Compiling Ruby into FPGAs

Shaori Guo[1] and Wayne Luk[2]

[1] Computing Laboratory, Oxford University, Parks Road, Oxford OX1 3QD, UK
[2] Department of Computing, Imperial College, London SW7 2BZ, UK

Abstract. This paper presents an overview of a prototype hardware compiler which compiles a design expressed in the Ruby language into FPGAs. The features of two important modules, the refinement module and the floorplanning module, are discussed and illustrated. Target code can be produced in various formats, including device-specific formats such as XNF or CFG, and device-independent formats such as VHDL. The viability of our floorplanning scheme is demonstrated by a compiler backend for Algotronix's CAL1024 FPGAs. The implementation of a priority queue is used to illustrate our approach.

1 Introduction

Compiling selected parts of application programs into hardware, such as FPGAs, has recently attracted much interest. This method holds promise of producing better special-purpose systems more rapidly than existing techniques. A number of hardware compilers (see, for example, [8], [11]) have been developed for designs described in various languages into hardware netlists, which can then be mapped onto FPGAs by vendor software.

This paper presents an overview of two important modules, the refinement module and the floorplanning module, in a prototype compilation system. The system is based on Ruby [4], [9], a relational language for capturing block diagrams parametrically. There are mechanisms in Ruby for describing spatial and temporal iteration, allowing succinct and precise design specification. Moreover, the explicit representation of different forms of spatial iteration simplifies the production of layouts, and the declarative nature of the language allows designs to be refined by simple equational reasoning. Our aim is to exploit these features of Ruby to provide an efficient hardware compilation system.

The refinement module enables users to focus on the high-level structure of a design without being overwhelmed by details such as the size of individual datapaths. It is based on a constraint-propagation procedure. Given the size of inputs and a library of bit-level operators, it automatically constructs efficient low-level designs rapidly and in a provably-correct manner; this facilitates exploring architectures and evaluating the effects of different bit-level data representations.

Another important module, the floorplanning module, is devised to reduce the time to place and route a netlist produced by a hardware compiler. Since Ruby expressions carry information about the way a circuit can be assembled from primitive parts, our method is designed to exploit the structure of the

source program in generating a layout. It is also possible for the user to guide the placement of components and to import layouts that are developed manually or by other tools. Much of our floorplanning procedure is syntax-directed and is therefore very efficient.

While our floorplanning scheme is largely device-independent, to demonstrate its viability a compiler backend has been developed for Algotronix CAL1024 FPGAs. The implementation of a priority queue will be used to illustrate this approach.

2 Ruby

Ruby is a language of functions and relations. It has been used in developing a wide range of designs including signal processing architectures [2] and butterfly networks [4], and it has also been used in producing implementations partly in hardware and partly in software [7]. Detailed descriptions of Ruby can be found, for instance, in [4] and [9].

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

$$\langle x, y\rangle\ max\ (maximum(x,\ y)),$$

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. The identity relation *id* is given by $x\ id\ x$. To select or regroup components of composite data, there are wiring primitives such as *fork*, π_1 and *rsh*, given by $x\ fork\ \langle x, x\rangle$, $\langle x, y\rangle\ \pi_1\ x$ and $\langle x, \langle y, z\rangle\rangle\ rsh\ \langle\langle x, y\rangle, z\rangle$. To reflect a component along its trailing diagonal, we can use the converse operator, given by

$$x\ R^{-1}\ y\ \Leftrightarrow\ y\ R\ x.$$

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 1a):

