ELSEVIER

Journal of Systems Architecture 47 (2001) 315–337

JOURNAL OF SYSTEMS ARCHITECTURE

www.elsevier.com/locate/sysarc

An integrated system for developing regular array designs

Shaori Guo [a,*], Wayne Luk [b]

[a] *Philips Semiconductors, Sunnyvale, California, CA 94086, USA*

[b] *Department of Computing, Imperial College of Science, Technology and Medicine, 180 Queen's Gate, London SW7 2BZ, UK*

Abstract

This paper describes an integrated system for developing regular array designs based on the block description language Ruby. Ruby supports concise design description and formal verification. A parametrised Ruby description can be used in simulating, refining and visualising designs, and in compiling hardware implementations such as field programmable gate arrays. Our system enables rapid design production, while good design quality is achieved by (a) the efficient instantiation of device-specific libraries, (b) the size optimisation of bit-level components using the design refiner, and (c) the exploitation of regularity information at source level in the library composition process. The development and implementation of several median filters are used to illustrate the system. © 2001 Elsevier Science B.V. All rights reserved.

Keywords: Regular array designs; Ruby; CAD tools; Median filters; FPGAs

1. Introduction

A regular array design is a circuit consisting of processing elements connected to their neighbours in a regular manner. Regular array designs have been widely used in signal and image processing, multimedia and communication systems [17,28]. Many designers recognise the importance of developing high performance regular array designs rapidly and cheaply, particularly for systems-on-a-chip applications or embedded systems where demanding computations have to be implemented efficiently.

We believe that the key to effective development of regular array designs is to use descriptions and support tools that can fully exploit their characteristics, such as regularity and spatial and temporal locality, for their specification, validation and implementation. Most VLSI CAD tools based on standard hardware description languages such as VHDL and Verilog, however, do not make the best use of these characteristics. A VHDL description of a regular array design, for example, is usually larger than it needs to be, and takes a long time to code, debug and modify. Besides, VHDL does not provide an efficient mechanism for capturing the regular layout of a regular array design, and final implementations often rely on an automatic placement and routing tool to generate physical layouts which may not preserve the inherent regularity in the original description. Consequently the placement and routing process is often time-consuming, and does not always result in area and time efficient designs. Furthermore VHDL is a complex language, and formal verification of VHDL designs is often difficult.

The objective of the work described in this paper is to build a system in which regular array

* Corresponding author.
E-mail address: scott_guo@yahoo.com (S. Guo).

designs can be efficiently captured, refined, validated and implemented. The system should significantly reduce design efforts and design cycles. To meet this objective, we have developed an integrated system which has the following main features:

1. The system is based on a simple and powerful notation for representing regular array designs. Both architecture and behaviour can be captured in a single parametrised description, and both high-level (such as word-level) and low-level (such as gate-level) aspects of a design can be described in a uniform framework.
2. The system facilitates graphical representations which can be generated automatically from high-level descriptions. It provides useful visual feedback which enables designers to rapidly obtain an overview of a design, to locate specific parts on which they can focus, and to obtain intuitions for design optimisation and verification.
3. The system supports a hierarchical design paradigm: there is a refiner that automatically produces, from high-level descriptions, efficient low-level designs that satisfy user-given constraints. This enables designers to focus on the high-level architectural aspects without being overwhelmed by low-level details.
4. There are also high-level synthesis tools such as hardware compilers to reduce design time and complexity. As a result, designers can spend more time on exploring alternative designs and on evaluating the effects of different data representations.
5. Our system provides a variety of ways for design validation, depending on the level of detail, generality and confidence in design correctness. It supports design visualisation, mixed bit-level, numerical and symbolic simulation as well as algebraic transformation.
6. Finally, the system is based on a declarative language which supports design optimisation by correctness-preserving transformations. This framework offers users confidence in the correctness of its optimisations, since their validity can be checked by calculation and proof.

The declarative language, Ruby, that we use was invented by Sheeran [38] and further developed by Jones [13–15], Luk [20–23] and others [36,37,39]. The main reasons that we have chosen Ruby rather than other hardware description languages such as VHDL [6,31] are as follows:

- Ruby supports succinct design capture. A circuit, especially one with a regular structure, can be described more concisely in Ruby than in other hardware description languages such as VHDL. The regularity information can be exploited in the synthesis and validation processes.
- Ruby can capture both the behaviour and the architecture of a design within a single notation and in an elegant manner; this is seldom the case with other languages.
- Ruby has a well-understood semantics. Consequently it is easy to document design decision, to refine a design, to compare alternative designs, and to demonstrate the correctness of an implementation.

Various median filters will be used to illustrate these points.

While several theories for regular array design [28] have been under development for over ten years, the contribution described in this paper appears unique in providing an integrated system for producing such designs, particularly in dealing with practical bit-level implementation aspects. The following reviews some other tools for automatic synthesis of regular array designs from high-level descriptions.

ALPHA is a system with a functional language based on the formalism of affine recurrence equations [7,40,41]. Regular array designs are derived through formal, correctness-preserving transformations applied to programs in ALPHA. The resulting design is specified as a variant form of ALPHA known as ALPHA0, which can be used for generating netlist, and the netlist can then be implemented using a commercial CAD tool. ALPHA has shown its strength in automatic synthesis of regular array designs, especially for applications related to affine problems. Our work is complementary to ALPHA in that Ruby is best for designs conveniently captured using binary relations, while ALPHA is best for designs conveniently captured using affine recurrence equations.

Another tool based on a functional language for hardware development is Lava [4]. Lava exploits polymorphism and higher order functions in the functional language Haskell [3] to capture design regularity which results in abstract and general descriptions. Functional language features such as type classes and monads have been exploited to implement standard circuit analyses such as simulation, formal verification and the generation of VHDL and EDIF for producing real circuits. So far, Lava does not include automatic refinement facilities or visual aids as our system does.

Singh prototyped a hardware compiler, known as the Glasgow Ruby Compiler, which compiles a dialect of Ruby into FPGA devices [39]. The version of Ruby which Singh has adopted is specialised for describing FPGAs, and uses a different convention from the one used in this paper. Layout information is explicitly specified in his Ruby expressions which are similar to those in the OAL language [26] that our descriptions are compiled to (see Section 3.5 for further details of our hardware compiler). A design sketcher [5] has also been designed as part of the Glasgow Ruby Compiler. The sketcher differs from ours in that it is based on an interface conversion scheme, and diagrams produced seem to contain more jogs.

Sharp and Rasmussen have been working on the T-Ruby design system for VLSI circuits starting from a high-level, mathematical specification of their behaviour [37]. The T-Ruby system provides facilities to perform design transformation and simulation, to prove the correctness of a design and to translate the Ruby descriptions into VHDL for synthesis by a commercial tool. T-Ruby, at present, does not include facilities such as automatic refiner, design sketcher or visualiser.

Li and Leeser [18] have developed the HML system based on the functional language SML [30]. HML supports advanced type checking and type inference techniques to verify hardware design rules, and designs can be translated into a synthesisable subset of VHDL. Unlike our approach, HML does not exploit regularity in design implementation, and no visualisation facilities have been reported so far.

The above overview is intended to provide an outline of related work, and is not meant to be exhaustive. For instance, many useful theoretical and practical results have been reported in the series of Conferences on Application-specific Array Processors (in which [7] appears). However, it is not always clear whether the tools described in related publications have been used in developing actual hardware implementations for working systems. We have used the tools presented in this paper for two FPGA-based systems: CHS2x4 [1] and Riley [16]. Further details can be found in Sections 3.5 and 4.7.

The paper is structured as follows. Section 2 presents a brief introduction to Ruby. An overview of the integrated system is described in Section 3. Section 4 presents the development of several regular median filter designs as a case study. Conclusions are drawn in Section 5.

The short introduction to Ruby in Section 2 is written mainly for readers who have a background in declarative languages, particularly those supporting higher order functions such as Haskell [3] and ML [30]. We have, however, separated the details involving the Ruby language from the essential ideas. Readers should be able to appreciate the main features of our system in Section 3 and the block descriptions in Section 4 (using Tables 1–5 in Sections 4.3–4.6 to understand the function and connectivity of the blocks), without following the intimate details of the Ruby expressions.

2. Introduction to Ruby

In this section, we provide a brief introduction to Ruby, the language used in our system for parametrised description of block diagrams. Our major focus will be put on the definitions and concepts relevant to this paper; further details about the background and the theoretical aspects of Ruby can be found elsewhere [13,21,37].

In Ruby a design is captured by a binary relation R, which relates the interface signals x and y in the form of xRy. For instance the max operator, which produces the maximum of two numbers, can be described by

$$\langle x, y\rangle \max z,$$

where $z = (\text{maximum}(x, y))$. So $\langle 3,4\rangle \max 4$ and $\langle 10,6\rangle \max 10$.

The min operator for finding the minimum of two numbers can be described in a similar way.

2.1. Combinational primitives

There are two kinds of combinational primitives in Ruby: wiring primitives and computation primitives.

Wiring primitives select or regroup components of composite data. The simplest wiring primitive is id, defined by:

$$x \text{ id } y \Longleftrightarrow x = y. \tag{1}$$

Other common wiring primitives are defined as follows:

$$x \text{ fork } \langle y,z\rangle \Longleftrightarrow x = y = z, \tag{2}$$

$$\langle x,y\rangle \pi_1 z \Longleftrightarrow x = z, \tag{3}$$

$$\langle x,y\rangle \pi_2 z \Longleftrightarrow y = z, \tag{4}$$

$$\langle x,y\rangle \text{ swap } \langle u,v\rangle \Longleftrightarrow (x = v) \wedge (y = u), \tag{5}$$

$$\langle x,\langle y,z\rangle\rangle \text{ rsh } \langle\langle u,v\rangle,w\rangle \Longleftrightarrow (x = u) \wedge (y = v) \wedge (z = w), \tag{6}$$

$$\langle\langle x,y\rangle,z\rangle \text{ lsh } \langle u,\langle v,w\rangle\rangle \Longleftrightarrow (x = u) \wedge (y = v) \wedge (z = w). \tag{7}$$

Computation primitives include operations on integers, such as:

$$\langle x,y\rangle \text{ add } z \Longleftrightarrow z = x + y, \tag{8}$$

$$\langle x,y\rangle \text{ mult } z \Longleftrightarrow z = x \times y, \tag{9}$$

$$\langle \mathbf{s},\langle x,y\rangle\rangle \text{ muxr } z \Longleftrightarrow z = \begin{cases} x, & \text{if } \mathbf{s} = 0, \\ y, & \text{otherwise,} \end{cases} \tag{10}$$

and operations on bits, such as:

$$\langle x,y\rangle \text{ and } z \Longleftrightarrow z = x \wedge y, \tag{11}$$

$$\langle x,y\rangle \text{ or } z \Longleftrightarrow z = x \vee y, \tag{12}$$

$$\langle x,y\rangle \text{ xor } z \Longleftrightarrow z = (\neg x \vee \neg y) \wedge (x \neg y), \tag{13}$$

$$x \text{ not } y \Longleftrightarrow y = \neg x. \tag{14}$$

2.2. Series and parallel composition

The central idea in Ruby is that complex designs can be formed by composing simpler designs, using combinators which are higher order functions describing common patterns of computation. For instance, two components R and S with a compatible interface can be put together by (;) to form a composite design $R\ ;\ S$ (Fig. 1(a)):

$$x(R\ ;\ S)y \Longleftrightarrow \exists t : (x\ R\ t) \wedge (t\ S\ y), \tag{15}$$

where x is the domain and y is the range of $(R\ ;\ S)$. Clearly, the domain of $(R\ ;\ S)$ is that of R, its range is that of S and the range of R is connected to the domain of S. The $\exists$ symbol means that, unlike x and y, t is not an interface signal of the composite and cannot be observed.

