A Declarative Approach to Incremental Custom Computing

Wayne Luk
Department of Computing,
Imperial College of Science, Technology and Medicine
180 Queen's Gate
London
England SW7 2BZ

Abstract

Incremental methods can be used to produce implementations rapidly and to facilitate multi-level design optimisation. This paper describes a declarative framework, based on the language Ruby, that supports incremental design and validation of custom computers. The key elements of the approach include parameterised descriptions, design transformation and data refinement. Several priority queue designs are employed to illustrate our techniques and the computer-based tools; we also present the use of our framework in producing a priority queue implementation using Algotronix CAL devices.

1 Introduction

Recent advance in reconfigurable logic technology, particularly field-programmable gate arrays (FPGAs), has led to a proliferation of commercial and experimental computing platforms based on these devices [3]. FPGA-based designs are clearly becoming increasingly popular, because of advantages such as short turnaround time, user reconfigurability and low development costs. Further exploitation of FPGA-based systems, however, is hampered by the lack of effective methods and tools that allow quick and risk-free design production and modification, while giving designers sufficient control when this is desirable [14].

The purpose of the research reported here is to establish a design framework that overcomes these drawbacks, and to explore its use in various applications. The framework should be incremental – it will allow rapid production and analysis of prototypes, as well as their further systematic refinement and adaptation when required; it should also be easy to learn and to use. Our approach should be useful to both novice and experienced designers. Application programmers and those without detailed knowledge of electronics will mainly use the available primitives, design templates and optimisation procedures to develop their designs, guided by a parameterised performance model. Hardware specialists will have opportunities for creating new primitives, design templates, performance models and so on for their designs; indeed their creations may provide a basis for the less experienced users.

The above considerations lead to several desirable properties for an incremental design framework. First, it should facilitate step-by-step development as well as re-using, adapting and documenting design experience [4]. Second, to allow designer control and to encourage design re-use at various levels of abstraction, the framework should be able to span the entire design hierarchy from architectural to gate level, and possibly includes device-dependent descriptions. Third, the description of designs and the user interface of tools should be simple and flexible. Next, design validation should also be incremental: it is beneficial to support hierarchical and mixed numerical, symbolic and bit-level simulation as well as algebraic transformation, depending on the required level of detail, generality and confidence in design correctness. Finally, the framework should be based on a sound formalism; experience suggests that much delay can be caused by design errors, and many systems of interest, like robotic or bio-medical systems, are safety-critical.

We are exploring various languages that take these properties into account. The rest of this paper describes a particular approach based on a declarative language; its advantages include having a concise notation and a simple reasoning framework and tools. Our presentation offers a pragmatic way of using declarative techniques in system design, which we hope will complement the theoretical expositions [6], [11].

A number of priority queue designs will be used to illustrate our approach, because (a) they are relatively simple and comprehensible, (b) some of the designs included here appear to be novel, (c) their hardware realisations should accelerate applications such as event-driven simulators [5], (d) their development is typical of many high-performance architectures, and (e) they can be implemented very efficiently in FPGAs, as shown in a later section.

2 Architectural description

Three ways of providing flexible and reusable structures are parameterised descriptions, design transformation and data refinement. A parameterised description produces specific designs by the instantiation of parameters; design transformations can be used to generate one parameterised description from another – usually the two will behave the same functionally but with different performance; data refinement relates operations on high-level data (such as integers) and operations on low-level data (such as bits). In this section we outline our framework, based on the declarative language Ruby, for parameterising and composing block diagrams and circuits. We shall focus mainly on illustrating the use of Ruby for incremental design; further details about the theoretical aspects of Ruby can be found elsewhere (such as [6] and [11]).

In Ruby the behaviour of a component is described by a binary relation, so that a squaring operation can be described by $x \ sq \ x^2$, where x is the domain and x^2 is the range of the relation sq. Since frequently we need to replicate or to rearrange the elements in a sequence, there are relations such as $fork$ and $swap$, given by $x \ fork \ \langle x, x\rangle$ and $\langle x, y\rangle \ swap \ \langle y, x\rangle$. Another example is zip_n, which relates a pair of n-sequences to a sequence of n pairs:

$$\langle\langle x_0, x_1, x_2\rangle, \langle y_0, y_1, y_2\rangle\rangle \ zip_3 \ \langle\langle x_0, y_0\rangle, \langle x_1, y_1\rangle, \langle x_2, y_2\rangle\rangle.$$

The simplest relation is probably the identity id, such that $x \ id \ x$; notice that the variable x in the description of id can itself be a sequence or any other data structure. This kind of parameterisation is common in declarative languages [2], but it is less common in imperative languages such as VHDL [1].

