Towards Portable Hierarchical Placement for FPGAs

Florent de Dinechin, Wayne Luk, Steve Mckeever

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Towards Portable Hierarchical Placement for FPGAs

Florent de Dinechin, Wayne Luk, Steve McKeever

No 3776

Octobre 1999

THÈME 2

ISSN 0249-6399 ISRN INRIA/RR--3776--FR+ENG

INRIA
RHÔNE-ALPES

Towards Portable Hierarchical Placement for FPGAs

Florent de Dinechin, Wayne Luk, Steve McKeever

Thème 2 — Génie logiciel
et calcul symbolique
Projet Arénaire

Rapport de recherche n ° 3776 — Octobre 1999 — 17 pages

Abstract:

Field Programmable Gate Arrays (FPGAs) are usually programmed using languages and methods inherited from the domain of VLSI synthesis. These methods, however, have not always been adapted to the new possibilities opened by FPGAs, nor to the new constraints they impose on a design. This paper addresses in particular the issue of laying out the various components of an architecture on an FPGA. The problem is to embed placement information in FPGA-oriented hardware description languages, in a way that is both expressive enough to be useful, and abstract enough to be portable from one FPGA architecture to the other. A generic placement framework is defined to address this problem, and two prototype implementations of this framework are presented, for Xilinx 6200 and Xilinx 4000 devices, on the example of a bit-serial complex multiplier.

(Résumé : tsvp)

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN (France)
Téléphone : 04 76 61 52 00 - International: +33 4 76 61 52 00
Télécopie : 04 76 61 52 52 - International: +33 4 76 61 52 52

Vers un placement hiérarchique reciblable pour les FPGAs

Résumé :

Les réseaux de cellules reconfigurables (FPGAs) sont programmés au moyen de langages et de méthodes héritées de la syntheèse VLSI, mais ces méthodes ne sont pas toujours adaptées aux nouvelles possibilités offertes par les FPGAs, ni aux nouvelles contraintes qu'ils imposent. Cet article s'intéresse en particulier aux questions de placement des différents composants d'un circuit dans le FPGA. Le problème est l'expression de contraintes de placement dans les langages de description de matériel pour ces architectures, et ce d'une manière suffisamment expressive pour être utile, mais suffisamment abstraites pour être portable d'une architecture à l'autre. Un cadre formel est défini pour répondre à cette attente, et deux prototypes d'implémentation de ce cadre sont présentées sur les architectures très différentes des circuits Xilinx 4000 et 6200, avec l'exemple d'un multiplieur bit-série complexe.

1 Introduction

The complex issue of programming FPGAs may be approached in a wide range of ways. One extreme is to consider that the designer should only have to sketch his design in an abstract way, leaving to automatic tools as much of the implementation job as possible, with as little human intervention as possible. This hands-off approach reduces development time and costs, at the expense of the performance of the implementation. At the other extreme, when performance is critical, the designer has to intervene in the whole design process. This may include low-level implementation work and require important expert knowledge and much longer development time.

Usually the implementation of a design on FPGA falls somewhere in the middle of these two extremes. The tools, while increasingly useful, still require a lot of technology-dependent knowledge from the designer. This balance between automation and manual intervention has to be considered in the three steps of a typical implementation flow for an FPGA : technology-mapping, placement and routing. One of the factors to consider in each of these steps, for example, is whether to keep the design hierarchy or flatten it to perform a global analysis.

Technology-mapping is obviously technology dependent, and currently well handled by automatic tools. So is routing, which the designer usually only controls by expressing simple and abstract timing constraints.

The placement problem is a difficult, NP-hard optimisation problem. Currently, the mainstream approach is to leave it to the back-end tools as well, because good heuristics exist which can take timing constraints into account (see for example [8] and its references), and placement constraints are difficult for the designer to manage because of the great number of degrees of freedom. However, most designs involve regular arrays of similar components, and there is often a straightforward "ideal" placement, ideal in the sense that it places connected components close to another. In such case, expressing this placement should greatly improve both implementation time and quality of the result : a general rule of place and route tools (both in VLSI and FPGA) is that if the placement is good (i.e. if it minimizes the distance between connected logic blocks) then the routing will be good as well and, perhaps more important, obtained quickly. The purpose of this paper is therefore to define a simple and

yet powerful way of expressing this kind of placement information in an FPGA design.

