A Framework for Developing Parametrised FPGA Libraries

W. Luk, S. Guo, N. Shirazi and N. Zhuang

Department of Computing, Imperial College, 180 Queen's Gate,
London SW7 2BZ, UK

Abstract. We suggest that the productivity of FPGA users can be improved by adopting design libraries which are optimally implemented, rich in variety, easy to use, compatible with incremental development techniques and carefully validated. These requirements motivate our research into a framework for developing FPGA libraries involving the industrial-standard VHDL language and the declarative language Ruby. This paper describes the main elements in our framework, and illustrates its application to the Xilinx 6200 series FPGAs.

1 Introduction

FPGA users often face a dilemma. The advance in microprocessor and custom hardware technologies is increasing the pressure for improving functionality and performance of FPGA-based implementations; such improvements, however, may necessitate the use of low-level, device-specific FPGA features, thus lengthening design cycles and reducing the opportunity for design reuse.

The productivity of FPGA users can be greatly enhanced by having library elements – or macros – which are efficient and easy to use. A single parametrised library can be used to generate a wide range of implementations that support multiple architectures, variable bit widths and trade-offs in speed, resource usage and reconfiguration time. Such libraries enable effective FPGA utilisation by exploiting device-specific features whenever desirable, allowing designers to achieve optimal performance and resource usage while minimising the need for knowledge of low-level details. In the past FPGA libraries were confined to relatively simple bit-level structures such as buffers, counters and adders; increasingly complex libraries for signal processing and other applications are beginning to appear.

Existing tools for library generation, however, suffer from several drawbacks. First, they only support a restricted number of parameters. For instance, while many tools support designs parametrised by data widths, they do not include the separation between connections as a parameter; in practice, the ability to control the separation between connections is often necessary so that components can be combined without additional routing between them. Other parameters, such as the number of pipeline stages in arithmetic designs, are also rarely supported by current commercial tools. Second, most library generators do not produce parametrised designs. Changing one parameter of a component may necessitate rebuilding every component of the entire design. Third, while some library generators provide simulation models for their circuits, there is no systematic way

of validating that the designs generated have the desired behaviour. Ensuring that libraries are free from errors is a growing concern, because the fact that they are parametrised makes simulation a poor choice for verification.

We have recently begun a project on parametrised library design which aims to overcome these drawbacks. In this project, a declarative language known as Ruby is used for exploring and validating design libraries; the resulting architectures are then implemented using a subset of VHDL. The following sections describe this framework in greater detail, and its use in library development for Xilinx 6200 series FPGAs will be illustrated.

2 Overview of Framework

Our aim is to improve the productivity of FPGA users: to develop designs with higher quality, in shorter time and at lower costs than existing ones. Specifically our framework should provide design libraries which are:

- optimally implemented – otherwise inefficiency is introduced each time a non-optimal library element is used;
- rich in variety – the availability of a wide variety of library elements is desirable to suit different user needs;
- easy to use – the libraries should be provided in a form which can be easily processed by many tools, and can be easily integrated with other designs;
- compatible with incremental development techniques – users should be able to modify their library-based implementations with little effort;
- carefully validated – there should be a high degree of confidence in the correctness of the libraries.

These requirements motivate our approach for developing FPGA libraries, which involves the declarative language Ruby for exploring and validating designs, which are then implemented using the VHDL language. Figure 1 shows the major elements in our framework – Rebecca [10] is the Ruby design environment

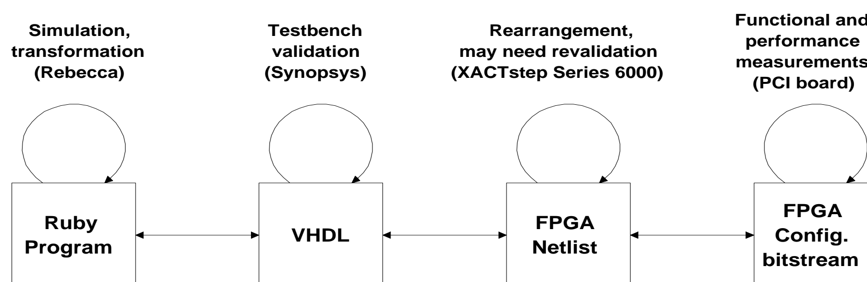

Fig. 1. Overview of our design framework. Rebecca, Synopsys and XACTstep Series 6000 are respectively the design tools for Ruby, VHDL and Xilinx 6200 FPGAs.