In general, we can describe the composition of n identical copies of a circuit R by repeated series composition R^n (Fig. 2(a)), where

$$\begin{aligned} R^0 &= \text{id}, \\ R^{i+1} &= R\ ;\ R^i \end{aligned} \tag{16}$$

id, the identity relation, has been defined in Section 2.1.

If there are no connections between R and S, the composite is represented by parallel composition $[R, S]$ (see Fig. 1(b)), where

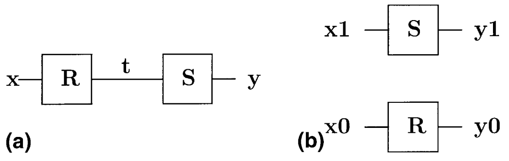


Fig. 1. (a) $R\ ;\ S$. (b) $[R,S]$.

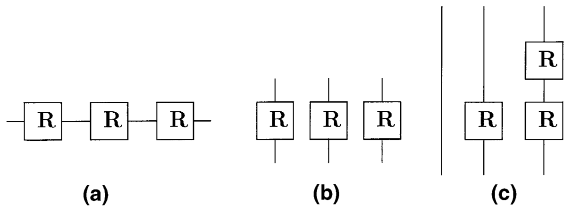


Fig. 2. (a) R^3, (b) $\text{map}_3\ R$ and (c) $\triangle_3\ R$.

$$\langle x_0, x_1\rangle [R, S] \langle y_0, y_1\rangle \iff (x_0 \; R \; y_0) \wedge (x_1 \; S \; y_1). \tag{17}$$

Given id is the identity relation defined in Section 2.1, we have the definitions of the following operators:

$$\mathsf{fst}\ R = [R, \mathrm{id}], \tag{18}$$

$$\mathsf{snd}\ R = [\mathrm{id}, R]. \tag{19}$$

Repeated parallel compositions of n copies of R are described by $\mathrm{map}_n\ R$ (Fig. 2(b))

$$\mathrm{map}_n\ R = [\underbrace{R, \ldots, R}_{n}]. \tag{20}$$

A triangular-shaped array is described by $\triangle_n\ R$ (Fig. 2(c)),

$$\triangle_n\ R = [\mathrm{id}, R, R^2, \ldots, R^{n-1}]. \tag{21}$$

2.3. Other combinators

Inverse is defined by

$$x \; R^{-1} \; y \iff y \; R \; x. \tag{22}$$

It can be treated as a reflected version of R.

Two rectangular components with connections on every side can be assembled together by the beside ($\leftrightarrow$) and the below ($\updownarrow$) operators such that:

$$\langle a, \langle b, c\rangle\rangle (R \leftrightarrow S) \langle \langle p, q\rangle, r\rangle \iff \exists t : (\langle a, b\rangle \; R \; \langle p, t\rangle) \wedge (\langle t, c\rangle \; S \; \langle q, r\rangle), \tag{23}$$

$$\langle \langle a, b\rangle, c\rangle (R \updownarrow S) \langle p, \langle q, r\rangle\rangle \iff \exists t : (\langle b, c\rangle \; S \; \langle t, r\rangle) \wedge (\langle a, t\rangle \; R \; \langle p, q\rangle). \tag{24}$$

Note that we adopt the convention that signals in the western and the northern sides are mapped onto the domain, and signals in the southern and the eastern sides are mapped onto the range [10].

Block diagram for beside and below are shown in Fig. 3(a) and (b), respectively. row_n is the generalisation of beside, which is defined as follows (Fig. 4(a)): given that $x = \langle x_0, \ldots, x_{n-1}\rangle$, $y = \langle y_0, \ldots, y_{n-1}\rangle$ are two n-tuples of signals and R is a relation, row_n is defined by

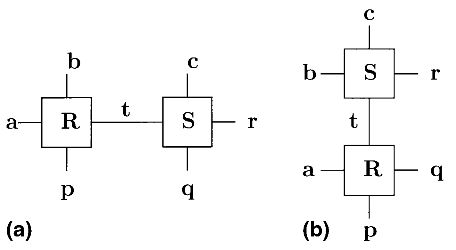


Fig. 3. (a) $R \leftrightarrow S$. (b) $R \updownarrow S$.

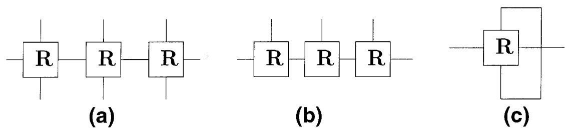


Fig. 4. (a) $\mathrm{row}_3\ S$, (b) $\mathrm{rdl}_3\ S$ and (c) loop R.

$$\langle a, x\rangle (\mathrm{row}_n\ R) \langle y, b\rangle \iff \exists s : (s_0 = a) \wedge (s_n = b) \wedge (\forall i : 0 \leqslant i < n . \langle s_i, x_i\rangle \; R \; \langle y_i, s_{i+1}\rangle). \tag{25}$$

Similarly, given $x = \langle x_0, \ldots, x_{n-1}\rangle$ is an n-tuple of signals and R is a relation, rdl_n (Fig. 4(b)) is defined by

$$\langle a, x\rangle (\mathsf{rdl}_n\ R) b \iff \exists s : (s_0 = a) \wedge (s_n = b) \wedge (\forall i : 0 \leqslant i < n . \langle s_i, x_i\rangle \; R \; s_{i+1}). \tag{26}$$

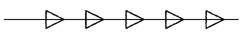
2.4. Sequential circuits

So far we have been using relations to model combinational circuits. There are primitives that do not possess a static interpretation. For example, a delay relation D is defined by

$$x \; D \; y \iff \forall t : x_{t-1} = y_t,$$

where x and y are time sequences containing data at successive clock cycles – this describes the behaviour of a latch. It has been shown that the Ruby laws for the static interpretation, such as those in Section 4.5, are also valid in the time sequence interpretation [13].

We use the symbols ▷ to represent delays. For example,



is a picture of D^5.

Latches are also used in serial circuits to prevent unbuffered loops in the feedback path. A design R containing an internal feedback path s can be modelled by the loop operator loop (see Fig. 4(c))

$$x(\text{loop } R)\ y \iff \exists s : \langle x,s\rangle\ R\ \langle s,y\rangle. \tag{27}$$

3. The integrated system

3.1. Overview

Fig. 5 shows an overview of the integrated system that we have implemented. The system consists of two parts. The first part includes an optimising transformer, a numeric/symbolic simulator and a performance analyser. The second part includes a Ruby refiner, diagram sketcher, design visualiser and a Ruby hardware compiler (labelled * in Fig. 5). Detailed descriptions on the first part can be found in [19,27], and this paper will focus on the second part, although the entire system will be outlined as follows:

- The optimising transformer provides assistance to optimise a high- or low-level specification according to user requirements on area and performance.
- The numeric/symbolic simulator performs numerical, gate-level and symbolic simulation.
- The performance analyser assesses the characteristics of a design; such as the number of a specific component, the critical path delay and the latency.
- The refiner produces a bit-level design from a high-level design, given the constraints on the input and/or output of a design.
- The Ruby sketcher produces design diagrams from the specification (high-level or low-level).
- The visualiser supports visualisation of both the behaviour and structure of a design by combining the facilities of the simulator and the sketcher.
- The hardware compiler compiles a Ruby program into hardware such as FPGAs.

There are three ways to check the correctness of a design with our integrated system: (1) formal verification, (2) numerical/symbolic simulation, and (3) design visualisation. We shall not describe formal

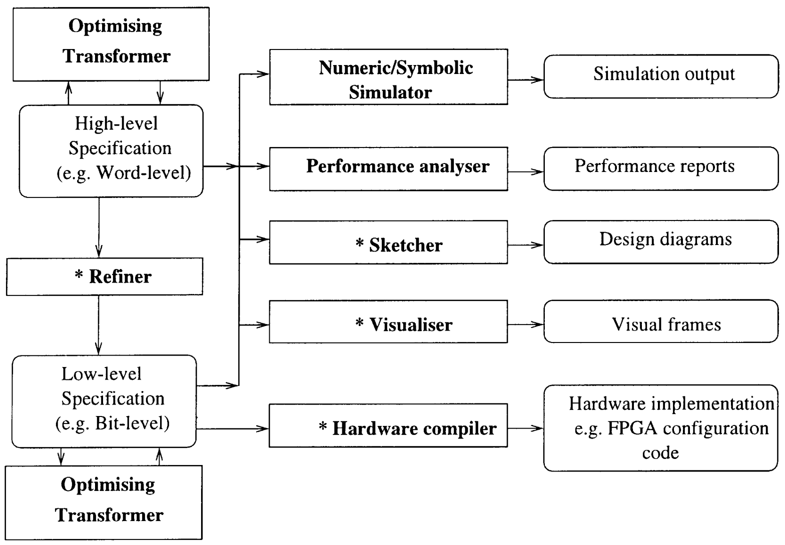


Fig. 5. Block diagram of the implemented design system. The components labelled * are the subject of this paper.

verification tools [37] or numerical/symbolic simulation tools [19] in this paper, since they have been covered elsewhere. We shall focus on using the design visualiser to validate our designs; relevant material will be presented in Sections 3.3 and 4.4.

Let us now look at the intended design flow supported by our integrated system. A design is started with its high-level specification. It is then validated using the numeric/symbolic simulator, or the visualiser. Next, the performance of the design can be analysed using the performance analyser, and the layout of the design can be obtained using the sketcher. If the design is not satisfactory, the optimising transformer may be employed to optimise the design. This process can be iterated several times until a satisfactory design is obtained. Once a satisfactory high-level design is obtained, a low-level design can be produced using the refiner. Like the high-level design, the low-level design can be further improved using the tools in the system. Finally, an optimal hardware design can be directly implemented using the hardware compiler.

The following sections describe the features of the sketcher, the visualiser, the refiner and the hardware compiler.

3.2. The sketcher

The sketcher produces design diagrams from Ruby programs. Since regular patterns are captured using combinators and computation patterns are explicitly represented in Ruby, the sketching procedure is largely syntax-directed and hence efficient. Furthermore, a simple sketching scheme has been developed which allows the component sizes to vary, so that connection positions can be adapted to align the inter-connecting wires either to the horizontal or to the vertical. Consequently, the sketcher can produce design diagrams with a minimum number of jogs, which minimises the effort for routing and improves the comprehensibilities of the produced diagrams. The sketcher also includes facilities for drawing particular parts of a circuit and for producing layouts to a specified level of detail.