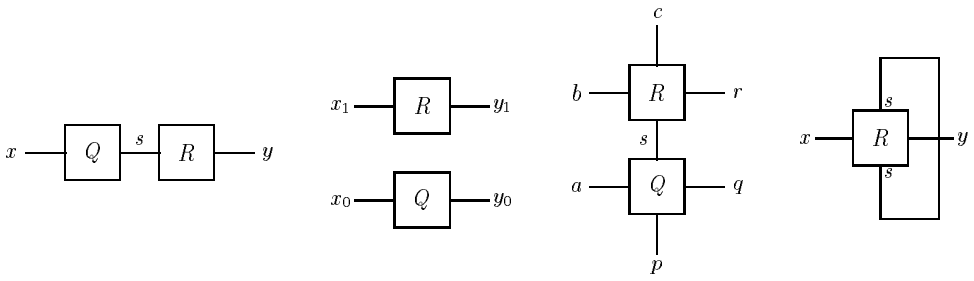
$$x\ (Q\ ;\ R)\ y\ \Leftrightarrow\ \exists s : (x\ Q\ s) \wedge (s\ R\ y).$$

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

If there are no connections between Q and R, the composite design is represented by parallel composition $[Q, R]$ (Figure 1b), where

$$\langle x_0,\ x_1\rangle\ [Q, R]\ \langle y_0, y_1\rangle\ \Leftrightarrow\ (x_0\ Q\ y_0)\ \wedge\ (x_1\ R\ y_1).$$

Repeated compositions of n copies of Q can be described by Q^n or $\mathsf{map}_n\ Q$, so for instance $fork^4 = fork\ ;fork\ ;fork\ ;fork$ and $\mathsf{map}_3\ rsh = [rsh, rsh, rsh]$.



a. $Q\ ;\ R$ b. $[Q, R]$ c. $Q \updownarrow R$ d. $loop\ R$

Fig. 1. Some Ruby operators.

Components with connections on four sides can be joined together by the *beside* and *below* operators; *below* (Figure 1c) is given by

$$\langle\langle a, b\rangle, c\rangle\ (Q \updownarrow R)\ \langle p, \langle q, r\rangle\rangle\ \Leftrightarrow\ \exists s : (\langle a, s\rangle\ Q\ \langle p, q\rangle) \wedge (\langle b, c\rangle\ R\ \langle s, r\rangle).$$

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 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", a squarer can be described by

$$x\ sq\ y\ \Leftrightarrow\ \forall t : x_t^2\ =\ y_t.$$

A latch can be modelled by a delay relation D, given by

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

A latch initialised to value i is denoted by $D\ i$.

Latches are used in designs with feedback to prevent unbuffered loops. A design Q containing an internal feedback path s can be modelled by the operator *loop* (Figure 1d):

$$x\ (loop\ R)\ y\ \Leftrightarrow\ \exists s : \langle x, s\rangle\ R\ \langle s, y\rangle.$$

3 Refinement

We can use Ruby to describe word-level designs, like the *max* or the *min* operator for integers. At bit-level, these operators can be built by logic gates which can also be captured in Ruby. The aim of our refinement system is to automatically produce the most efficient bit-level design from a high-level description.

Bit-level designs produced by the refinement system should satisfy constraints specified by the designer. Examples of constraints include the speed, size, latency and power consumption of a design, the maximum and minimum values of inputs and outputs, or a combination of the above. Of course, if the constraints are too strict, there may not be any bit-level design that satisfies them all. Our efforts so

far have been concentrated on constraints specifying the maximum and minimum values of inputs for a circuit.

There may be many possible bit-level designs which can implement a given word-level design. Also each data representation (such as two's complement representation) will result in a specific family of bit-level implementations. The refinement system can refine a word-level design into several bit-level implementations, depending on the bit-level data representation.

The refinement module is based on a constraint-propagation algorithm. The maximum and minimum values of inputs are propagated across the circuit. For a given component, once all constraints on its inputs are known, the constraints on 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 the data paths. A bit-level Ruby design can then be constructed. As an example, consider a priority queue which can be specified in Ruby as follows.