that we are developing; Synopsys is a well-known industrial system which deals with VHDL synthesis; XACTstep Series 6000 is a development tool from Xilinx supporting their 6200 series FPGA design; and a PCI-based platform [11] is used to test our library implementations. The translation from Ruby to parametrised VHDL is currently done by hand, and we are exploring ways of automating this procedure. Producing FPGA netlists and generating the configuration bitstreams are both automatic.

As later discussions will show, the use of Ruby helps to meet the above requirements, particularly those for optimal implementation, richness in variety and careful validation. The use of VHDL, an industry standard, contributes mainly to optimal implementation, ease of use and compatibility with incremental development techniques. Details of these two languages will be given in the sections to follow. Let us first outline the architecture of the Xilinx 6200 series FPGAs to which our libraries are targeting.

Each Xilinx 6200 FPGA consists of a rectangular array of cells, each of which can be configured as a two-input function block, or as a multiplexer, or as a latch [3]. There is a hierarchical routing network for connecting cells; however all of our current libraries only require nearest neighbour connections, which are provided as one input link and one output link in each of the four directions for each cell. Links can be used for signal routing. Figure 2a shows a cell configured as an xor-gate, with its inputs a and b from the west and east side and the output c to the south side; the a input is also extended to the east side. Figure 2b shows how this cell can be combined with an and-gate to form a halfadder; this unit can then be used in more complex designs discussed later.

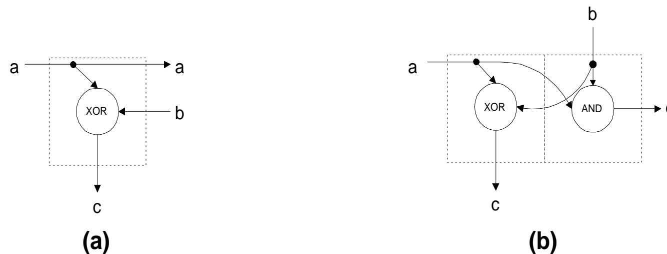

Fig. 2. (a) A Xilinx 6200 FPGA cell configured as an xor-gate. (b) A halfadder using two 6200 FPGA cells.

3 Ruby

Ruby [5] is a declarative language which has been used in describing and developing various architectures, including image processing designs [1], butterfly networks [9] and systolic data structures [4]. A design in Ruby is represented as a binary relation R, which relates the interface signals x and y in the form of $x\,R\,y$. x is the domain of R and corresponds to connections on the west or the north side, while y is the range of R and corresponds to connections on the south or the

east side. An inverter, for instance, can be described by $(x\ \ not\ \ y) \Leftrightarrow (y = \neg x)$. The two designs in Figure 2 can be expressed as:

$$\begin{array}{lll} a \ \ xorcell \ \ \langle c, \langle b, a\rangle\rangle & & \text{where } c = xor\ \langle a, b\rangle, \\ \langle a, b\rangle \ \ hadd \ \ \langle c, d\rangle & & \text{where } c = xor\ \langle a, b\rangle \text{ and } d = and\ \langle a, b\rangle. \end{array}$$

The function *xor* and *and* model the corresponding FPGA cell operations. A piece of wire is modelled by *id*, the identity relation, which is given by $(x\ \ id\ \ y) \Leftrightarrow (x = y)$.