Components with connections on all four sides are modelled by relations that relate pairs to pairs, such that the domain corresponds to the connections on the west and the north side, and the range corresponds to those on the south and the east. A halfadder, for instance, can be specified by $\langle x, y\rangle \ hadd \ \langle c, s\rangle$, where c and s are respectively the carry and the sum outputs (Figure 1).

Composition operators can be used to assemble components to form composite designs. First, let us consider how two adjacent circuits can be put together in series and in parallel. Two circuits Q and R in series is denoted by $Q\,;R$, a composite circuit with Q and R sharing a hidden compatible interface s (Figure 2a):

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

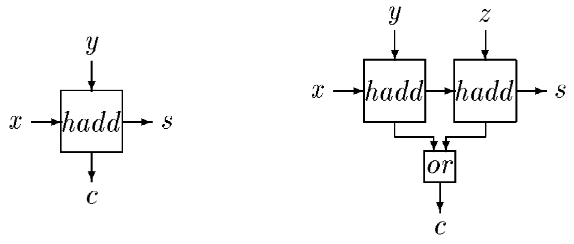

Figure 1 Halfadder $hadd$, and fulladder $fadd$ made from halfhadders and an or-gate.

so $x\,(sq\,;sq)\,x^4$. The ";" operator is known as relational composition, and can easily be shown to be associative. Parallel composition of two components Q and R, denoted by $[Q, R]$, represents the combination with no connection between Q and R (Figure 2b),

$$\langle x, y\rangle\,[Q, R]\,\langle u, v\rangle \quad \Leftrightarrow \quad (x\ Q\ u) \ \wedge \ (y\ R\ v),$$

hence $\langle x, y\rangle\,[sq, (sq\,;sq)]\,\langle x^2, y^4\rangle$.

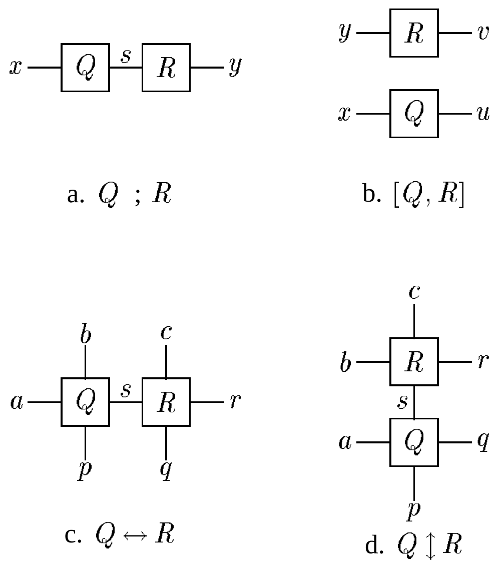

Figure 2 Some binary operators in Ruby.

For convenience, we also have the abbreviations

$$\begin{aligned}\mathsf{fst}\ R &= [R, id],\\ \mathsf{snd}\ R &= [id, R].\end{aligned}$$

Notice that Ruby expressions can be used to indicate relative placement of components, possibly for the benefit of an automatic layout system. Moreover many Ruby operators have useful algebraic properties for optimising designs, as we shall see later.

