept the Berkeley Logic Interchange Format (BLIF) for working environment convenience, but the EDIF format will be the standard way to input the designs in a future version.

Place-and-route The placer relies on a simulated annealing algorithm. All nodes (logic functions) are placed randomly before the annealing starts. At each step, pairs of nodes are swapped. The global interconnection cost is evaluated. In case it is higher than the previous one, all the swapped nodes are discard even if local costs are lower. This new placement is considered for the next iteration. Bad moves can be accepted with a decreasing probability to prevent the algorithm from being trapped into a local minimum. The router starts once the placement is achieved, so that any pre-placed solution or partially placed and routed circuit can be considered. The router is a PathFinder-like router, owning a negotiated scheme for resolving congestion over the routing resources. The router iterates until no congestion remains. All resources have an associated cost depending on the congestion of the resource. Each iteration detects all the signals sharing a congested resource to be ripped up and rerouted. This mechanism forces the signals to use unnecessarily congested resources, as these resources make the cost of the route prohibitive relative to alternative routes.

Regular Editing The placer router is well suited for random logic. However, when the objective is to achieve high performance, circuits must be structured as regular assemblies of patterns. To specify regular circuits, a higher level tool than the placer-router is needed. The regular editor provides a way to replicate modules within the FPGA structure. The layout can be parameterized by the size of modules, and geometric positions can be either absolute or relative. The regular editor takes as input a structural description. This description is a 1D or 2D array of modules (or BLIF descriptions).

Floorplanning The editor process no optimization over the assembly of modules. This leads to acceptable results only if the modules are of comparable size. The floorplanner replaces the common behavior of the editor. The placement of the modules is processed under some optimization criteria (global area, routing cost, etc.). A floorplanner applies a divide and conquer strategy when placing and routing large circuits: they can be partitioned, either at a logic level or at an application level and split into several sub-circuits. Then, these smaller circuits are placed and routed. The floorplanner recomposes the global circuit from the produced modules.

Estimators The quality of the place-and-route stage must be analyzed, with respect to several criteria. The technology mapping stage results in a set of nodes and nets to be implemented that depend on the logic granularity. The placer processes an annealing schedule over the location of the nodes based on a cost function and a bounding box. The algorithm is parametrized to enable different quality/cpu time trade-offs. A timing analyser is being added. Frequency performance estimation of the circuits will be then possible. An important point is that every resource owns a private set of parameters, so that the virtual layer can be *tuned* on demand by extracting useful information from the physical layer.

3.3 Implementation

The Generic FPGA Programming Tool Set is developed in the Visualworks object-oriented programming environment, using SmallTalk as programming language. An FPGA structure is then represented as a composition of classes we can express at four distinct levels:

– The first level is called *the abstract model.* It represents all the different kinds of elements that can be found inside a FPGA structure: routing, registers, logic blocks, organization, etc. Each of these elements is represented by an abstract class[1] which describes its properties and its behavior.
– The second level is called *the concrete model.* It represents the specific elements of an FPGA structure: CLBs, patterns, etc. These elements are modeled through a set of concrete classes.
– The third level is produced by instantiation of the classes of the concrete model. This model, called *the architecture*, is a copy of the FPGA structure the concrete model describes. This level is the one that the tool set manipulates.
– The last level relies on the abstract model to produce the concrete model. It is instantiated by compilation of the FPGA structure specification. It permits fast architectural exploration as parameters (from the specification) can be quickly modified, allowing the concrete model to be rebuilt from new values.

Each tool manipulates the elements of the FPGA structure through its software interface (API). This API is described within the abstract model, so that any element of an *architecture* inherits from and conforms to it. As a consequence, the tools manipulate only the abstract level. They are decoupled from the model so that the model can be upgraded with few or no changes to the tools. Similarly, new tools can be added which run over existing models.

4 Example: Virtual Structure for Linear Systolic Arrays

We illustrate the functionalities of the generic FPGA programming tool set by defining a virtual FPGA structure well suited for implementing linear systolic arrays.

4.1 Architecture

Figure 3 represents the virtual FPGA structure we want to implement. It is a linear array of two-input N-bit cells. A cell can be configurated as a logic or arithmetic function. The two inputs can be connected to neighboring cells ranging from 7 to the left to 7 to the right. To program a systolic architecture on this virtual FPGA structure, one has to place-and-route a systolic processor and to replicate it over the whole structure. Of course, the router must ensure that the routing of a single processor is suitable for replication.