Since the diagram generation process is syntax-directed, the quality of the diagram depends largely on the Ruby source program. To optimise the quality of the diagram, there is a module in the sketcher which performs source-level transformation. An example of the source-level transformation is shown in Fig. 6. Fig. 6(a) is the diagram of Ruby expression $\mathsf{row}_2\ ([A,B];Q;[C,D])$ produced without source-level transformation, while Fig. 6(b) is the diagram of the same Ruby expression generated after source-level transformation. For this example, given the relation $[-]$ such that x $[-]$ $\langle x\rangle$, the following Ruby law can be used to optimise the design to become the one in Fig. 6(b):

$$\mathsf{row}_n\ ([A,B];\ Q;\ [C,D]) = \mathsf{row}_n\ ([A,B];(\mathsf{snd}[-];\ \mathsf{row}_1 Q;\ \mathsf{fst}[-]^{-1});\ [C,D]). \tag{28}$$

More Ruby laws have been identified for layout optimisation and the details can be found in Ref. [8].

Like most tools in our system, the design sketcher is written in the language SML [30]. There is a parser for converting expressions in concrete syntax to the corresponding internal representations in abstract syntax. Other main modules include:

- a preprocessor for source level transformation;
- a type checker based on the unification algorithm [32] to ensure that interconnected blocks have a compatible interface;
- a mode manager that decides the level of detail at which the layout should be drawn, according to the source Ruby expression;
- a placer which produces a hierarchical placement of the layout;

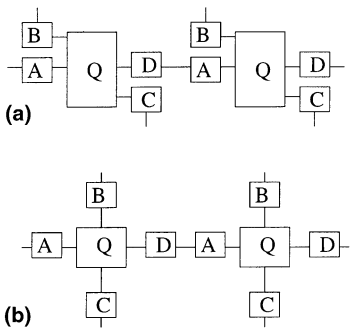


Fig. 6. Comparison of schematics from two equivalent Ruby expressions.

- a sizer which adds dimensions and connection points to the description of primitive cells;
- a router which ensures that the connections between adjacent cells are joined together properly;
- an output generator, which takes the result from the router and generates diagrams in formats such as Latex and Tcl [29].

3.3. The visualisation system

The visualisation system is a graphics-based tool that integrates the visualisation of design behaviour and structure by combining the sketcher and simulation facilities.

The main modules in our system are shown in Fig. 7. There is a graphical user interface for convenient interaction between the user and the system, which does not appear in the figure for the sake of clarity. The Ruby program to be visualised and simulation data are first supplied to the visualisation system as input. The sketcher produces a circuit schematic and a netlist which specifies the connectivity of components. The numerical/symbolic simulator takes the netlist and simulation data to produce a state table which records circuit states in terms of time sequences. The visualiser displays the schematic superimposed with numerical or symbolic data values indicating circuit states at each clock cycle. The circuit states change as simulation progresses and the visualisation sequences are accordingly produced showing the circuit state changes. We shall illustrate the operation of the system in Section 4.4.

Fig. 8 shows a snapshot of the graphical user interface of our visualisation system. The centre of

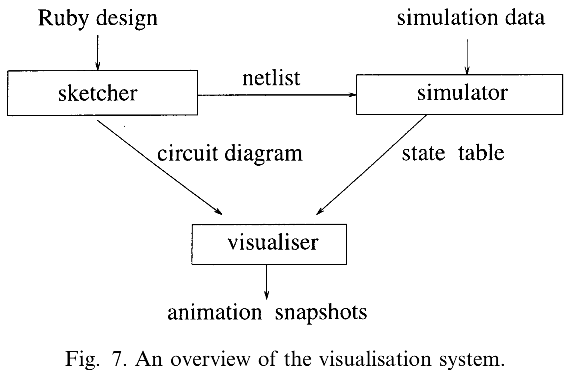


Fig. 7. An overview of the visualisation system.

the frame displays the block diagram of a convolver design. The operation of the design is animated by projecting a dataflow model on the block diagram. One can select to view data values on specific input and output ports and internal paths. Numerical, symbolic and bit-level simulation and their combination are supported and the animation speed can be adjusted.

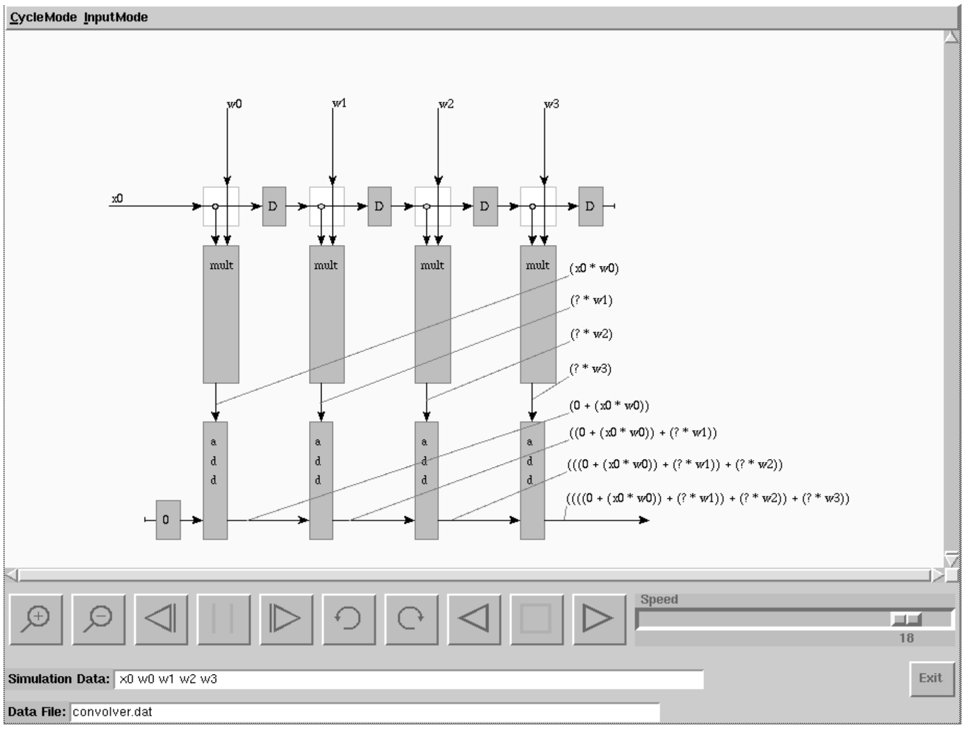
There are two simulation modes: one simulates the design cycle by cycle and the other supports sub-cycle simulation, showing how signals propagate through combinational components one after another. Fig. 8 shows the visualiser running symbolic simulation cycle by cycle. The button at the top left-hand corner allows the selection of simulation modes and input options. Simulation data can be provided from a file or from the command line input at the bottom. Various controls can be used to magnify designs, to choose step-by-step, continuous or cyclic simulation, and to adjust the simulation speed.

Our visualiser has been used in developing hardware libraries and designs involving run-time reconfiguration [25].

3.4. The refiner

In our system, both high-level, such as word-level, and low-level, such as bit-level designs can be described in Ruby. A high-level design operates on abstract data types such as integer or real. At low-level, abstract data such as integers are replaced by concrete data, such as bits; the associated operations on abstract data are also replaced by concrete ones. The data refinement from a high-level design to a low-level design is automatically performed using the refiner.

The bit-level implementation differs principally from the word-level one in that some constraints such as the size of components have to be considered, since in reality we cannot build circuits which are arbitrarily large. Also, there may be many possible bit-level implementations which meet the requirements of a given word-level design. One of the features of the refiner is that it can produce the most efficient bit-level design which satisfies the constraints given by the designer.


CycleMode InputMode
w0
w1
w2
w3
x0
D
mult
add
0
(x0 * w0)
(? * w1)
(? * w2)
(? * w3)
(0 + (x0 * w0))
((0 + (x0 * w0)) + (? * w1))
(((0 + (x0 * w0)) + (? * w1)) + (? * w2))
((((0 + (x0 * w0)) + (? * w1)) + (? * w2)) + (? * w3))
Speed
18
Exit

Simulation Data: x0 w0 w1 w2 w3

Data File: convolver.dat

Fig. 8. A snapshot of our visualiser carrying out symbolic simulation. The block diagram is generated automatically from a Ruby program. *D* represents a register.

The current version of the refiner takes constraints specifying the maximum and minimum values of external inputs to a circuit, and optionally the size of any internal connections. A constraint-propagation algorithm has been developed to calculate the size of a particular component. The idea is that the maximum and minimum values of inputs are propagated across the circuit. Once all constraints on a given component's inputs are known, the constraints on its internal connections and its outputs can be derived. Resolving the constraints fixes the size of the components and the width of the output data path. Given a library of parametrised bit-level operators and their sizes, our constraint-propagation procedure can be used to determine the widths of all data paths. A bit-level Ruby design can be constructed that is guaranteed to be the most efficient design produced by the constraint-propagation algorithm.

Another important feature of the refiner is that it can refine a word-level design into a number of efficient bit-level implementations, depending on the bit-level data representations such as unsigned and two's complement representations. As a result, the refiner facilitates exploring architectures and evaluating the effects of different bit-level data representations.

3.5. The hardware compiler

The hardware compiler compiles into hardware a bit-level Ruby program generated by the refiner, and the target code can be produced in various formats so that the same design can target different devices and different technologies. Some of the formats are device-independent such as structural VHDL and EDIF netlist for both ASIC development process and FPGA implementation, and some are device-dependent such as XNF and CFG which can only be implemented using the specific FPGAs. We have used the VHDL compiler in implementing designs on the Riley system [16].

An important feature of our hardware compiler is that there is a floorplanning module which provides rapid placement and routing of components. The floorplanning module generates layout by exploring the structure of the source description and it is therefore fast. This is important because a major bottleneck in automatic hardware synthesis is the time to place and route the netlist produced by a hardware compiler. Besides, such source language-guided placement and routing can preserve the inherent regularity of a bit-level regular array design. It is hence possible to produce area and time efficient designs.

The floorplanning module is divided into two parts: the global placement and routing part and the detailed placement and routing part. The former is device-independent while the latter is device-dependent. The separation between the device-independent and device-dependent parts is desirable because (a) such arrangement facilitates re-targeting: it is comparatively easy for us to build a new floorplanning module for a different device as we only need to add the device-dependent part; (b) floorplanning a complex design can be extremely difficult, and a "divide and conquer" approach should reduce the complexity of the problem; (c) some algorithms used for the Ruby sketcher (Section 3.2) can be applied to the first part of the floorplanning module; (d) such arrangement makes it possible to separately explore various device-independent and device-dependent optimisation techniques.

The viability of our floorplanning scheme has been demonstrated by a proof-of-concept compiler backend [8] customised for CAL1024 FPGAs [1], a precursor of Xilinx 6200 FPGAs. The simplicity of CAL1024 FPGAs enables us to focus on the essential tasks of mapping the high-level Ruby block descriptions into low-level device-specific FPGA cells, described using the OAL language [26]. Fig. 9 shows the overall architecture of our Ruby to CAL1024 compiler. The intermediate representation IR1 contains information about relative placement of each component, while the intermediate representation IR2 describes the dimension and connection position of each component.

The next section illustrates the system using a case study. The objective is to show how step-by-step design development can be achieved using our framework. We show the design flow of several median filter designs which includes word-level architecture development and optimisation, data refinement, bit-level optimisation and hardware compilation.