$$N = 4. \tag{1}$$

$$pq = pqcell^N. \tag{2}$$

$$pqcell = loop\ ((sort2 \updownarrow mux2) \updownarrow ([id,\ D\ 127]\ ;\ fork2)). \tag{3}$$

$$sort2 = fork\ ;\ [min,\ max]. \tag{4}$$

$$mux2 = fork\ ;\ [muxr\ 2,\ \pi_1]. \tag{5}$$

$$fork2 = {\pi_1}^{-1}\ ;\ [fork^{-1},\ fork]\ ;\ rsh. \tag{6}$$

Let us briefly introduce the correspondence between the Ruby program and the pictorial description of the priority queue; further details about possible designs and their development can be found in [9]. The Ruby descriptions for the word-level design (Figure 2) are shown above, which is implemented as a linear array of a repeating unit *pqcell* (expression 2), and the length of the array is 4 (expression 1). The repeating unit *pqcell* (expression 3) consists of three parts: an insertion sorter cell $sort2$ (expression 4), a selection unit $mux2$ (expression 5) and a data distribution unit $fork2$ (expression 6). There is an internal path in *pqcell* where the minimum output of the sorter is fed back while the maximum value is output to the next cell (expression 3). A latch (shown as a small triangle) is placed on the top of the feedback path, and it is initialised to the value 127.

Suppose the constraint specified by the designer is that the input data are natural numbers no larger than 127. Given that a bit is either T (True) or F (False), 127 is represented by $\langle T, T, T, T, T, T, T\rangle$. The refinement system produces a bit-level Ruby program with *min* replaced by $min\_un\_b\ 7\ 7$, where 7 represents the number of bits of the input. It also replaces *max* with $max\_un\_b\ 7\ 7$ in expression 3, $muxr\ 2$ with $muxr2\_bit\ 7$ in expression 4, and $D\ 127$ with $map\ 7\ (D\ T)$ in expression 5. The bit-level descriptions include instantiations from a library of parametrised bit-level components, which contains, for instance, max_un_b, min_un_b and $muxr2\_bit$, the bit-level implementations of

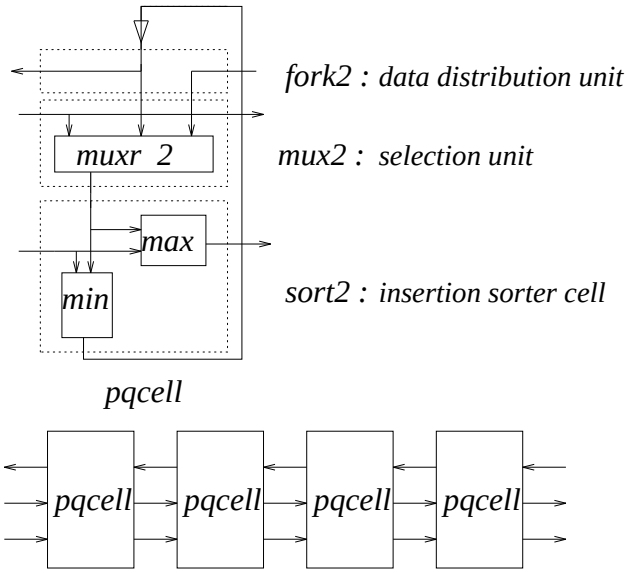


Fig. 2. A priority queue ($n = 4$).

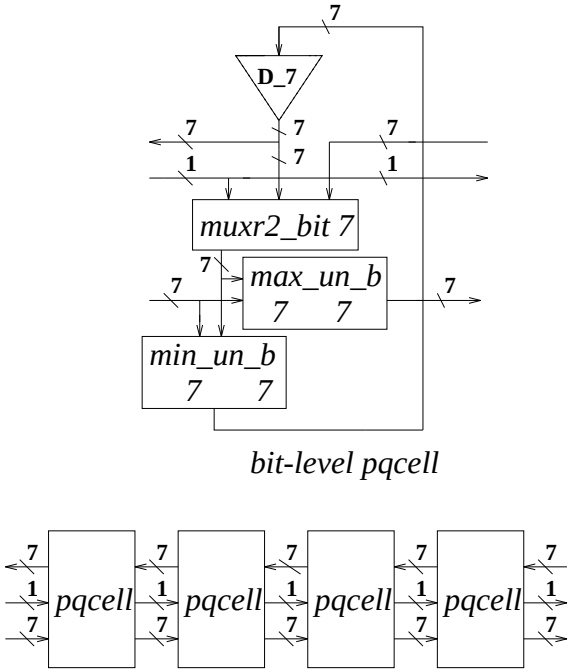


Fig. 3. Bit-level priority queue, with inputs not larger than 127 ($n = 4$).

max, *min* and the multiplexer *muxr* 2 operating on unsigned integers. The bit-level implementation of the priority queue is shown in Figure 3. Notice that the big triangle *D_7* represents seven *D* latches in parallel.