Another strong motivation of expressing placement is run-time partial reconfiguration, as allowed by some recent FPGAs. The idea is that one can change a small part of a design at run-time, for example to change the value of a hard-wired constant, without the time penalty of reconfiguring the whole design [5]. For this feature to be useful and efficient, one needs to control the placement of the components which will be swapped in and out at run time.

We investigate a placement framework which is *adaptable* (or *portable*) : it can easily be adapted to suit a specific FPGA technology. There are several motivations to this approach. The first is the design of libraries of macro-components similar to the "intellectual property cores" which have appeared in the VLSI world. These cores are usually placed and routed optimally, and we wish to offer the same possibility in the FPGA world. The second motivation of an adaptable placement is to provide a simple back-end to high-level tools. For example, Alpha [3, 1] is a language for the manipulation of recurrence equations which can generate systolic arrays. Currently, Alpha outputs VHDL which can be synthesized for most FPGAs. However the placement information – important in the case of systolic arrays, and present in Alpha – cannot be included in this VHDL in a general way. We believe that Alpha should output its systolic arrays in some HDL including generic placement information. The mapping of this placement on a target FPGA should then be done by back-end tools for each FPGA, just like technology mapping.

This paper is organized as follows. In section 2 we discuss placement questions specific to FPGA programming, and the features that a general placement framework should have. Section 3 proposes such a framework. Sections 4 and 5 describe two of its implementations : in the Pebble HDL [4] and in the PamDC development system [2]. Both implementations are compared on the same example design. Section 6 discusses the results of these experiments.

2 Layout approaches for FPGAs

Placement issues in the design hierarchy The hierarchy of a complex design may be split into three (roughly defined) levels for which the problem of

technology dependence is very different. Figure 1 illustrates this decomposition in the case of an FFT using bit-serial arithmetic. In this example, the whole chip may contain a full FFT, or only a part of it, depending on its size. The FFT is built from a certain number of butterfly elements. Each of these elements is made primarily of a complex multiplier and two adders. A complex multiplier is made of four real multipliers and two adders. A bit-serial Lyon fixed-point multiplier is a systolic array made of two types of basic cells [6] [1]. Finally each cell is made of a few gates and registers.

The lowest level The base bricks of a design are primitive gates and simple base blocks, either combinatorial (a full adder...) or sequential (a small finite state machine, the basic cell the bit-serial multiplier on Fig. 1...).

The optimal design of such components depends obviously on the technology, notably on the granularity of the FPGA (how many FPGA cells you need to implement a given function). These components may be hand-coded, although their small size ensures that *the vendor tools optimise them efficiently*. They may also be best expressed as *behaviourial* hardware description languages.

The top level The main two placement problems there are to take the size of the chip into account, and to match the inputs and outputs of the design with the actual boundaries of the chip. These questions are technology-, chip- and application-dependent, and will usually need manual intervention. Most

[1] Bit serial arithmetics have the advantage that the size of a bit-serial adder is constant with respect to the number of bits (in standard arithmetic it is proportional to this number). More important, the size of the multiplier is linear with respect to the number of bits, where standard multipliers have a quadratic size. Even more important, there is no need for data busses, as the data are propagated in series, which proves a real advantage on FPGAs where routing is limited like the 6000 series. The speed penalty of bit-serial communications is compensated by pipelining, and by the fact that the clock speed of bit-serial operators is independent of the bit number.