4. Case study: Median filter designs

4.1. Introduction

Median filtering is a special, but popular, case of ranked order filtering. It has been widely used in the area of digital image processing (see, for instance, [33]). One of its common applications is in edge detection algorithms for image feature extractions. Many such algorithms use the significant changes of the grey levels of pixels in an image to indicate where edges exist. Since noise causes false edges to be detected, the smoothing of images is indispensable. If a linear filter is used, it will not only remove noise, but blur the potential edges as well, and hence will result in mis-locating edges or missing them altogether. A median filter, however, does not have this problem, since it can reduce noise spikes without extensively blurring or damaging edges. Some other fields in which median filters have been successfully applied include speech signal processing [34] and data compression [2].

There are a number of reasons for choosing median filter designs to illustrate our framework. First, the implementations described in this section are complex enough to illustrate the capability of our integrated system but still simple enough to be understood. Second, median filters are a popular topic in signal processing in general and in non-linear filtering for feature detection in particular. Due to the non-linear nature of the median filter, it is highly desirable in many applications to implement median filters in the form of regular arrays for high performance and small area. Some of our designs are similar to those in [35], which have been implemented in 2 μm CMOS technology. Finally, we show that our median filter designs can be very efficiently implemented in FPGAs.

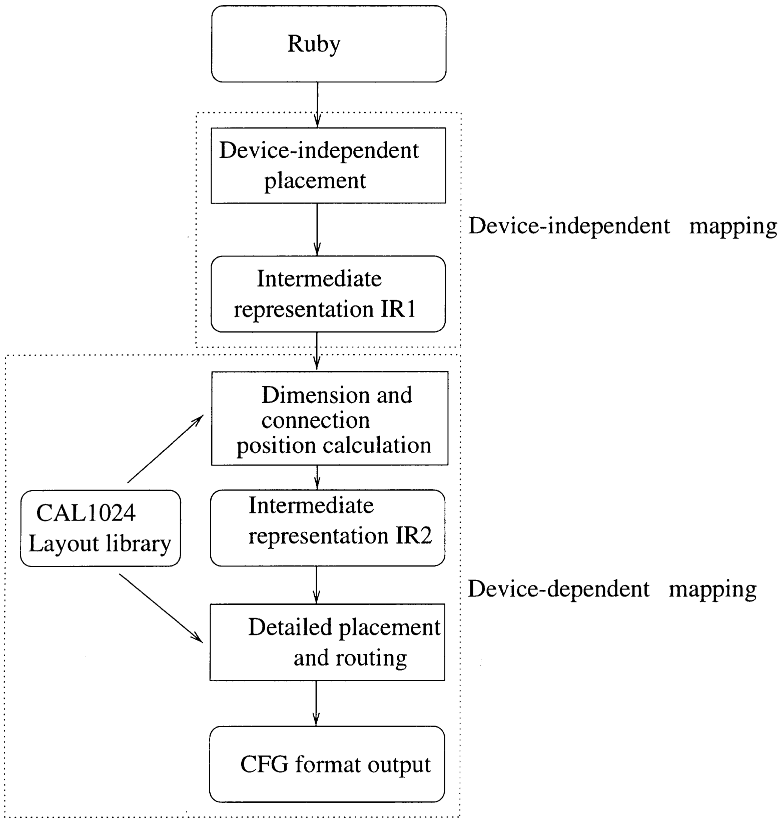

Ruby
Device-independent placement
Intermediate representation IR1
Device-independent mapping
Dimension and connection position calculation
CAL1024 Layout library
Intermediate representation IR2
Device-dependent mapping
Detailed placement and routing
CFG format output

Fig. 9. Overall structure of the Ruby to CAL1024 compiler.

4.2. Specification

We shall restrict our discussion here to one dimensional median filtering, although a two-dimensional separated median filter can be directly performed using two one-dimensional median filters. More general descriptions on median filtering can be found, for example, in [12,33].

To specify the median filtering operation, let us assume that a filter window is slid along the input sequence to be filtered. Fig. 10 shows the case when the filter size is 5. At each filter window position, the elements inside the window are sorted and the median element is extracted as output. In each cycle when the filter window progresses, one new element enters the filter window and one element that has been kept in five successive windows is deleted. Since there is only one element which is different between two successive window positions, the sorting result of the current window can be exploited to simplify the sorting process of the next window. For instance, in Fig. 10, given that elements in window *W*2 have already been sorted, the next window, *W*3, can be obtained from *W*2 simply by deleting element *X*2 and inserting element *X*7 so that the resulting sequence is still ordered.

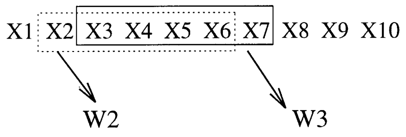


Fig. 10. Two successive filter windows (filter size = 5).

Based on the above observation, we now describe a median filter as a finite state machine which involves an ordered state $s = \langle s_0, s_1, s_2, s_3 \rangle$. Given an input i where $s_2 < i \leqslant s_3$, an ordered array $u = \langle s_0, s_1, s_2, i, s_3 \rangle$ is produced by inserting i into the state s. From the ordered array, the median filter outputs the median s_2 and generates a new state $s' = \langle s'_0, s'_1, s'_2, s'_3 \rangle$ by deleting the element which has been kept in the filter for five successive states.

4.3. Parametrised description

We need three blocks to implement the architecture Mstl that we adopt for the state transition logic described above: InstSortB, LocatorB and DeleterB, which are stacked on top of one another (Fig. 11). In addition, there are some interface and rewiring circuits Intfi and Intfo at the input and output, respectively.

Before we get into the detailed Ruby expressions, let us give a brief explanation about the architecture of Mstl so that readers can get a feel for our design without following the Ruby descriptions. The components in Mstl and their input and output signals are described in Tables 1 and 2. **Intfi** takes a new input i at each clock cycle and passes it to InstSortB; similarly, it connects the signal v from LocatorB to DeleterB. It also outputs a, the element to be deleted, and b, a boolean signal. b is defined as follows: given that v is the minimum element of u, $b = 0$ if $v = a$; otherwise $b = 1$. Intfo outputs the median o from InstSortB and discards signals a, t and d, which are outputs from blocks LocatorB and DeleterB. Block InstSortB performs insertion sorting and generates the median. It takes i, the new input element, and s, the current state which is sorted in ascending order, and then generates a sorted array u and the median o. LocatorB locates the element to be deleted within the current state. It takes as input u, the current state containing a sorted array of data, and a, the element to be deleted, and the boolean b. In addition to outputting v, a, and t, a boolean indicating whether a is the maximum value in u, LocatorB generates r, an array of boolean and element pairs, such that the boolean value true indicates that the element to be deleted has not been found. Otherwise, the element to be deleted has been found. DeleterB deletes the element located by LocatorB. It takes r, the array of boolean and element pairs, and v, the minimum element of array u, and generates the new state s' and d, the element to be deleted.

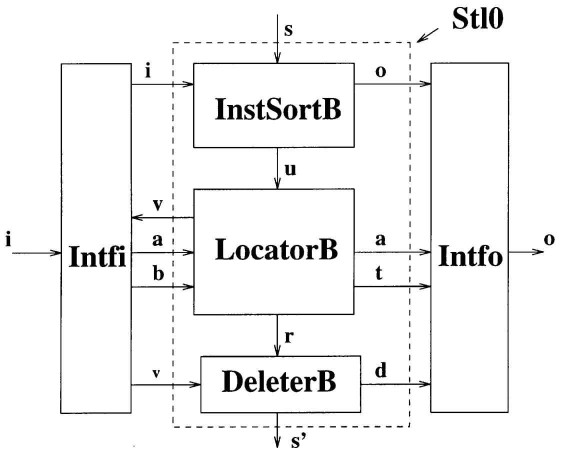


Fig. 11. Block diagram of the state transition logic Mstl.

Let us now derive a parametrised description for the median filter. The block Mstl can be captured in Ruby as

$$\mathrm{Mstl} = \mathsf{fst}\ \mathrm{Intfi};\ \mathrm{Stl0};\ \mathsf{snd}\ \mathrm{Intfo}, \qquad (29)$$

where

$$\mathrm{Stl0} = (\mathrm{DeleterB} \updownarrow \mathrm{LocatorB}) \updownarrow \mathrm{InstSortB}.$$

The definition of ";", fst, snd, $\updownarrow$ etc can be found in Section 2. The input interface Intfi obtains a new input i, records the element a which will be deleted, and rearranges wires. It is defined by

Table 1
The components in the block Mstl in Fig. 11

Block	Parameter	Function
Mstl	n: the size of filter window	Median filtering
Intfi	n: number of shift register	Input interface
InstSortB	n: number of repeating cell	Sorting and median extraction
LocatorB	n: number of repeating cells	Location of the element to be deleted
DeleterB	n: number of repeating cells	Deletion of the element located
Intfo		Output interface

Table 2
Input/output signals of each block in **Mstl** in Fig. 11

Block	Signal	Type	from/to	Signal function
Mstl	i	in	external	External input of the filter
	o	out	external	External output of the filter
Intfi	i	in	external	External input of the filter
	v	in	LocatorB	Minimum element
	i	out	InstSortB	Filter input
	a	out	LocatorB	Element to be deleted
	b	out	LocatorB	Boolean indicating if a equals v
	v	out	DeleterB	Minimum element
InstSortB	i	in	Intfi	External input of the filter
	s	in	external	Current state
	u	out	LocatorB	Sorted element array
	o	out	Intfo	Median element
LocatorB	u	in	InstSortB	Sorted element array
	a	in	Intfi	Element to be deleted
	b	in	Intfi	Boolean indicating if a equals v
	v	out	Intfi	Minimum element of array u
	r	out	DeleterB	Array of boolean-element pairs
	a	out	Intfo	Element to be deleted
	t	out	Intfo	Boolean indicating if the maximum element equals a
DeleterB	r	in	LocatorB	Array of boolean-element pairs
	v	in	Intfi	Minimum element of array u
	s'	out	external	New state (the next state)
	d	out	Intfo	The element to be deleted
Intfo	o	in	InstSortB	Median element
	a	in	LocatorB	Element to be deleted
	t	in	LocatorB	Boolean indicating if the maximum element equals a
	d	in	DeleterB	Deleted element

$$i \text{ Intfi } \langle\langle v, \langle\langle b, a\rangle, v\rangle\rangle, i\rangle. \tag{30}$$

Given that $n+1$ is the filter size, and $i(t)$ and $a(t)$ are both time sequences, we have $a(t) = i(t-n)$ and $b = a$ xor v.

A parametrised implementation of **Intfi** is given by

$$\text{Intfi} = \text{fork} \ ; \ \mathsf{fst}\ (D^n); \ \text{lstcomp}, \tag{31}$$

where $\langle x, y\rangle$ lstcomp $\langle\langle z, \langle\langle s, x, \rangle\rangle, y\rangle$ and $s = x$ xor z.