Complex designs in Ruby can be formed by composing simpler designs. For instance, two components Q and R with a compatible interface connected in series is denoted by $Q\ ;\ R$ (Figure 3a):

$$x\ (Q\ ;\ R)\ y \ \Leftrightarrow\ \exists s : (x\, Q\, s) \wedge (s\, R\, y).$$

The $\exists$ symbol means that, unlike x and y, s is not an interface variable of the composite and cannot be observed.

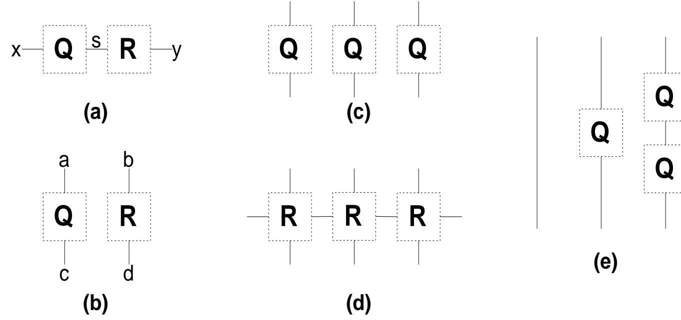

Fig. 3. (a) $Q\ ;\ R$. (b) $[Q, R]$. (c) $\mathsf{map}_3\, Q$. (d) $\mathsf{row}_3\, R$. (e) $\triangle_3$ Q.

If there are no connections between Q and R, the composite design is represented by parallel composition $[Q, R]$ (Figure 3b), where

$$\langle a, b\rangle\, [Q, R]\, \langle c, d\rangle \ \Leftrightarrow\ (a\, Q\, c)\ \wedge\ (b\, R\, d).$$

Parametrised operators are used to capture regular patterns of composing designs. Repeated compositions of n copies of Q are described by Q^n or $\mathsf{map}_n\, Q$ (Figure 3c), so for instance $not^3 = not\ \ ; not\ \ ; not$ and $\mathsf{map}_3\ xor = [xor, xor, xor]$. A row of n components is represented by $\mathsf{row}_n\, R$ (Figure 3d), while a triangular-shaped array is described by $\triangle_n\, Q$ (Figure 3e). As we shall show later, these parametrised operators provide a concise and yet versatile way of describing architectures.

To deal with designs operating on time-varying data, a relation in Ruby can be considered to relate an infinite sequence of data in its domain to another

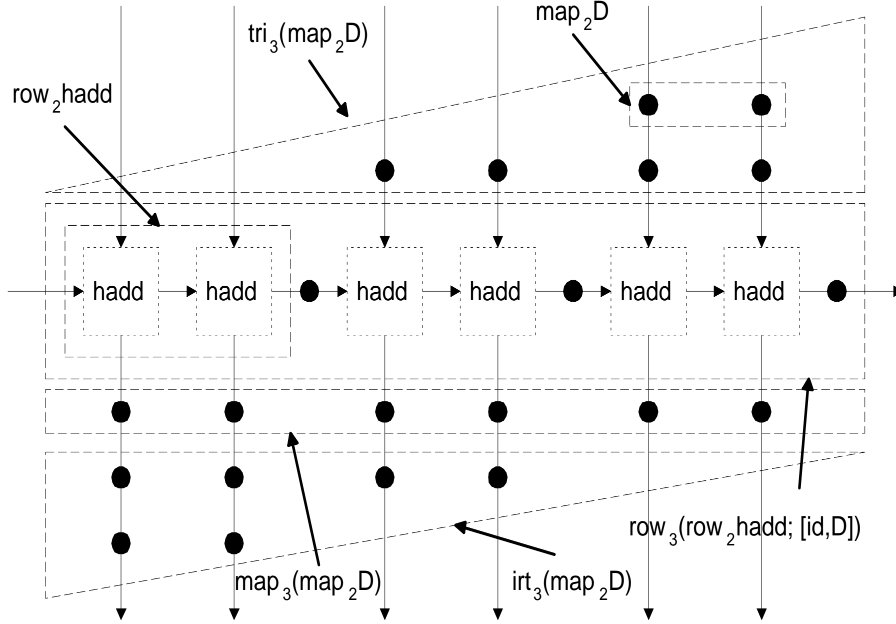

Fig. 4. Design *padd*1: an m-stage pipelined adder ($m = 3$, $k = 2$, $n = 6$). Note that tri corresponds to $\triangle$ and irt corresponds to $\tilde{\triangle}$. The black disks represent latches.