Numerically, on the 6000 series, a bit-serial multipliers for n-bits number needs $12n$ cells, whereas the standard multiplier requires $4n^2$ cells. Therefore a bit-serial approach is worthwile as soon as $n = 4$. The ratio of performance is roughly constant : the bit-serial approach has a constant critical path of about 20ns and the time of a multiplication is n times this period, while the period of the multiplier is roughly proportional to n with a proportionality constant of about 10ns.

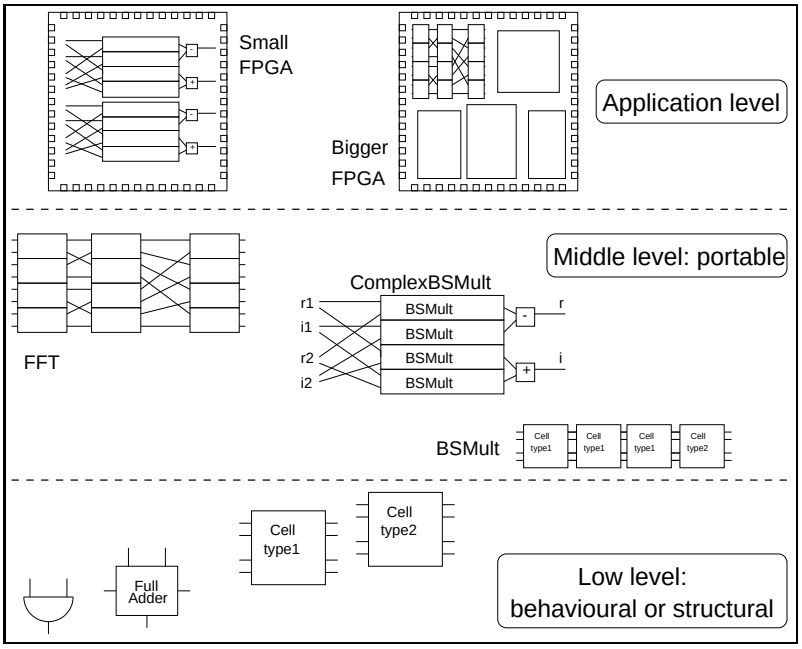

FIG. 1: A typical design hierarchy for a FFT

FPGAs, however, have facilities to alleviate the last problem. Examples include the "Veraring" for Xilinx 5000 devices and the "wireless registers" for Xilinx 6200 devices.

Intermediate levels Components such as FFTs, filters or microcontrolers often have a straightforward placement ensuring both compactness and locality of routing, especially for regular designs. The dependence to technology of this placement should only be in the low-level components used. The example we take throughout this paper is a bit-serial complex multiplier (see Fig. 1) whose basic cell is a 4×3 block on a Xilinx 6200 series and a 2×2 block on the 4000 series, the rest of the placement being the same on both technologies. That is what we will call an adaptable placement.

Such a placement, however, should be flexible enough to depend on more factors than just the technology : it should adapt to the lower-level components used, since there is often a range of tradeoffs there between time and area. It

should also be able to depend on the top-level placement : one may want, for instance, to rotate a placement (according to the external inputs and outputs) or to change its aspect ratio (according to chip size).

Only recent FPGAs allow designs large enough to have various intermediate layers. However it is obvious that the ongoing increase of the number of gates in an FPGA will make these component layers increasingly important (FPGA libraries begin to appear, albeit not adaptable and without placement).

Adaptable placement It is currently possible, using standard languages as VHDL or Verilog, to port the same middle-hierarchy component on almost any FPGA family, provided one is content with the push-button implementation approach.

It should be fairly easy to standardize the expression of timing constraints in such languages. The task is more difficult if we want to standardize the placement constraints, because placement is much more technology dependent, as the metrics change from one FPGA to another. To address the problem of adaptable placement, we first have to answer the following question : what is common to the placement and routing problem on most current (and hopefully future) FPGAs ? Answers are few :

- The logic blocks are laid out as a grid, in such a way that a 2D cartesian coordinates system can be used to describe placement. However the granularity and the size of this grid varies from one FPGA to another.
- Local connexions should be encouraged : routing resources are limited, and expensive in term of time. Since the routing is programmable, signals from one component to the other have to go not only through wires, but also through switches : there is logic on the wires as well. This is a major difference between FPGA and VLSI, and its cost is bound to increase with integration.