There are compiler backends for converting a bit-level description into various formats, such as XNF (Xilinx Netlist Format) or VHDL. The physical mapping onto FPGAs can then be carried out using commercial tools. An alternative implementation path will be sketched in the next section.

4 Floorplanning

A major bottleneck in automatic hardware synthesis is the time to place and route the netlist produced by a hardware compiler. The aim of our floorplanning module is to expedite the placement and routing procedure by exploring the structure of the source descriptions. To achieve high quality layouts, our floorplanning scheme includes facilities which allow combination of layouts produced both automatically and manually.

The floorplanning procedure consists of two phases. The first phase is the global placement and routing, which is mainly device-independent. In this phase a design is modelled as a rectangular block with connecting points on its four sides. Our floorplanning scheme allows the variation of block sizes, so that connecting positions between two adjacent blocks match each other to minimise the routing between them. In the second phase, the detailed routing within the blocks and their interface will be determined.

Consider first the global placement and routing phase. A design in Ruby is represented by a binary relation, while in pictorial form it is modelled as a rectangular block. A convention is required for assigning the domain and range variables of a relation to each side of the block – this step is known as direction assignment. The following convention is chosen: the domain data will be mapped onto the western or northern side, while the range data will be mapped onto the southern or eastern side [4].

Following this convention, the layout of a relation with its domain in the form of a two-tuple $\langle x, y \rangle$ can be a block with x on the western side and y on the northern side, or both x and y on either the western or the northern side. Similarly, the layout of a relation with its range in the form of a two-tuple can be a block with some of its connecting points on the southern side and some on the eastern side, or all of them on either the southern or the eastern side. One can show that, for a relation with both its domain and range in the form of a two-tuple, there are nine possible layouts [3]. The choice of which layout to adopt is determined by context or by a default convention. For instance some combinators in Ruby carry contextual information about possible direction assignment; the *below* combinator requires two of its domain and two of its range connections to be horizontal (Figure 1c).

After direction assignment, we check the compatibility of the interfaces between connected components. Since polymorphism is allowed in the domain and range of some Ruby primitives such as *fork*, a simple structure comparison is

insufficient. Instead a general unification algorithm was used to determine the most general substitution for the domain and range components, so that the interface constraints can be satisfied.

Sometimes information on direction of signal flow is necessary for certain devices, such as the cells used in Algotronix's CAL1024. In these cases we apply a constraint-propagation algorithm to determine the direction of signal flow for each Ruby wiring constructs.

The placement stages of our floorplanning system are not time-consuming because we exploit the structure of Ruby programs for placement. If we want to include a circuit which has been placed and routed manually or by other tools, we need to specify its size and the connection positions. Interface between the original and the imported layouts can then be produced by the compiler. A pair of curly braces are employed in the source Ruby program to indicate which part of the circuit should be laid out separately. The right curly brace is followed by a pair of parentheses which enclose the name of the manual layout file, so that the compiler can import this part of the layout and link it with others.

Further descriptions of our syntax-guided placement technique can be found in [3].

5 Device-Specific Mapping

While our approach to global placement and routing is largely device-independent, the detailed placement and routing flattens each block produced after global placement and routing, and it requires information specific to a particular device. To demonstrate the viability of our floorplanning scheme, a compiler backend has been customised for CAL1024 FPGAs developed by Algotronix (now Xilinx Development Corporation).