infinite sequence in its range; elements in these infinite sequences can be regarded as values appearing at an interface at successive clock cycles. Given that $\forall t$ denotes "for all values of t", an inverter can be described by

$$x \ not \ y \ \Leftrightarrow \ \forall t : y_t \ = \neg x_t .$$

A latch can be modelled by a delay relation $\mathcal{D}$, given by

$$x \ \mathcal{D} \ y \ \Leftrightarrow \ \forall t : x_{t-1} = y_t .$$

An advantage of Ruby is that a single parametrised expression can be used to generate a wide range of architectures with different performance trade-offs. Consider the description of an m-stage pipelined adder *padd*1 which adds a single-bit word to an n-bit unsigned word, where $n = mk$ (Figure 4):

$$\begin{aligned} padd1 &= inskew \ ; \ \mathsf{row}_m \, (\mathsf{row}_k \, hadd \ ; \ [id, \mathcal{D}]) \ ; \ outskew, \\ inskew &= [id, \triangle_m \, (\mathsf{map}_k \, \mathcal{D})], \\ outskew &= [(\mathsf{map}_m \, (\mathsf{map}_k \, \mathcal{D}) \ ; \ \tilde{\triangle}_m \, (\mathsf{map}_k \, \mathcal{D})), \ id]. \end{aligned}$$

Note that the triangular-shaped array $\tilde{\triangle}_n \, R$ can be obtained by reflecting $\triangle_n \, R$ in a vertical axis. The components *inskew* and *outskew* are arrays of registers

for 'skewing' the input and output of the pipelined adder; they ensure that all bits in the same word will emerge in the same clock cycle.

This design has a latency of m cycles and $m(n+1)$ latches. A fully-pipelined design with an n-stage pipeline can be obtained by having $k = 1$ and $m = n$, while a single-stage pipelined adder can be obtained by having $k = n$ and $m = 1$.

Another advantage of Ruby is its capability for validating parametrised designs. Since Ruby is based on the mathematical notion of relations, we can verify formally that an efficient but non-obvious design has the same behaviour as a design which is obvious but inefficient. For instance using correctness-preserving transformations [8], it can be shown that the design *padd*1 behaves the same as a row of n copies of the *hadd* component with an m-stage register at each of their outputs. In other words, one can show that:

$$padd1 = \mathsf{row}_m\,(\mathsf{row}_k\; hadd)\ ;\ [\mathsf{map}_m\,(\mathsf{map}_k\,\mathcal{D}),\ \mathcal{D}]^m \qquad \text{where } mk = n.$$

Transformations that relate one design to another can be arranged to form a design tree so that designers can rapidly explore the available design options and assess their trade-offs. Further discussions about design transformations, such as those for serialising and transposing designs, and the use of design trees can be found elsewhere [7].

Design in Ruby is supported by the Rebecca development environment. Rebecca allows mixed high-level (such as integer-level) and low-level (such as bit-level) simulation, and mixed numerical and symbolic evaluation. There are also tools for animating designs, assisting design transformations and compiling designs into non-parametrised VHDL, EDIF and various proprietary netlist description formats ([2], [6]).

Let us summarise how Ruby meets the requirements for a library design framework discussed in Section 2. First, the parametrised operators in Ruby, such as map and row, provide the guiding principles for parametrising architectures; they also make it easy to develop designs with nearest-neighbour connections and regular or systolic structures, which map particularly well onto FPGAs to give optimal implementations. Second, the parametrised operators enable concise design descriptions, allowing designs to be developed and modified quickly which helps to build up a rich body of library elements. Finally, Ruby provides a framework for simulating, visualising and formally verifying parametrised designs – an important part of design validation.