Our placement framework tries to address these constraints.

3 An adaptable placement framework

The adaptable placement framework we define here is based on several existing placement methods, notably the RLOC hierarchical relative placement

attributes for the Xilinx 6000 series [11], the placement engine of PamDC [10, 2], and the high-level placement features of Ruby [9]. We atttempt to combine and generalise these techniques in an useful way.

We describe here our placement framework in a very abstract way, wich is independent of any HDL. Note that implementations of this framework will depend a lot on the HDL, as we show in sections 4 and 5.

Basics For each instance of a component of the hierarchy, its placement is very classically defined by a standard set of *variables* holding the size of its bounding box, the coordinates of its inputs and output pins, and the coordinates of its sub-components. Coordinates are relative to some reference point of the component, say the bottom-left corner. We will denote, for a component `A`, the width and height of the bounding box of A as `A.w` and `A.h`, and its coordinates as `A.x` and `A.y` (for simplicity, we will ignore in the following the placement of the I/O ports).

These variables use the coordinate system *of the target FPGA*. This coordinate system is discrete, therfore these variables will be integers. It will thus be possible to express *absolute* placement (for example for the device-specific part of a design at the top level) as for example :

```
A.x = 12;   A.y = 0
```

The placement of a middle-hierarchy component is adaptable if it is not expressed absolutely, but as a function of the variables of its sub-components. For example, if the component `C` is built out of components `A` and `B`, the following expresses that `A` is placed to the right of `B` :

```
B.x = A.x + A.w;   B.y = A.y
```

This placement is adaptable since it is a function of the variable `A.w`. More generally, an adaptable placement is described as a *system of equations* (and possibly inequations) relating variables. An actual placement, on an actual FPGA, will be a solution to this system.

These relations may involve other variables of the HDL, including notably the generic parameters of a component and loop indices. Moreover, although it is not the subject of this paper, a standardized expression of timing constraints should eventually be integrated in this framework.

Program 1 is the complete example of a adaptably-placed adder in our Pebble implementation. Note that this is mostly a toy example, since an adder

will probably belong to the low level of our hierarchy decomposition. A more realistic sized example is that of the FFT seen in Fig. 1, where the relative placement of all the components is apparent. The corresponding program will be partly given later.

Program 1 A placed adder in Pebble

```
BLOCK AddCompactTall (n:GENERIC)
                     [a, b: VECTOR (n-1..0) OF WIRE; cin: WIRE]
                     [sum:  VECTOR (n-1..0) OF WIRE; cout: WIRE]
   LOCAL carry: VECTOR (n..0) OF WIRE;
BEGIN
   carry(0) <- cin;
   GENERATE FOR i = 0..(n-1)
      BEGIN
         fa(i): FullAdder [a(i), b(i), carry(i)] [s(i), carry(i+1)];
         LET fa(i).x = 0;  LET fa(i).y = i*fa(0).h;
      END;
   cout <- carry(n);
   LET w = fa(0).w;   LET h = n*fa(0).h;
END
```

This program consists of a header, defining a generic parameter `n`, then input busses (vectors of wires) `a` and `b` and input wire `cin`, then output bus `sum` and output wire `cout`, then a local bus `carry`. The body of the program then structurally describes the adder, in a way very similar to structural VHDL.

The placement is expressed by equations as in previous examples (here `fa(i)` is a component instance name). It is adaptable in the sense that it is independent on the size of the full adder `fa`.

In this basics approach, some of the equations have to define the size of the component, usually as a composition of the sizes of the sub-components (see the last lines in Prog 1, where the dotless variables are the variables of the block `AddCompactTall` itself). Using higher-level placement constructs (see the following paragraph), however, makes it possible to compute the size automatically.