CAL1024 arrays are orthogonally connected structures obtained by replicating a basic cell which has one input port and one output on each of its four sides. An input port can be programmed to connect to one or more output ports, or to a function unit which can be programmed to behave as a two-input combinational logic gate or as a latch. The output of this function unit may also connect to one or more output ports. Hence a CAL cell may be used to perform processing and routing simultaneously. Figure 4 shows a CAL cell with its northerly output connected to its easterly input, and its easterly output is the Boolean conjunction of its westerly and northerly inputs.

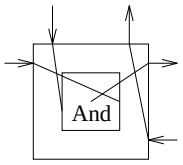


Fig. 4. CAL Cell.

During global placement and routing, two kinds of blocks are produced: blocks for combinational primitives such as *AND* and wiring primitives like *fork*. For combinational primitive blocks, we have developed a simple river routing algorithm to connect the connecting points on the four sides of the block to the cell performing the logic function of the primitive. A simple switch-box routing algorithm has also been devised to implement the detailed routing for the wiring blocks. The output of the floorplanner is a program in OAL [6], a variant of Ruby specialised for CAL devices. The OAL compiler can then be used to generate CFG files used for FPGA programming.

Although the floorplanner can perform the placement and routing fully automatically, the quality of the final implementation may be inferior to one produced by hand or by other tools. It is our intention to give the designer the flexibility to use our compiler for global placement and routing, while part of or all of the detailed placement and routing can be produced by other means. For instance, a designer may wish to develop by hand the repeating unit of an array-based circuit, since any inefficiency in the basic cell will be multiplied many times. The compiler can incorporate existing CAL designs into the implementation according to the annotations specified by the designer in the source program, as described in section 4.

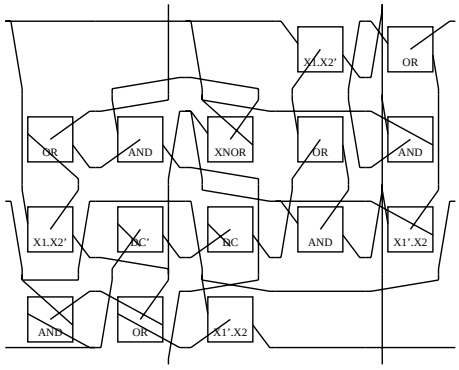


Fig. 5. CAL implementation of a bit-level priority queue cell.

Consider a priority queue implementation obtained by optimising the bit-level design in section 3 (see [9]). The bit-level repeating unit (Figure 5) was developed by hand and is highly optimised; this unit is then replicated vertically to form a column which corresponds to the core of a *pqcell* in Figure 3. The CAL implementation of the priority queue is shown in Figure 6. Note that the number and order of the interface connections correspond to those in Figure 3, except that the two bottom outputs of the rightmost *pqcell* are discarded.

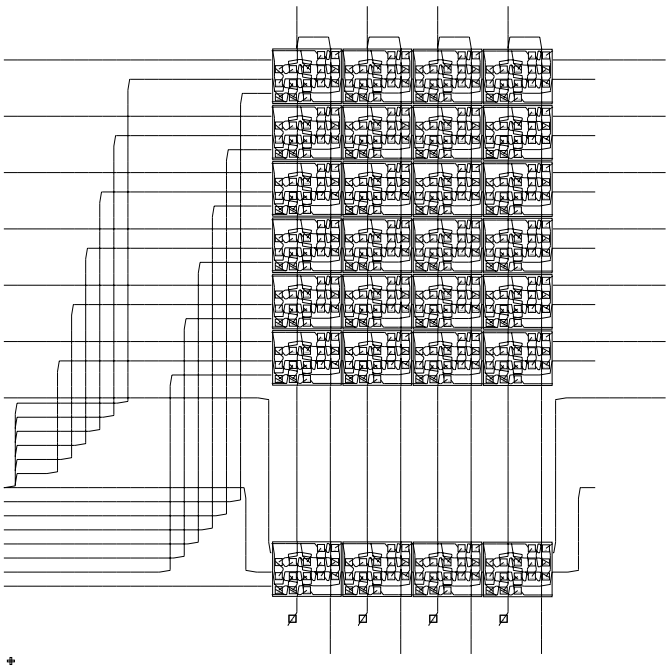


Fig. 6. CAL implementation of a bit-level priority queue (n = 4, m = 7).