The output interface Intfo outputs the median o and discards a, t and d which are outputs from blocks LocatorB and DeleterB. Intfo is defined as

$$\langle\langle d, \langle t, a\rangle\rangle, o\rangle \text{ Intfo } o. \tag{32}$$

Block Stl0 is the core part of the state transition logic which is made up of three sub-blocks: InstSortB, LocatorB and DeleterB.

InstSortB takes as input the current state s, which is sorted in ascending order and the input element i. It produces an array u sorted in ascending order from s and i and the median o. For instance, given that $s = \langle s_0, s_1, s_2, s_3\rangle$ and $s_2 < b \leqslant s_3$, InstSortB will insert b in s such that s is still sorted:

$$\langle b, \langle s_0, s_1, s_2, s_3\rangle\rangle \text{ InsertSortB } \langle\langle s_0, s_1, s_2, b, s_3\rangle, s_2\rangle. \tag{33}$$

A parametrised version of InstSortB is

$$\text{InsertSortB} = \text{InsertSort} \ ; \ \text{apr}_n \ ; \ \mathsf{map}_{(n+1)}\text{fork} \ ; \ (\text{zip}_{(n+1)})^{-1} \ ; \ \mathsf{snd}\ \text{MedianExtract}, \tag{34}$$

where zip and apr are Ruby functions for regrouping wires. For instance, $\langle\langle x_1,x_2,x_3\rangle, \langle y_1,y_2,y_3\rangle\rangle$ zip$_3$ $\langle\langle x_1,y_1\rangle,\langle x_2,y_2\rangle,\langle x_3,y_3\rangle\rangle$, and $\langle\langle x_1,x_2,x_3\rangle,y\rangle$ apr$_3$ $\langle x_1,x_2,x_3,y\rangle$. InsertSort is an insertion sorter which is given by

$$\text{InsertSort} = \mathsf{row}_n \text{ scell}, \tag{35}$$

where $\langle t,u\rangle\text{scell}\langle\min(t,u),\max(t,u)\rangle$.

MedianExtract extracts the median from a sorted array, which is described for $n = 4$ by

$$\langle s_0,s_1,s_2,s_3,s_4\rangle \text{ MedianExtract } s_2. \tag{36}$$

A parametrised version implementing MedianExtract is

$$\text{MedianExtract} = (\text{flatr}_{(n+1)})^{-1} \ ; \ \pi_2^{((n+1)/2)} \ ; \ \pi_1, \tag{37}$$

where flatr is a Ruby function for flattening a wiring structure. For instance, $\langle x_1,\langle x_2,\langle x_3,\langle x_4,\langle x_5,\langle\rangle\rangle\rangle\rangle\rangle\rangle$ flatr$_5$ $\langle x_1,x_2,x_3,x_4,x_5\rangle$.

The block LocatorB takes as inputs the sorted array u, the element to be deleted a, and a boolean b indicating if $a = v$, where v is the minimum element of u, and generates v, r, a, and t shown in Table 2. Let us look at an example. Assume $u = \langle 3,5,6,7,10\rangle$, $a = 5$. Then $b =$ 'T', $t =$ 'F', and the array of boolean and element pairs $r = \langle\langle T,5\rangle,\langle F,6\rangle,\langle F,7\rangle,\langle F,10\rangle\rangle$.

A parametrised version implementing LocatorB is shown below

$$\begin{aligned}\text{LocatorB} = {} & \mathsf{snd}\ (\text{apr}_n)^{-1} \ ; \ (\text{Locator} \\ & \leftrightarrow (\langle\langle\langle x,y\rangle,z\rangle,z\rangle\$\text{wire}\langle x,y\rangle) \ ; \ \text{lsh}),\end{aligned} \tag{38}$$

where Locator (Fig. 13) contains a row of lctcell,

$$\text{Locator} = \mathsf{row}_n \text{ lctcell}, \tag{39}$$

and lctcell is defined by

$$\begin{aligned}&\langle\langle\langle a,b\rangle,c\rangle,d\rangle \text{ lctcell } \langle\langle e,f\rangle,\langle\langle g,h\rangle,i\rangle\rangle \iff (c \\ &\quad = d) \wedge (h = b) \wedge (e = a) \wedge (f = i) \wedge (g \\ &\quad = (a \text{ and } (b \text{ xor } i))).\end{aligned}$$

The block DeleterB takes as inputs r, the array of boolean and element pairs, and v, the minimum element of array u. It generates the next state s' and d, the element to be deleted. The new state s' differs from array u only in that s' does not contain the element which has been kept in five previous states.

DeleterB can be realised as a row of cell dltcell

$$\text{DeleterB} = \mathsf{row}_n \text{ dltcell}, \tag{40}$$

where

$$\langle x,\langle s,y\rangle\rangle \text{ dltcell}\langle m,n\rangle \iff \begin{cases} (m = x) \wedge (n = y) & \text{if } s = 1, \\ (m = y) \wedge (n = x) & \text{otherwise.}\end{cases}$$

Composing the above blocks together, the complete median filter can now be expressed by the following state machine

$$\begin{aligned} M_0 &= \text{loop}(\text{Mstl} \ ; \ \mathsf{fst}(\mathsf{map}_n D)) \\ &= \text{Intfi} \ ; \ \text{loop}(\text{Stl0}; \mathsf{fst}(\mathsf{map}_n D)) \ ; \ \text{Intfo} \\ &= \text{Intfi} \ ; \ \text{Mcore0} \ ; \ \text{Intfo},\end{aligned}$$

where

$$\text{Mcore0} = \text{loop}(\text{Stl0} \ ; \ \mathsf{fst}(\mathsf{map}_n D)). \tag{41}$$

$$\text{Stl0} = (\text{DeleterB} \updownarrow \text{LocatorB}) \updownarrow \text{InstSortB}. \tag{42}$$

Note that the correct operation of M_0 requires the feedback latches to be initialised with ∞.

At this stage of the design development, the sketcher described in Section 3.2 proves to be useful for inspecting the architecture of the design, and the behaviour of the design is readily validated using our simulation and visualisation facilities. Design validation is described in the following section.

4.4. Design validation

To study the behaviour of the median filter M_0, one can use the visualiser to examine Intfi, Intfo, InstSortB, LocatorB and DeleterB separately and then their composition. For brevity, we shall only present the visualisation of the integral design M_0 here.

To visualise M_0, we supply simulation data for the input i at each clock cycle, and the visualiser can display the architecture of M_0 and the values of user-selected wires. Fig. 12 shows a frame from an animation sequence when a number 8 is inserted into the median filter. The animation se-

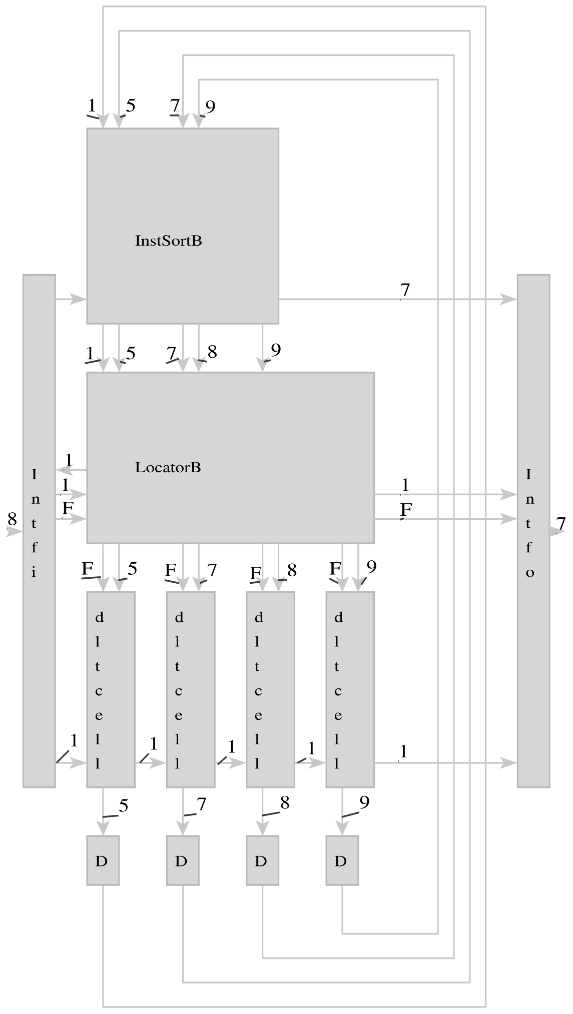


Fig. 12. A frame extracted from a design animation sequence. 8 is input. As shown is the frame, current state $s = \langle 1,5,7,9 \rangle$. After inserting the current input 8, InstSortB generates a sorted array $u = \langle 1,5,7,8,9 \rangle$ and outputs median 7. Before this clock cycle, 1, 9, 5 and 7 have already been input in order, hence 1 is the element to be deleted at the current clock cycle. Therefore, the new state s' equals $\langle 5,7,8,9 \rangle$.

quence shows that the design M_0 behaves as we expected.

4.5. Word-level optimisation

In the preceding subsection, a parametrised architecture M_0 of median filter is obtained and validated using our system. The next task is to optimise M_0 to increase regularity and to produce a systolic implementation by pipelining. While the algebraic laws of Ruby facilitate the systematic transformation and proof of correctness, we shall also make use of design diagrams for obtaining insights into our optimisation. As described in Section 4.3, the sketcher described in Section 3.2 can be employed in early stages of development to rapidly generate diagrams of Ruby designs. The behaviour of the transformed designs can be studied through design simulation using the simulator or the visualisation system described in Section 3.3.

To simplify our transformation, we shall at first ignore the input and output interfaces but focus on the structure of Mcore0 depicted in Fig. 13. The input and output signals of the repeating cells are described in Table 3. Our basic idea of optimising Mcore0 is to decompose the state transition logic containing three repeating structures, InsertSort, Locator and Deleter (Fig. 13), into a cascade of state machines forming a single repeating structure, Mcorecell (Fig. 14), and introducing pipelining to increase the performance of the design.

Let us now examine the architecture of Mcore0. Clearly, blocks Deleter, Locator and InsertSort can be merged to form a repeating structure Mcore1 if the block MedExtract is ignored and the maximum output of InsertSort is rewired through the output port. From the algebraic law

$$(\mathsf{row}_n\ Q) \updownarrow (\mathsf{row}_n\ R) = \mathsf{row}_n\ (Q \updownarrow R),$$

we obtain

$$\begin{aligned} Mcore1 = \mathrm{loop}(((&\mathsf{row}_n, dltcell) \\ &\updownarrow (\mathsf{row}_n\ \mathrm{lctcell})) \updownarrow (\mathsf{row}_n\ \mathrm{scell}); \\ &\mathsf{fst}(\mathsf{map}_n\ D)) \qquad (43) \\ = \mathrm{loop}(&\mathsf{row}_n\ \mathrm{Stl1}), \qquad (44) \end{aligned}$$

where

$$\mathrm{Stl1} = (\mathrm{dltcell} \updownarrow \mathrm{lctcell}) \updownarrow \mathrm{scell}. \qquad (45)$$

Now let us apply the theorem for state machine decomposition [21],

$$\mathrm{loop}(\mathsf{row}_n\ R) = (\mathrm{loop}\ R)^n, \qquad (46)$$

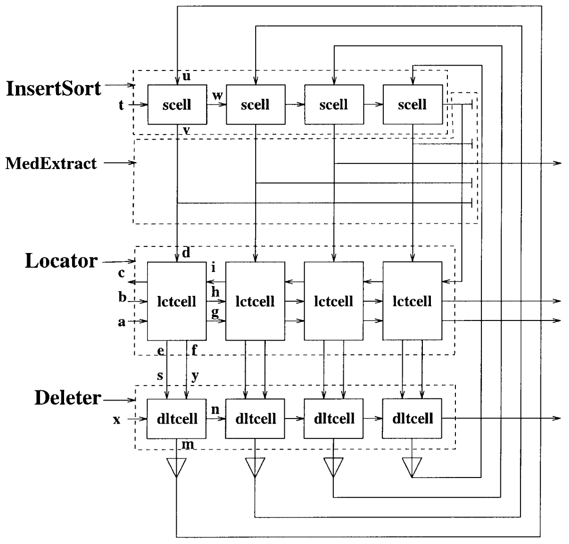


Fig. 13. Block diagram of Mcore0 (filter size = 5).