To get an actual placement of this adder, one must somehow inject in this system of equations the sizes of the low-level component `FullAdder`. This

component will typically be automatically placed and routed by the back-end tools.

High-level placement expressions These relations may have a high-level syntax, for example to express relative placement. For examples the two previous equations
`B.x = A.x + A.w; B.y = A.y`
may be written `Beside(A,B)`. See program 3 for a real example. Classical placement constructions such as a row, column or matrix of component may also be expressed this way. Placement transformations such as symmetries and rotation also have simple expressions in this framework. The exact syntax of high-level placement, however, is very dependent on the HDL, and its reduction to basic equations is not always as simple as in the case of the `Beside`. This is the subject of current active research, and well beyond the scope of this paper.

Resolution This system of constraint is resolved in some way. This may require interaction with the designer : in most cases the placement will initially be under-constrained. Unknown placement may be either left to the back-end tools, or pointed to the designer for him to add relevant relations.

4 Polynomial placement in Pebble

Pebble is a structural HDL developed at Imperial College [4]. It was therefore a good choice to experiment on adaptable placement.

Polynomial placement expressions For this implementation, placement information is restricted to polynomial equations of several variables, these variables including the generics and loop indices. These polynomials are handled internally in their normal developed form. The purpose is to avoid unrolling all the loops, so as to manipulate all the placement information of a regular design in a compact form. This not only allows to perform several static checks, but also greatly improves the user interface in the resolution of the placement, as seen below. There are drawbacks, though : placement expressions can only use

the $+$, $-$ and $\times$ operators, which excludes in particular the very useful *max* operator. Similarly this choice limits the expressive power of the `generate if` statement. However what remains is a language well suited to regular designs.

Resolution A hierarchical placement resolution engine has been written. It propagates known values in the polynomials equations, until one of these turn into either a contradiction (which generates an error) or a definition for the value of a new variable. The placement of the block is considered successful if all the standard variables and generics of the instances are known. A known variable is defined as a polynomial function of the block generics and (possibly) the loop indices.

When a placement is underdefined (which is usually the case initially) the resolution engine points out the still undefined variables, and asks the designer to give their values. Thanks to the polynomial internal form, this process is very explicit, with typical messages like :

```
Couldn't resolve equation: add2.x - add0.x - add0.w - n*add1.w = 0
Unknown variables: add2.x, add0.x
```

This message invites the designer to define either `add0.x` or `add2.x`.

Implementation on the 6000 series A successful placement can then be translated into Xilinx `RLOC` attributes for VHDL. The advantage of our framework over standard `RLOC`s is the flexibility : one may change the size of a component without having to propagate this change by hand to all the parts of the design where this component is involved. This is examplified in the following.

The complex multiplier example We wrote a hierarchical complex multiplier (the biggest part of the bit-serial FFT which could fit on our 6216 chip). We first wrote it without any placement and performed logical simulations. Then we began adding placement constraints and running the code through our placement engine, until the only unresolved equations were due to unknown sizes for the low-level components `cell1` and `cell2`. The corresponding adaptable placed code is given as programs 2 and 3 (note the use of BESIDE and BELOW constructs).

Program 2 A placed bit-serial multiplier in Pebble

```
/* Lyon's bit-serial multiplier (adaptable placement) */
BLOCK bsmult (n) [xi,yi,ri,clk,rst: WIRE]
                 [pso, ro: WIRE]
  LOCAL  x, y, ps, r : VECTOR (0..n) OF WIRE ;
         xo, yo : WIRE
  BEGIN
    x(0) <- xi;  y(0) <- yi;  ps(0) <- FALSE;  r(0) <- ri;
    GENERATE FOR i = 1..n-1
      BEGIN
        cells1(i): bsmultcell [x(i-1), y(i-1), ps(i-1), r(i-1), clk, rst]
                              [x(i), y(i), ps(i), r(i)]
      END;
    LET cells1(i).x = cells1(i).w*(i-1);  LET cells1(i).y=0;
    cell2: bsmultcell2 [x(n-1), y(n-1), ps(n-1), r(n-1), clk, rst]
                       [x(n),y(n),ps(n),r(n)];
    LET cell2.x = cells1.w * (n-1);  LET cell2.y=0;
    xo <- x(n);  yo <- y(n);  pso <- ps(n);  ro <- r(n);
    LET h = cell2.h;   LET w = cell2.x + cell2.w
  END
```