A 8-bit cell has been designed for experimentation purposes and synthesized for the Xilinx Virtex family. In addition to all the basic logic functions, a cell can perform high level arithmetic operations such addition/subtraction or min/max operations. A cell includes all the required configuration mechanism and fits into 65 Virtex slices. A Virtex-1000 component (filled at 80%) can then house 150

[1] An abstract class owns no instances, and acts as a template for subclass production.

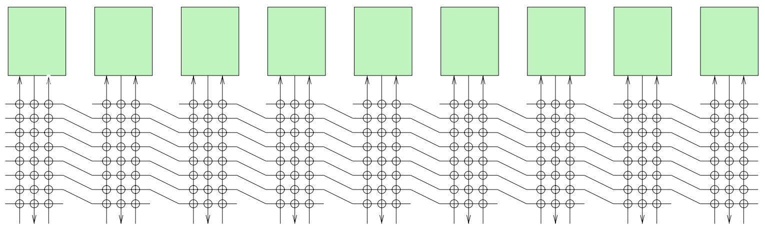


Fig. 3. Virtual Linear Systolic Array: it is composed of a linear array of 2-input N-bit configurable operators. The output of one configurable operator can reach the input of 14 adjacent operators (7 to its left and 7 to its right)

such configurable 8-bit operators. By extension, the Virtex-II XCV2-10K, for example, will be able to fit 800 8-bit configurable operators.

4.2 Using the Tool Set

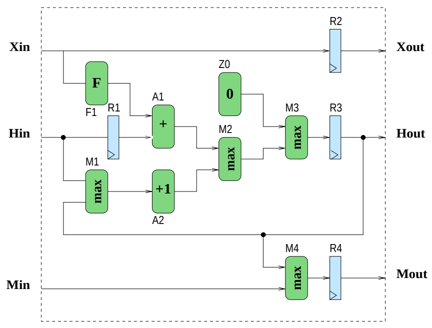
To highlight the tool set functionalities let us take a systolic architecture derived from a real application: the comparison of DNA sequences. The purpose, here, is not to describe how a systolic array can implement this task (details of implementation can be found in [7]) but to show how such an architecture can be easily implemented.

Figure 4 represents a systolic DNA processor and its associated netlist. It is composed of a few 8-bit operators. Each line of the body of the cell represents a zero-, a one- or a 2-input function that a configurable operator can house. The process for implementing an array is first to place and route one processor, then to replicate it over the virtual structure.

A virtual structure representing a linear array of 800 8-bit configurable operators has been specified. Then we run the place-and-route tool giving as input the BLIF netlist corresponding to one processor. It takes only 3 seconds on a standard PC to place-and-route one processor. Then we replicate it 88 times to fill up the virtual array. It just takes a fraction of second to perform this last task.

This first experiment highlights the possibilities of the FPGA tool set as a good candidate for virtual FPGA structures. First it confirms that specifying a new FPGA structure is fast. It takes less than a couple of hours to write down the specification and half a day for tuning the graphical editor (cf figure 5) parameter. Although this FPGA structure is quite simple, it gives an idea of the time one can expect to spend for getting the minimum basic set of tools. It is also worth mentioning that once the general structure is set, further slight modifications can be performed very quickly.

Second, a high level specification of a FPGA structure permits some abstraction. In our case, the datapath width of the cells is not stated. Neither is the connection width between the processors. We just specify that an operator connects its neighbors in an interval ranging from 7 operators to the left to 7



```
CELL  (Xin, Hin, Min, Xout, Hout, Mout)
    FCT    F1 (Xin, F1.out)
    REG    R1 (Hin, R1.out)
    REG    R2 (Xin, Xout)
    ADD    A1 (R1.out, F1.out, A1.out)
    INC    A2 (M1.out, A2.out)
    MAX    M1 (Hin, Hout, M1.out)
    MAX    M2 (A1.out, A2.out, M2.out)
    CTE    Z0 (Z0.out)
    RMAX M3 (Z0.out, M2.out, Hout)
    RMAX M4 (Min, Hout, Mout)
END
```

Fig. 4. Systolic processor for DNA comparison: schematic and netlist description

operators to the right. The place-and-route step is thus independent of the size of the configurable operators. It is also very fast since a set of wires (a bus) is routed as a single entity.