6 Future Work

In the refinement module of our compilation system, we have focused on constraints specifying the maximum and minimum values of inputs for a word-level circuit. Our method can be extended to take into consideration other kinds of constraints: examples include critical path, latency or the number of a particular component. If no solutions exist that satisfy all user-specified constraints, we can choose the solution that satisfies most of the high-priority constraints.

The CAL backend of our compiler demonstrates the viability of our floorplanning module. We have not, however, optimised the switch-box routing or the river-routing algorithms, and the layouts produced automatically can become rather large. For better results, we can use methods like min-cut or simulated annealing hierarchically in placement and routing [10]. Device-specific compaction techniques should also be studied.

Much of our method for generating layouts is syntax-directed. The quality of the compiled implementation depends largely on the Ruby source program which describes the design; therefore source transformation can be adopted for optimisation. One way to automate this step is to have an accurate performance estimation procedure to drive the transformation engine.

It will also be interesting to extend our work to support partial and run-time reconfiguration of FPGAs, to support developing multi-chip systems, and to support implementing asynchronous and self-timed designs [1].

Acknowledgements

The support of Xilinx Development Corporation, Scottish Enterprise, Department of Computing, Imperial College and Oxford University Hardware Compilation Research Group is gratefully acknowledged. S. Guo thanks the Sino-British Friendship Scholarships Foundation for their support.

References

1. E. Brunvand, "Using FPGAs to implement self-timed systems", *Journal of VLSI Signal Processing*, vol. 6, 1990, pp. 173-190.
2. S. Guo, W. Luk and P. Probert, "Developing parallel architectures for range and image sensors", in *Proc. IEEE Int. Conf. on Robotics and Automation*, IEEE Computer Society Press, 1994, pp. 2205–2210.
3. S. Guo and W. Luk, "Producing design diagrams from declarative descriptions", to appear in *Proc. Fourth Int. Conf. on CAD and CG*, SPIE, 1995.
4. G. Jones and M. Sheeran, "Circuit design in Ruby", in *Formal Methods for VLSI Design*, J. Staunstrup (ed.), North-Holland, 1990, pp. 13–70.
5. W. Luk, "Analysing parametrised designs by non-standard interpretation", in *Proc. Int. Conf. on Application-Specific Array Processors*, S.Y. Kung, E. Swartzlander, J.A.B. Fortes and K.W. Przytula (eds.), IEEE Computer Society Press, 1990, pp. 133–144.
6. W. Luk and I. Page, "Parameterising designs for FPGAs", in *FPGAs*, W. Moore and W. Luk (eds.), Abingdon EE&CS Books, 1991, pp. 284–295.
7. 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.
8. W. Luk, D. Ferguson and I. Page, "Structured hardware compilation of parallel programs", in *More FPGAs,* W. Moore and W. Luk (eds.), Abingdon EE&CS Books, 1994, pp. 213–224.
9. W. Luk, "A declarative approach to incremental custom computing", in *Proc. IEEE Workshop on FPGAs for Custom Computing Machines*, D.A. Buell and K.L. Pocek (eds.), IEEE Computer Society Press, 1995.
10. M. Newman, W. Luk and I. Page, "Constraint-based hierarchical hardware compilation of parallel programs", in *Field-Programmable Logic: Architecture Synthesis and Applications*, LNCS 849, Springer-Verlag, 1994, pp. 220–229.
11. M. Wazlowski et. al., "PRISM II: compiler and architecture", in *Proc. IEEE Workshop on FPGAs for Custom Computing Machines*, D.A. Buell and K.L. Pocek (eds.), IEEE Computer Society Press, 1993, pp. 9–16.

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