We then had these two low-level components placed by Xact6000 automatically, reported the sizes (4×4) in the Pebble code (there would be no theoretical difficulty in having this size information moved around by the tools themselves, however this would involve more development work to read proprietary file formats), and performed the final placement in our tool. Then we output VHDL with RLOCs to Xact6000, which could route it in a few seconds (when we disable the RLOC output, Xact6000 is no longer even able to place and route the design). This lead to the bit-serial complex multiplier given in fig. 2. One can see that the basic type 1 cell is a 4 by 4 block, and that some of the cells are only used for routing. The timing analyzer estimated the frequency of the bit-serial multiplier at 45MHz.

Later we spent some time optimizing the initial automatic placement of the basic cells. The type 1 cell changed from a 4 by 4 block to a 4 by 3 block. We then only needed to change one line in the Pebble code to get a new placement

Program 3 A placed complex multiplier in Pebble

```
/* complex multiplier: (a+ib)*(c+id) */

BLOCK ComplexBsmult (n) [a,b, c,d, ri,clk,rst: WIRE]
                        [rp, ip, ro: WIRE]
  LOCAL
    pac, pbd, pad, pbc, roac, robd, road, robc : WIRE
  BEGIN
    ac:  bsmult (n) [a,c,ri,clk,rst] [pac, roac];
    bd:  bsmult (n) [b,d,ri,clk,rst] [pbd, robd];
    ad:  bsmult (n) [a,d,ri,clk,rst] [pad, road];
    bc:  bsmult (n) [b,c,ri,clk,rst] [pbc, robc];
    sub: bssub [pac,pbd,clk,rst] [rp];
    add: bsadd [pad,pbc,clk,rst] [ip];
    LET ac.x = 0;   LET ac.y = 0;
    LET BELOW(ac,bd,ad,bc);
    LET BESIDE(bc, add);
    LET BESIDE (bd,sub);
    LET w = ac.w+add.w;  LET h = 4*ac.h;
    ro <- roac;
   END
```

of the whole complex multiplier shown in fig. 3, and again, the whole computing time of our tool and Xact6000 to get this layout, routing included, was less

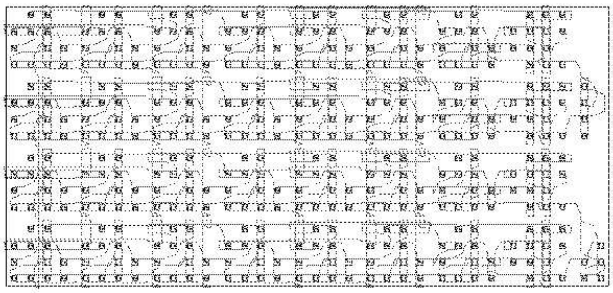

FIG. 2: Complex multiplier in XactStep 6000 (first version)

RLOCs for the `bsmult` component :

```
attribute rloc of cell1 : label is "X,3*n-3,Y,0,";
attribute rloc of cell2 : label is "X,3*i-3,Y,0,";
```

RLOCs for the `ComplexBsmult` component :

```
attribute rloc of ac : label is "X,0,Y,0,";
attribute rloc of bd : label is "X,0,Y,4,";
attribute rloc of ad : label is "X,0,Y,8,";
attribute rloc of bc : label is "X,0,Y,12,";
attribute rloc of sub : label is "X,3*n+1,Y,4,";
attribute rloc of add : label is "X,3*n+1,Y,12,";
```