4 VHDL

When we are satisfied with a Ruby design, we convert it into VHDL so that the libraries are expressed in an easily usable form. The motivations for using VHDL include: (a) it is an international standard with a large user community; (b) it is reasonably vendor-independent but extensible by the user; (c) it supports parametrised, re-usable designs; (d) there are many VHDL tools and some of them, such as the Alliance system, are available in the form of freeware; (e) it is relatively straightforward to integrate VHDL descriptions from different

sources. Currently we can compile Ruby descriptions into non-generic VHDL; the compilation into parametrised VHDL is under development.

Although VHDL can be used to capture parametrised library elements, the placement and routing of components within such elements will often need to be specified to ensure optimality. Although this step is probably unnecessary for random logic implementation, it is often desirable for structured designs in which inefficiency in a repeating unit will be amplified many times. The solution to this requirement is the use of the VHDL mechanism for user-defined attributes: this method allows us to specify location and shape constraints for particular blocks. The XACTstep Series 6000 development tool (Figure 1) will take these constraints into account when it places and routes the design. Using this technique, we have been able to capture highly-optimised parametrised libraries in VHDL. If users feel that under specific circumstances automatic tools can do better, they can simply comment out the constraint specification before the place-and-route stage.

Architectural parameters, such as the number of stages of pipelining, can be included as generics in a VHDL description. As an example, the following is a VHDL `ENTITY` description for the design *padd*1 (Figure 4):

```
ENTITY padd1 IS
  GENERIC ( m       : integer := 3;   -- number of pipeline stages
            n       : integer := 6);  -- input word size
  PORT    ( cin     : IN  std_logic;                    -- carry input
            inword  : IN  unsigned ( (n-1) downto 0 ); -- input word
            outword : OUT unsigned ( (n-1) downto 0 ); -- output word
            cout    : OUT std_logic;                    -- carry output
            clk     : IN  std_logic;  -- clock input for latches
            clr     : IN  std_logic); -- clear input for latches
END padd1;
```

If we adopt the structure in Figure 2b for implementing the halfadders in *padd*1, then the instantiation $m = n = 4$ will result in the fully-pipelined design in Figure 5, while the instantiation $m = 2$ and $n = 4$ will result in the two-stage pipelined design in Figure 6.

We have been able to extend this framework to include as a generic parameter the separation between adjacent connections. Figure 7a shows a fully-pipelined 6-bit fulladder array, with outputs separated by one cell; Figure 7b shows a two-stage pipelined 6-bit fulladder array, with outputs separated by two cells. Both implementations are generated by instantiating a single parametrised VHDL description. The trade-offs in features such as speed, area, power consumption and reconfiguration time of various instantiations can be recorded so that users can select the most appropriate values for the parameters for a given application.

5 Summary

Our framework is intended for rapid production of parametrised FPGA libraries, with emphasis on optimality of implementations and ease of use. It has been

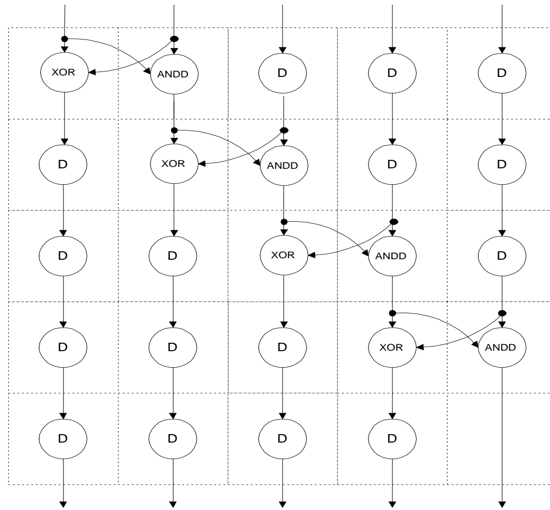

Fig. 5. A fully-pipelined implementation of *padd*1, with $m = 4$ and $n = 4$. The component ANDD corresponds to an and-gate with a latched output.