Third, the regularity can be fully exploited, both for placement and routing. The editor provides the possibility to replicate a regular pattern that has been previously placed-and-routed. These features usually don't exist, or are extremely limited, in other tools, leading to a poor exploitation of the regularity [5]. In the context of automatic loop parallelization, for example, the synthesis step for deriving a hardware regular array from high level specifications can now be shrunk to a few minutes [3]. To keep the global synthesis time low, the mapping time onto a reconfigurable structure needs to be very fast.

5 Conclusion

The tool set we have presented does not aim to challenge the today's best placers and routers (such as VPR [2]) in terms of speed. The objective is to offer a prospective tool that fits non conventional architecture features and benefits from the absence of layers from the high level behavioral logic to the hardware execution management.

Our approach focuses on genericity, by decoupling the tools from the model. No assumption is made over the range of representable architectures. This means that an architecture is described as an arbitrary composition of resources (logic, routing, ...). The routing channels can start and end at any location on the architecture. The logic blocks can be LUTs, operators, potentially pieces of hardwired circuits, etc. These last features make this approach particularly well suited for designing non conventional FPGA structures such as specialized virtual FPGAs.

References

1. L. Lagadec, B. Pottier Object Oriented Meta tools for Reconfigurable Architecture, Conference on Reconfigurable Technology: FPGAs and Reconfigurable Processors

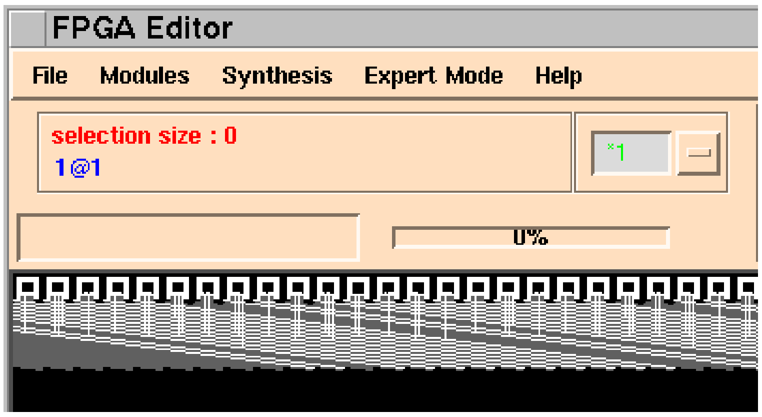


Fig. 5. FPGA editor: it shows a partial view of the DNA systolic array.

for Computing and Applications, SPIE Proceedings 4212 in Photonics East 2000, Boston, 2000.

2. V. Betz, J. Rose, A. Marquardt, Architecture and CAD for Deep-Submicron FPGAs, Kluwer Academic Publishers, 1999.
3. D. Lavenier, P. Quinton, S. Rajopadhye, Advanced Systolic Design, in Digital Signal Processing for Multimedia Systems, Chapter 23, Parhi and Nishitani Eds, March 1999.
4. G. Fabregat, G. Leon, Olivier Le Berre, B. Pottier, Embedded system modeling and synthesis in OO environments. A smart-sensor case study, in *Compiler and Architecture Support for Embedded Systems (CASES'99)*, 1999.
5. E. Fabiani, D. Lavenier, Placement of Linear Arrays, FPL 2000: 10th International Conference on Field Programmable Logic and Applications, Villach, Austria, Aug 2000
6. W. Fornaciari, V. Piuri, Virtual FPGAs: Some steps behind the physical barrier. In Parallel and Distributed Processing (IPPS/SPDP'98 Workshop Proceedings), LNCS 1388, 1998.
7. P. Guerdoux-Jamet, D. Lavenier Systolic Filter for fast DNA Similarity Search ASAP'95, Strasbourg, France, 1995.
8. C. Fraser, D. Hanson, A retargetable C compiler: design and implementation, The Benjamin/Cumming Publishing Company, Inc., 1995.
9. L. McMurchie, C.Ebeling, PathFinder: A Negotiation-Based Performances-Driven Router for FPGAs, in *FPGA'95*, 1995.
10. J. Babb, R. Tessier, A. Agarwal, Virtual Wires: Overcoming Pin Limitations in FPGA-based Logic Emulators, Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines'93, 1993.