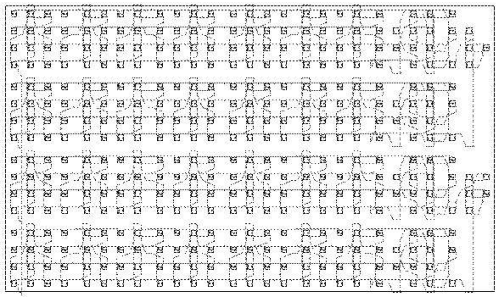

FIG. 3: Complex multiplier in XactStep 6000 (optimized version)

than ten seconds. The estimated frequency of the bit-serial multiplier is here 54MHz. This figure also shows some of the RLOCs generated in the VHDL code by our placement tool.

5 Prototype implementation in PamDC

Work is underway on a translator from Pebble to PamDC [2] to target the 4000 series of FPGAs. To evaluate its feasability we translated by hand the whole design, trying to port the placement as well (having to rewrite the whole hardware description to target a second FPGA family otherwise somehow contradicts the objective of adaptability).

The object-oriented model of PamDC and its built-in placement engine simplified the implementation of our framework on this environment. We just needed to define a new `PlacedNode` class inheriting from the basic `Node` class.

Placement for the 4000 series PamDC outputs annotated netlist files for the Xilinx back-end tool XactM1. This last tool is very timing-oriented, therefore we present two series of experiments, the first without and the second with timing constraints. Each of these experiments consists of implementing a non-placed complex multiplier, and a placed one.

Figure 4 shows the result of the untimed experiment. The unplaced design is evaluated at 71Mhz by the timing analyser, and the placed one is evaluated at 94 MHz. This shows that, in the absence of timing constraints, placement improves timing and area as expected.

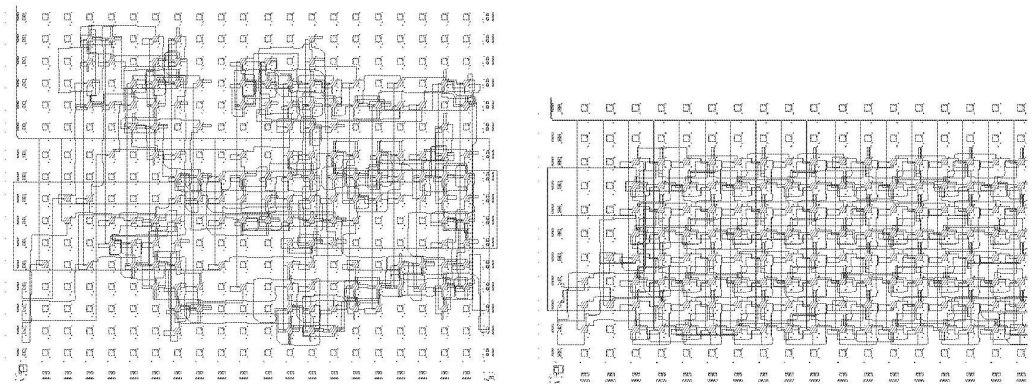

FIG. 4: Unplaced and placed complex multiplier on the 4000

However, when we repeat the experiment with a timing constraint of 100Mhz (which is the maximum allowed by the logic depth involved) on all the nets, we obtained very similar layouts, but both placed and unplaced designs are evaluated at little more than 100Mhz. Besides, the unplaced design is obtained more than twice as fast as the placed one (5mn30 vs. 2mn30) : obviously, giving a placement annoys XactM1 more than it helps.

The only advantage of our placed design seems therefore to be compacity. This, however, is only the case because XactM1 has spare space and has no reason not to use it : when we try to increase the multiplier to 21 bits (in which case it occupies more than 95% of the chip resources), XactM1 can still place it and get a 100MHz estimated frequency for an initially unplaced design.