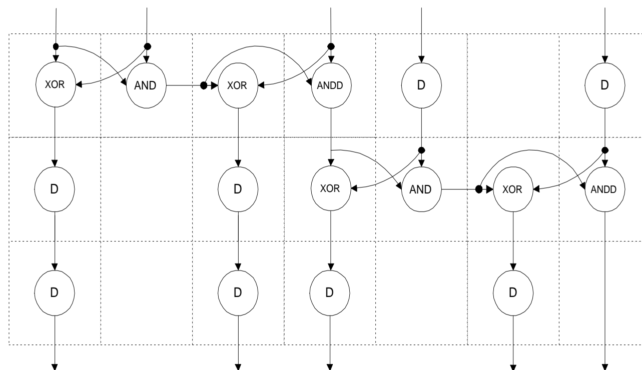

Fig. 6. A two-stage pipelined implementation of *padd*1, with $m = 2$ and $n = 4$.

applied to the development of reusable building blocks for Xilinx 6200 series FPGAs. The key elements in our framework include the use of Ruby for exploring, parametrising and verifying designs, and the use of VHDL for capturing the structure and constraints of parametrised implementations. This approach supports incremental design – since users can integrate the parametrised VHDL libraries with their own code, there is no need to generate library components individually when a different instantiation of parameters is required.

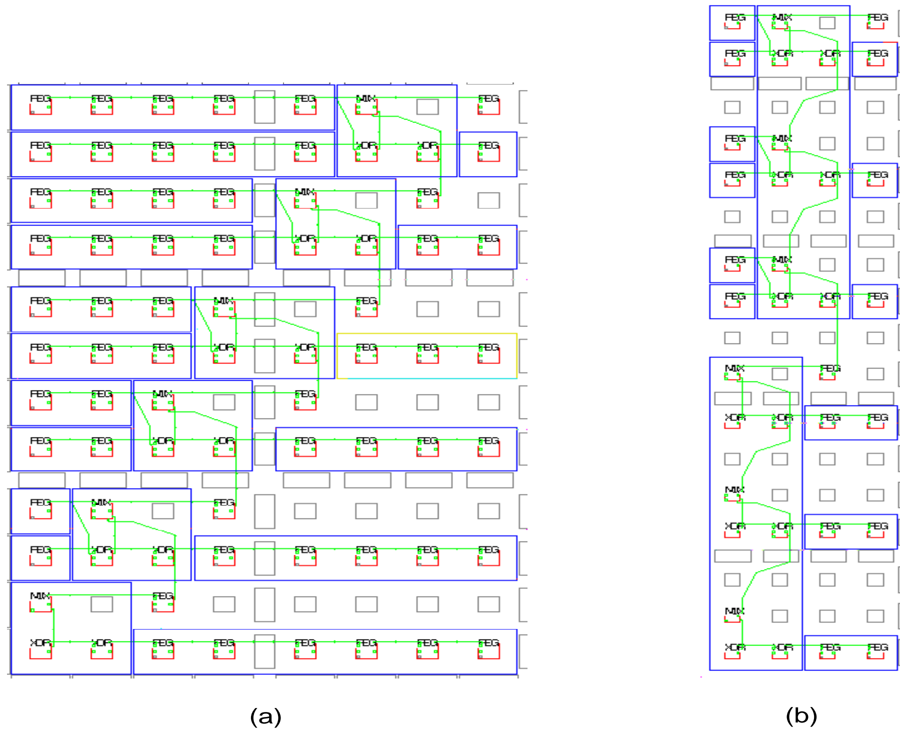

Fig. 7. (a) A fully-pipelined fulladder array, the outputs of which are separated by one cell. (b) A fulladder array pipelined by every three cells, the outputs of which are separated by two cells.