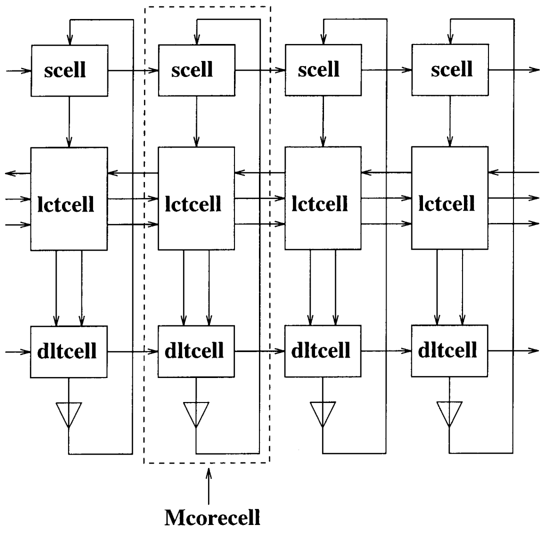


Fig. 14. Block diagram of the state machine Mcore1 (filter size = 5).

we obtain a regular state machine (Fig. 14)

$$\mathrm{Mcore1} = (\mathrm{Mcorecell})^n, \tag{47}$$

where

$$\mathrm{Mcorecell} = \mathrm{loop}((\mathrm{dltcell} \updownarrow \mathrm{lctcell}) \updownarrow \mathrm{scell}\ ;\ \mathrm{fst}\ D). \tag{48}$$

To generate the medians from the state machine, we need to output the sorted array, from which the median can be extracted. This can be achieved by slightly modifying the architecture Mcorecell obtained above while still retaining its regular feature (Fig. 15).

Table 3
Input/output signals of the repeating cells in Mcore0 in Fig. 13

Block	Signal	Type	from/to	Signal function
scell (sorting cell)	t	in	filter input interface	External input of the filter
	u	in	current state	An element of current state
	v	out	lctcell	$\min(t, u)$
	w	out	next scell cell	$\max(t, u)$
lctcell (locating cell)	a	in	scell	Control signal for deleting cell
	b	in	next lctcell	The element to be deleted
	d	in	filter input interface	An element from the sorted array
	i	in	filter input interface	An element from the sorted array
	c	out	filter input interface	Passthrough for signal d
	e	out	next lctcell	Passthrough for signal a
	f	out	next lctcell	Passthrough for signal i
	g	out	dltcell	$g = a$ and (b xor i)
	h	out	dltcell	Passthrough for signal b
dltcell (deleting cell)	y	in	lctcell	An element of the sorted array
	s	in	lctcell	Control signal for deleting operation
	x	in	filter input interface	The minimum element of the sorted array
	m	out	a register	$m = x$ if $s = 1$; $m = y$ otherwise.
	n	out	next dltcell	$n = y$ if $s = 1$; $n = x$ otherwise.

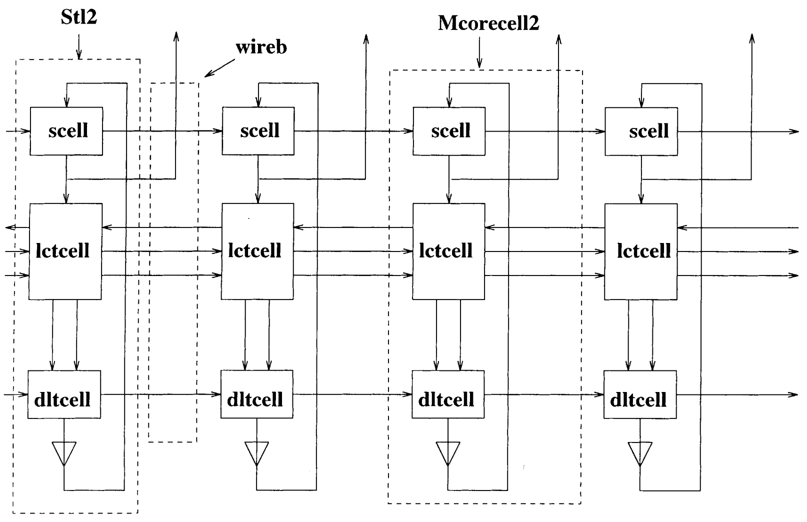


Fig. 15. Block diagram of the state machine Mcore2 (filter size = 5).

The Ruby expression for capturing the architecture shown in Fig. 15 is given by

$$\mathrm{Mcore2} = \mathsf{rdl}_n\ \mathrm{Mcorecell2}, \tag{49}$$

where

$$\mathrm{Mcorecell2} = \mathsf{fst}\ \mathrm{Stl2}\ ;\ \mathrm{wireb}, \tag{50}$$

and

$$\mathrm{Stl2} = (\mathrm{dltcell} \updownarrow \mathrm{lctcell}) \updownarrow (\mathrm{scell}\ ;\ \mathsf{fst\ fork}\ ;\ \mathrm{lsh}),$$

$$\langle\langle\langle x, \langle\langle y, z\rangle, s\rangle\rangle, \langle u, v\rangle\rangle, t\rangle\ \mathrm{wireb}\ \langle\langle x, \langle\langle y, z\rangle, s\rangle\rangle, v\rangle.$$

Considering the input and output interface, we get the systolic median filter M_2

$$M_2 = [\mathrm{Inputb}, \mathrm{Medb}]\ ;\ \mathrm{Mcore2}\ ;\ \mathrm{Rightb}, \tag{51}$$

where Inputb and Rightb are interfaces, and Medb extracts medians from the sorted array. They are described in Ruby as follows:

$$\mathrm{Inputb} = \mathrm{Intfi}.$$

$$\mathrm{Medb} = ((\mathrm{flatr}_n)^{-1}\ ;\ \pi_2^{n/2}\ ;\ \pi_1)^{-1}$$

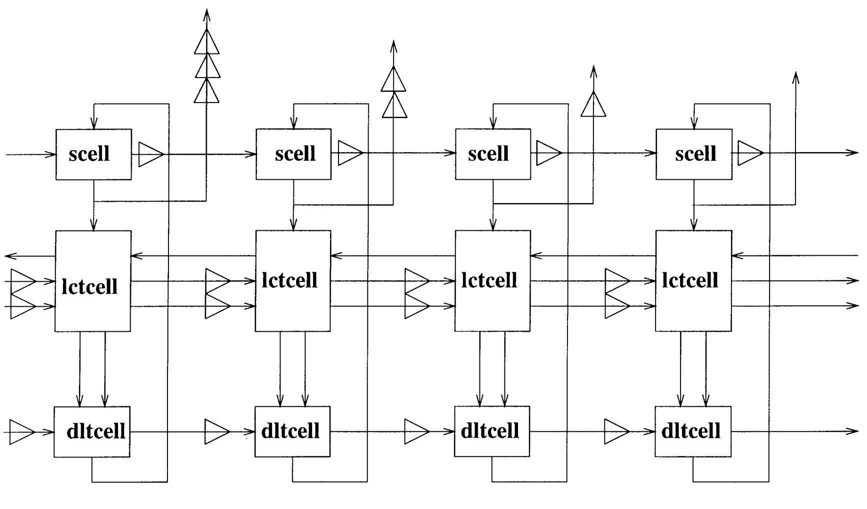


Fig. 16. Block diagram of a fully pipelined median filter (filter size = 5).

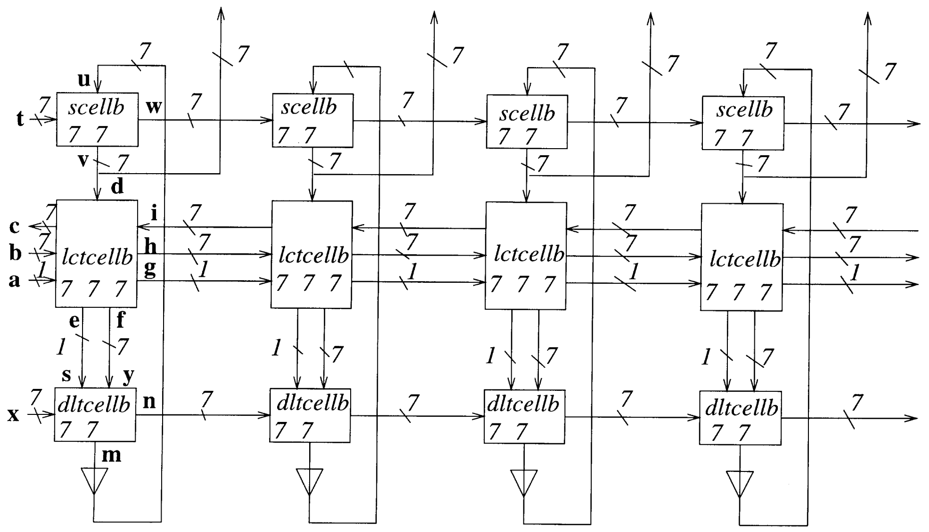


Fig. 17. Block diagram of a bit-level median filter. The interfaces are ignored for brevity.

and

$\langle\langle x,\langle\langle y,z\rangle,s\rangle\rangle,s\rangle$ Rightb $\langle\rangle$.

Here Intfi has been defined in expression (31).

Further optimisation of M_2 can be achieved by introducing various degrees of pipelining which includes registers between adjacent Mcell [22]. For instance, a fully pipelined version of implementation M_2 is shown in Fig. 16. On the top of the diagram, there are 6 registers forming a triangular-shaped array at the output; they ensure that all data at the output will emerge in the same clock cycle.

Other optimisations using transposition [20] and serialisation [21] are also possible, but we shall not go into the details here.