6 Discussion and conclusion

Some negative results The main result of this second experiment is thus that it is useless, on a 4010 and using XactM1 tools, to spend effort on the placement : one gets as efficient a circuit, both in terms of speed and area, from an unplaced description. Expressing placement doesn't even reduce the implementation time as it does under Xact6000. Our example design is special in that it is completely linear, with few inputs and outputs, and the experiment should be repeated with more complex netlist structures. However, these re-

sults with the 4000 series justify the current trend in FPGA synthesis, which is to center design optimization on timing constraints more than on placement constraints, and to discard the design hierarchy to perform the place and route.

On the other hand, our approach proves very useful and efficient on the 6000 series, with its hierarchical routing and hierarchical approach to place and route. However this FPGA family is notorious for the poor quality of its automatic place and route tools.

Long term considerations We still believe that the long-term evolution of the FPGA technology will make the flatten-all approach of current place-and-route tools less and less viable [7]. The complexity of the place and route problem increases much faster than the size of the design (however heuristics will improve as well, along with routing resources).

Besides, the VLSI world is more and more relying on libraries of "intellectual property" cores, and this begin to filter to the FPGA world. The weakness of the flatten-all approach in this case is already examplified in FPGA-based PCI coprocessor where the (rather huge) PCI interface is implemented on the FPGA, and has to be placed and routed along the user design in each iteration of the development process. This may add several hours to the design compilation time.

Xact6000 is very bad at the push-button approach, however it has a clever approach to hierarchical place and route which allows placement reuse, which considerably speeds development of big designs consisting of several well defined functional units. We believe that the ideal back-end tools should allow for this hierarchical approach to place-and-route, while retaining the efficient automatic approach of current tools for the lower components of the hierarchy. In this case, a unified placement framework will be useful.

Current and future work This work now needs to be extended in two directions : on its back end, we need to study more FPGA families and their most recent back-end tools to find ways to exploit placement information if it is available. On the front end, there is a lot of ongoing work on language aspects, concerning the high-level placement constructs which have only been evoked here. Another point is that we have only considered equations so far in

placement, while it is clear that inequations should also be incorporated in the framework. This, of course, will lead to more complex resolution strategies.

Acknowledgements

This work was partly done at the Imperial College of Science, Technology and Medicine in London, UK, and partially supported by an INRIA post-doctoral fellowship.

Références

[1] Florent de Dinechin. Libraries of schedule-free operators in Alpha. In *Application Specific Array Processors*. IEEE Computer Society Press, July 1997.

[2] Digital Equipment Corporation. *PamDC : a C++ Library for the Simulation and Generation of Xilinx FPGA Designs*, 1997.

[3] 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.

[4] W. Luk and S. McKeever. Pebble : a language for parameterized and reconfigurable hardware design. In *International Workshop on Field Programmable Logic and Applications*, Tallin, Estonia, September 1998.

[5] W. Luk, N. Shirazi, and P. Cheung. Compilation tools for run-time reconfigurable designs. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 56–65, Napa Valley, CA, April 1997.

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

[7] J. Rose and D. Hill. Architectural and physical design challenges for one-million gate FPGAs and beyond. In *FPGA'97, ACM Symposium on FPGAs*, pages 129–132, Monterey, CA, February 1997.

[8] S. A. Senouci, A. Amoura, H. Krupnova, and G. Saucier. Timing driven floorplanning on programmable hierarchical targets. In *FPGA'98,*

ACM/SIGDA International Symposium on FPGAs, pages 85–92, Monterey, CA, February 1998.

[9] R. Sharp and O. Rasmussen. Using a language of functions and relations for VLSI specification. In *FPCA'95, Conference on Functional Programming Languages and Computer Architecture*, pages 45–54, La Jolla, CA, June 1995.

[10] J. Vuillemin, P. Bertin, D. Roncin, H. Shand, M. andTouati, and P. Boucard. Programmable active memories : Reconfigurable systems come of age. *IEEE Transactions on VLSI Systems*, 4(1) :56–69, March 1996.

[11] Xilinx Corporation. *XactStep Series 6000 User Guide*, 1997.

Unit´e de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399