Library evaluation and validation can be carried out at three levels: applying the simulation, animation and verification tools in Ruby; using the VHDL testbench facilities; and testing the synthesised designs on a suitable hardware platform. Users do not need to know about Ruby to use our parametrised VHDL libraries.

Using this framework, we have been able to create many parametrised libraries in a short amount of time. The libraries that we produce include arithmetic circuits such as comparators, adders and multipliers, storage structures such as register files, and signal processing designs such as Gaussian filters and edge detectors. Most of these libraries achieve over 95% utilisation of FPGA cells within a rectangular block, and all of them can be placed in any location and in any orientation. Preliminary analysis indicates that our libraries are very competitive when compared with other library designs implemented in comparable technology.

Future work includes applying our library development techniques to producing complex function cores for applications such as video processing and com-

munications, developing strategies for optimally instantiating and composing library elements to satisfy user constraints, and providing coherent compilation and evaluation facilities for parametrised libraries using Ruby, VHDL and other languages, tools and hardware platforms.

Acknowledgements

Many thanks to John Gray for providing much of the inspiration behind this work, and to Geoffrey Brown, Mike Dean, Douglas Grant, Tom Kean, John O'Leary and Seng Shay Ping for their help. The support of Xilinx Development Corporation and the UK Overseas Research Student Award Scheme is gratefully acknowledged.

References

1. M. Aubury and W. Luk, "Binomial filters", *Journal of VLSI Signal Processing*, Vol. 12, No. 1, January 1996, pp. 35–50.
2. P.Y.K. Cheung, W. Luk and P.I. Mackinlay, "Hardware-software cosynthesis for the Riley system", in *Hardware-Software Cosynthesis for Reconfigurable Systems*, IEE Digest 96/036, 1996, pp. 10/1-10/5.
3. S. Churcher, T. Kean and B. Wilkie, "The XC6200 FastMap processor interface", in *Field Programmable Logic and Applications*, W. Moore and W. Luk (eds.), LNCS 975, Springer, 1995, pp. 36–43.
4. S. Guo and W. Luk, "Compiling Ruby into FPGAs", in *Field Programmable Logic and Applications*, W. Moore and W. Luk (eds.), LNCS 975, Springer, 1995, pp. 188–197.
5. G. Jones and M. Sheeran, "Circuit design in Ruby," in *Formal Methods for VLSI Design*, J. Staunstrup (ed.), North-Holland, 1990, pp. 13–70.
6. W. Luk, G. Jones and M. Sheeran, "Computer-based tools for regular array design", in *Systolic Array Processors*, J. McCanny, J. McWhirter and E. Swartzlander (eds.), Prentice-Hall International, 1989, pp. 589–598.
7. W. Luk, "Systematic serialisation of array-based architectures", *Integration, the VLSI Journal*, Vol. 14, No. 3, February 1993, pp. 333-360.
8. W. Luk, "Systematic pipelining of processor arrays," in *Transformational Approaches to Systolic Design*, G.M. Megson (ed.), Chapman and Hall Parallel and Distributed Computing Series, 1994, pp. 77–98.
9. W. Luk and T. Wu, "Towards a declarative framework for hardware-software codesign", in *Proc. Third International Workshop on Hardware/Software Codesign*, IEEE Computer Society Press, 1994, pp. 181–188.
10. W. Luk, "A declarative approach to incremental custom computing", in *Proc. IEEE Symposium on FPGAs for Custom Computing Machines*, P. Athanas and K.L. Pocek (eds.), IEEE Computer Society Press, 1995, pp. 164–172.
11. W. Luk, N. Shirazi and P.Y.K. Cheung, "Modelling and optimising run-time reconfigurable systems", in *Proc. IEEE Symposium on FPGAs for Custom Computing Machines*, K.L. Pocek and J. Arnold (eds.), IEEE Computer Society Press, 1996.

This article was processed using the LaTeX macro package with LLNCS style