4.6. Data refinement and bit-level optimisation

Using the refinement system described in Section 3.4, it is straightforward to obtain an optimised bit-level version for the word-level median filter described in Eq. (51) in the preceding section. For instance, given the input elements in the range 0 to +127 (7-bit grey level image data, for example), our refinement system generates the bit-level implementation shown in Fig. 17, where the widths of data paths have been indicated. scellb$_{77}$ is the parametrised bit-level version of scell, with two parameters, both of them are 7 for this example, specifying the width of its first and second input. lctcellb$_{777}$ is the bit-level version of lctcellb with three parameters, respectively, specifying the width of its second, third and fourth inputs

Table 4
The parametrised bit-level repeating cells in Fig. 17

Block	Parameter	Function
scellb	j: the size of input t. $j = 7$ in Fig. 17. k: the size of input u. $k = 7$ in Fig. 17.	Sorting cell
lctcellb	j: the size of input b. $b = 7$ in Fig. 17. k: the size of input d. $k = 7$ in Fig. 17. l: the size of input i. $l = 7$ in Fig. 17.	Locating cell
dltcellb	j: the size of input x. $x = 7$ in Fig. 17. k: the size of input k. $k = 7$ in Fig. 17.	Deleting cell

Table 5
Input/output signals of the bit-level repeating cells in Fig. 17

Block	Signal	Type	size	from/to	Signal function
scellb	t	in	7	filter input interface	External input of the filter
	u	in	7	current state	An element of current state
	v	out	7	lctcellb	$\min(t,u)$
	w	out	7	neighbouring scellb cell	$\max(t,u)$
lctcellb	a	in	1	scellb	Control signal for deleting cell
	b	in	7	neighbouring lctcellb	The element to be deleted
	d	in	7	filter input interface	An element from the sorted array
	i	in	7	filter input interface	An element from the sorted array
	c	out	7	filter input interface	Passthrough for signal d
	e	out	1	neighbouring lctcellb	Passthrough for signal a
	f	out	7	neighbouring lctcellb	Passthrough for signal i
	g	out	1	dltcellb	$g = a$ and (b xor i)
	h	out	7	dltcellb	Passthrough for signal b
dltcellb	y	in	7	lctcellb	An element of the sorted array
	s	in	1	lctcellb	Control signal for deleting operation
	x	in	7	filter input interface	The minimum element of the sorted array
	m	out	7	a register	$m = x$ if $s = 1$; $m = y$ otherwise.
	n	out	7	neighbouring dltcellb	$n = y$ if $s = 1$; $n = x$ otherwise.

(labeled as b, d and i, respectively, in Fig. 17). Similarly, dltcellb$_{77}$ is the parametrised bit-level version of **dltcell** and its first parameter describes the first input (labelled as x in Fig. 17) and the second parameter specifies the third input (labelled as y in Fig. 17). The parametrised bit-level repeating cells and their input and output signals are described in Tables 4 and 5. These cells can be implemented using hardware libraries in a specific technology (see Section 4.7).

Although the hardware libraries used for implementing scellb, lctcellb and dltcellb can be highly-optimised and technology-specific, the overall implementation using FPGAs is usually inefficient due to the wiring congestion between the connected blocks. It is desirable to further optimise the design at the bit-level. The objective of such an optimisation is to develop bit-level cells which can be replicated to form Mcore2.

Such cells are shown in Fig. 18. A column of cell A implements Intfi (Fig. 11), which serves as the input port of the filter. It is composed of a bit-wise shift register array and some other logic gates for comparison. Note that for the cell at the most significant bit position, the input K_8 should be hardwired to logic zero for initialisation. The number of registers the shift register contains depends on the window size of the filter. For instance, for a median filter with size 5, there should be 4 registers. Block Stl2 (Fig. 15) is implemented using a column of cell B, which is made up of a bit-wise comparator COMP and two two-way multiplexers MUX1 and MUX2. An init wire and an OR gate are introduced for presenting the latch to a desired value. The purpose is to initialise the latch to logic one so that each latch is initialised to the maximum number that the filter can input. Also, for the cell at the most-significant bit position, the input I_8 should be hardwired to logic zero, while for the cell at the least-significant bit position, C_0 should be wired to logic one. C and D are the periphery cells for signal propagation.

The above bit-level cells can be used to implement a median filter with any filter size and any number of input bits. As an example, a median filter with filter size 5 and 7-bit input is shown in Fig. 19.

Using the Ruby simulator and the visualiser, it is very straightforward to validate the bit-level design against the word-level design. One can also use algebraic transformations to verify that the bit-level design is a faithful implementation of the word-level description.

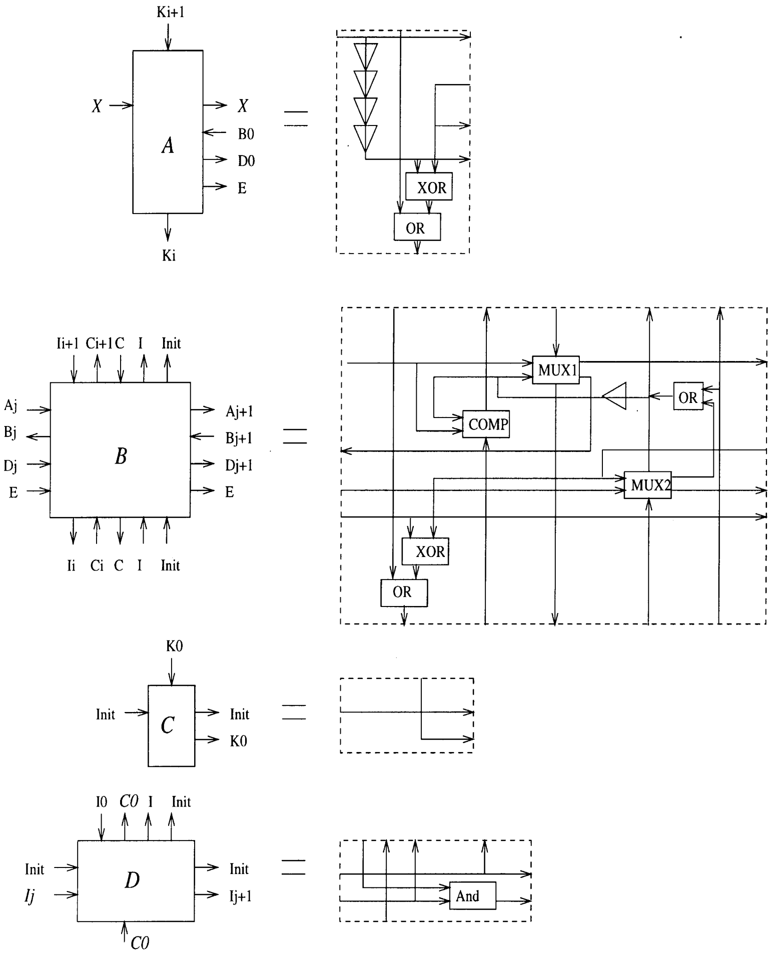


Fig. 18. Designs of bit-level cells *A*, *B*, *C* and *D*.

4.7. FPGA implementation

With our hardware compiler, we can directly compile the bit-level Ruby description of the median filter into hardware such as FPGAs. For the highest performance of a design, however, it is desirable to exploit device-specific features. For a particular implementing technology we may, for instance, manually place and route the repeating cells to achieve the optimal layout, because any inefficiency in a repeating unit will be amplified many times for the whole design.

Our compiler takes only a few seconds to generate the CAL-based implementation of a median filter with a 5-element window and 8-bit input. This design, with a compact implementation of the repeating cell, is shown in Fig. 20. This example illustrates that the regularity information from the Ruby source code can be used to produce a compact layout. A hardware implementation based on this median filter design has been used to filter noise and locate edges from range data generated by range sensors [11].

5. Conclusion

In this paper, we have presented an overview of an integrated system for developing regular array designs. It has been demonstrated that the system supports a simple and powerful notation for representing regular array designs, and both the

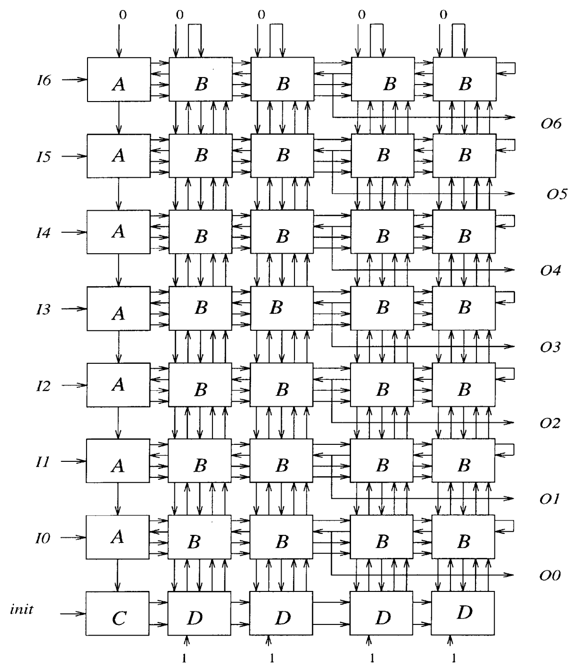


Fig. 19. A bit-level median filter with window size 5 and 7-bit input. $I0, \ldots, I6$ are the inputs while $O0, \ldots, O6$ are the outputs, and init is used for initialisation.

architecture and the behaviour of an array can be captured in a single parametrised description. Various facilities have been developed and integrated for refining, sketching, simulating, optimising, visualising and compiling regular array designs. We have used several median filter designs to illustrate our system. The following steps sum up the procedure for implementing a specific algorithm using our integrated system.

- Develop a hardware architecture for the given algorithm.

Initially, the architecture may not be efficient but should be obviously correct. More efficient designs can be obtained from the initial design by optimisation.

- Capture the architecture as a parametrised expression in Ruby.

This step may require parametrising the design description, and using available library blocks.

- Check the correctness of the design by formal verification, numerical and symbolic simulation, and design visualisation.

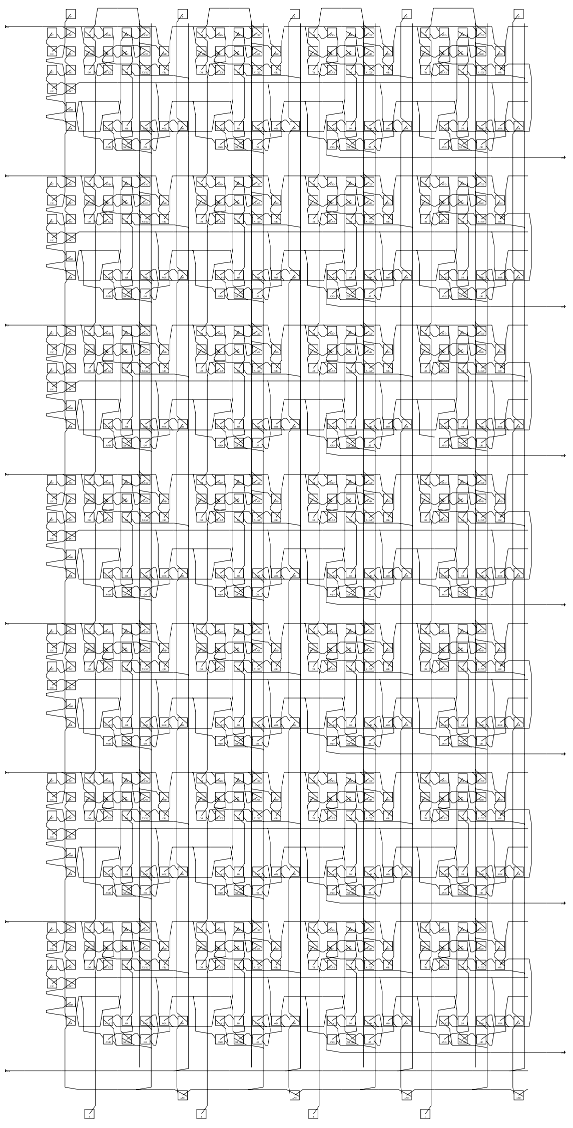


Fig. 20. CAL implementation of a median filter (input width = 8 bit and window size = 5).

This step is to ensure that the design conforms to the intended behaviour.

- Apply correctness-preserving transformations to optimise the design to satisfy user constraints, like employing pipelining [22] to increase parallelism, or serialisation [21] to reduce size, or decomposing state machines to increase regularity.

It is often helpful to use design diagrams as a vehicle to gain insights into design transformations.

In such cases, the design sketcher should prove to be a useful tool in rapidly generating design diagrams.

- Develop the most efficient implementation of the repeating units before hardware compilation, using device-specific descriptions like OAL if necessary.

The system has been used in developing a number of regular array designs, including a systolic convolver [11], a systolic priority queue [9], a systolic sorter [10], a sine calculator [8], and in developing reconfigurable libraries for FPGAs [24,25]. It has also been used in producing implementations partly in hardware and partly in software [23].

There are a number ways in which our system can be refined and enhanced. The current version of the refiner, for example, deals only with constrains of maximum and minimum values of inputs for a word-level circuit. Future work to enhance it includes extending our approaches to take into consideration other kinds of constraints such as critical path delay, latency or size.

The sketcher can automatically produce design diagrams from Ruby programs. A tool capable of producing Ruby programs from design diagrams would be very useful, because schematic representations of designs produced by other design system, such as Viewlogic and Mentor Graphics, can then be automatically incorporated in our design framework for further development.

Both the refiner and the hardware compiler rely on various kinds of parametrised bit-level libraries to facilitate efficient design. These include both device-dependent and device-independent libraries. It is desirable to develop a wide variety of implementations such as digit-serial and pipelined designs to facilitate selecting and reusing the most appropriate ones.

Acknowledgements

We thank Penny Probert for her suggestions and encouragement. The support of the Sino-British Friendship Scholarships Foundation, Lady Margaret Hall, Xilinx Inc., and the UK Engineering and Physical Sciences Research Council (Grant number GR/24366, GR/54356 and GR/59658) is gratefully acknowledged.

References

[1] Algotronix Ltd, CHS2x4 Custom Computer, 1992.

[2] G.A. Arce, N.C. Gallagher, State description for the root signal set of median filters, IEEE Transactions on Acoustics, Speech and Signal Processing ASSP–30, December, 1982, pp. 894–902.

[3] R. Bird, Introduction to Functional Programming using Haskell, Second ed., Prentice-Hall, Europe, 1998.

[4] P. Bjesse, K. Claessen, M. Sheeran, S. Singh, Lava: hardware design in Haskell, ACM international Conference on Functional Programming'98, ACM Press, New York, 1998.

[5] C.J. Block, A model for representing Ruby circuits, in: Proceedings of the Glasgow Workshop on Functional Programming, 1996, http://www.dcs.gla.ac.uk/research/fpga/papers/ps/Model.ps.

[6] D.R. Coelho, The VHDL Handbook, Kluwer, Dordrecht, 1989.

[7] F. de Dinechin, Libraries of schedule-free operators in Alpha, in: Proceedings of the International Conference on Application Specific Array Processors, IEEE Computer Society Press, Silver Spring, MD, July 1997.

[8] S. Guo, Techniques and tools for developing Ruby design, D.Phil. thesis, Computing laboratory, University of Oxford, 1997.

[9] S. Guo, W. Luk, Compiling Ruby into FPGAs, in: W. Moore, W. Luk (Eds.), Field Programmable Logic and Applications, Lecture Notes in Computer Science, vol. 975, Springer, Berlin, 1995, pp. 188–197.

[10] S. Guo, W. Luk, Producing design diagrams from declarative descriptions, in: S. Yang, J. Zhou, C. Li (Eds.), Proceedings of the Fourth International Conference on CAD/CG, SPIE, 1995, pp. 1084–1093.

[11] S. Guo, W. Luk, P. Probert, Developing parallel architectures for range and image sensors, in: Proceedings of the IEEE Internatinal Conference on Robotics and Automation, IEEE Computer Society Press, Silver Spring, MD, 1994, pp. 2205–2210.

[12] T.S. Huang, G.T. Yang, G.Y. Tang, A fast two-dimensional median filtering algorithm, IEEE Transactions on Acoustics, Speech and Signal Processing ASSP–27, February 1979, pp. 13–38.

[13] G. Jones, M. Sheeran, Circuit design in ruby, in: J. Staunstrup (Ed.), Formal Methods for VLSI Design, North-Holland, Amsterdam, 1990, pp. 13–70.

[14] G. Jones, M. Sheeran, Relations and refinement in circuit design, in: C. Morgan, J. Woodcock (Eds.), 3rd Refinement Workshop, Springer Workshops in Computing, 1991.

[15] G. Jones, M. Sheeran, A certain loss of identity, in: J. Launchbury, P. Sansom (Eds.), Glasgow Functional Programming Workshop, Springer, Berlin, 1992.

[16] I. Kostarnov, S. Morley, J. Osmany, C. Soloman, A reconfigurable approach to low cost media processing, in: W. Luk, P.Y.K. Cheung, M. Glesner (Eds.), Field-Programmable Logic and Applications, Lecture Notes in Computer Science, vol. 1304, Springer, Berlin, 1997, pp. 79–90.

[17] S.Y. Kung, VLSI Array Processors, Prentice-Hall, Englewood Cliffs, NJ, 1988.
[18] Y. Li, M. Leeser, HML: an innovative hardware description language and its translation to VHDL, in: Proceedings of the CHDL'95, 1995.
[19] W. Luk, Analysing parametrised designs by non-standard interpretation, in: S.Y. Kung, E. Swartzlander, J.A.B. Fortes, K.W. Przytula (Eds.), Proceedings of the International Conference on Application-Specific Array processors, IEEE Computer Society Press, Silver Spring, MD, 1990, pp. 133–144.
[20] W. Luk, Optimising designs by transposition, in: G. Jones, M. Sheeran (Eds.), Design Correct Circuits, Springer, Berlin, 1991, pp. 332–354.
[21] W. Luk, Systematic serialisation of array-based architectures, Integration (the VLSI Journal) 14 (3) (1993) 333–360.
[22] W. Luk, Systematic pipelining of processor arrays, in: G.M. Megson (Ed.), Transformational Approaches to Systolic Design, Chapman and Hall Parallel and Distributed Computing Series, 1994, pp. 77–98.
[23] W. Luk, T. Wu, Towards a declarative framework for hardware-software codesign, in: Proceedings of the Third International Workshop on Hardware/Software Codesign, IEEE Computer Society Press, Silver Spring, MD, 1994, pp. 181–188.
[24] W. Luk, S. Guo, N. Shirazi, N. Zhuang, A framework for developing parametrised FPGA libraries, in: R.W. Hartenstein, M. Glesner (Eds.), Field-Programmable Logic, Smart Applications, New Paradigms and Compilers, Lecture Notes in Computer Science, vol. 1142, Springer, Berlin, 1996, pp. 24–33.
[25] W. Luk, S. Guo, Visualising reconfigurable libraries for FPGAs, in: Proceedings of the 31 Asilomar Conference on Signals, Systems, and Computers, IEEE Computer Society, Silver Spring, MD, 1998, pp. 389–393.
[26] W. Luk, I. Page, Parametrising designs for FPGAs, in: W. Moore, W. Luk (Eds.) , FPGAs, Abingdon EE&CS Books, 1991, pp. 284–295.
[27] W. Luk, G. Jones, M. Sheeran, Computer-based tools for regular array designs, in: J. McCanny, J. McWhirter, E. Swartzlander (Eds.), Systolic Array Processors, Prentice-Hall, Englewood Cliffs, NJ, 1989, pp. 589–598.
[28] G.M. Megson (Ed.), Transformational Approaches to Systolic Design, Parallel and Distributed Computing Series, Chapman & Hall, 1994.
[29] J.K. Ousterhout, Tcl and Tk toolkit, Addison-Wesley professional computing series, 1996.
[30] L.C. Paulson, ML for the Working Programmer, Cambridge University Press, Cambridge, 1991.
[31] D.L. Perry, VHDL, McGraw-Hill, New York, 1991.
[32] S.L. Peyton Jones, The Implementation of Functional Programming Languages, Prentice-Hall, Englewood Cliffs, NJ, 1987.
[33] I. Pitas, Digital Image Processing Algorithms, Prentice-Hall, Englewood Cliffs, NJ, 1993.
[34] L.R. Rabiner, M.R. Sambur, C.E. Schmidt, Applications of nonlinear smoothing algorithm to speech processing, IEEE Transactions on Acoustics, Speech and Signal Processing, 1975.
[35] R. Rocella, R. Saletti, 70 MHz 2 μm CMOS bit-level systolic array median filter, IEEE Journal of Solid-State Circuits 28 (5) (1993) 530–535.
[36] R. Sharp, O. Rasmussen, Transformational rewriting with Ruby, in: Computer Hardware Description Languages and Their Applications (CHDL'93), Elsevier, Amsterdam, 1993, pp. 243–360.
[37] R. Sharp, O. Rasmussen, The T-Ruby design system, Formal Methods in System Design 11 (3) (1997) 239–264.
[38] M. Sheeran, Ruby–a language of relations and high-order functions, in: G. Birtwistle (Ed.), Proceedings of the Third Banff workshop on hardware verification, Springer, Berlin, 1990.
[39] S. Singh, Architectural descriptions for FPGA circuits, in: Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines, IEEE Computer Society Press, Silver Spring, MD, 1995.
[40] H. Verge, C. Mauras, P. Quinton, The ALPHA language and its use for the design of systolic arrays, Journal of VLSI Signal Processing 3 (1991) 173–182.
[41] D. Wilde, O. Sie, Regular array synthesis using Alpha, in: Proceedings of the International Conference on Application-specific Array Processors, IEEE Computer Society Press, Silver Spring, MD, 1994, pp. 200–211.

Shaori Guo received his B.Eng. and M.Eng. degrees in Electronic Engineering from the University of Electronic Science and Technology of China in 1985 and 1988, respectively, and his D.Phil. in Computer Science in 1997 from Oxford University, England. From 1996 to 1998, he worked as a research associate in Department of Computing, Imperial College, University of London. He joined Philips Semiconductors in Southampton in 1998 as a senior IC development engineer, and has been working for Philips Semiconductors in Sunnyvale, California since May, 1999. His current research involves methodologies and techniques for efficient architecture modelling, and specification and design of high performance video and telecommunications IC's.

Wayne Luk is a member of academic staff in Department of Computing, Imperial College, University of London. His research interests include theory and practice of customising hardware and software for specific application domains, such as graphics and image processing, multimedia, and communication. His current work involves high-level compilation techniques and tools for parallel computers and embedded systems, particularly those containing reconfigurable devices such as field-programmable gate arrays. He received his M.A., M.Sc., and D.Phil. in engineering and computing science from University of Oxford.