Components with connections on four sides can be joined together by the $beside$ (Figure 2c) and $below$ (Figure 2d) operators; $beside$ is given by

$$\begin{aligned}&\langle a, \langle b, c\rangle\rangle\ (Q \leftrightarrow R)\ \langle\langle p, q\rangle, r\rangle \\ &\quad \Leftrightarrow\ \exists s.\ (\langle a, b\rangle\ Q\ \langle p, s\rangle)\ \wedge\ (\langle s, c\rangle\ R\ \langle q, r\rangle).\end{aligned}$$

The definition of $below$ is similar. A fulladder, given by $\langle x, \langle y, z\rangle\rangle\ fadd\ \langle c, s\rangle$, can be obtained by connecting together two halfadders and an or-gate (Figure 1): $fadd\ =\ hadd \leftrightarrow hadd\ ;\ \mathsf{fst}\ or.$

Repeated relational and parallel composition n-times are given by R^n (Figure 3a) and $\mathsf{map}_n\ R$ (Figure 3b), while repeated beside and repeated below are $\mathsf{row}_n\ R$ (Figure 3c) and $\mathsf{col}_n\ R$ (Figure 3d). These operators can be defined by recursion [6]. To describe a design with feedback, we use the operator loop, given by

$$x\ (\mathsf{loop}\ R)\ y \quad \Leftrightarrow \quad \exists s.\ \langle x, s\rangle\ R\ \langle s, y\rangle$$

(Figure 3e).

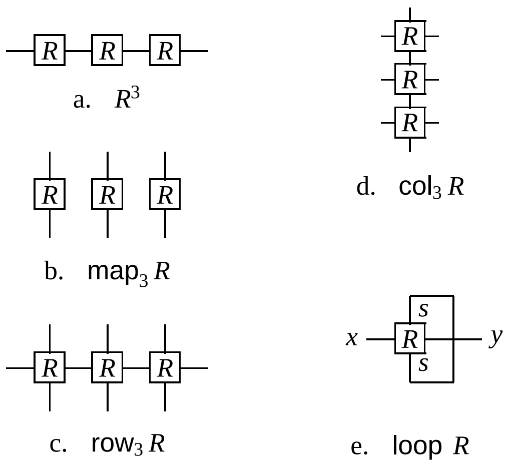

Figure 3 Some Ruby operators that capture common computation patterns.

Let us now explain how priority queues can be captured in Ruby. Recall that two operations can be performed on a priority queue: inserting a data record into a set, and extracting from a set the record with the smallest key according to some linear ordering. One way of implementing a priority queue involves an ordered state $\langle s_0, s_1, s_2, s_3\rangle$, such that an insertion with an input a where $s_0 \leq a < s_1$ produces the next state $\langle s_0, a, s_1, s_2\rangle$. An extraction operation on the state $\langle s_0, s_1, s_2, s_3\rangle$, on the other hand, will result in the next state $\langle s_1, s_2, s_3, \infty\rangle$ and output s_0, where ∞ denotes the greatest element of the linear ordering.

The architecture $Qstl$ that we adopt for the state transition logic described above consists of three blocks $IdShl$, $Select$ and $Insert$ stacking on top of one another (Figure 4),

$$Qstl \quad = \quad (Insert \updownarrow Select) \updownarrow IdShl.$$

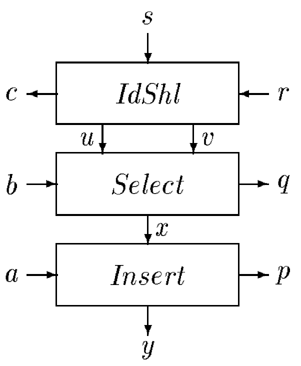

Figure 4 The block structure of the state transition logic $Qstl$.

$IdShl$ provides two outputs u and v. u is the same as the current state s, and v is a left-shifted version of s. In other words, given that $s = \langle s_0, s_1, s_2, s_3\rangle$, then:

$$\begin{aligned}&\langle c, s\rangle\ IdShl\ \langle\langle u, v\rangle, r\rangle \\ &\quad \Leftrightarrow\ (c = s_0)\ \wedge\ (u = s) \wedge\ (v = \langle s_1, s_2, s_3, r\rangle)\end{aligned} \tag{1}$$

Note that in our priority queue implementation, r is connected to a constant supply of ∞s to replace the extracted elements. A parameterised version of $IdShl$ is

$$IdShl \quad = \quad \mathsf{snd}\ fork\ \ ;\ swap \leftrightarrow (\mathsf{row}_n\ shlcell) \tag{2}$$

where $\langle d, d\rangle\ shlcell\ \langle e, e\rangle$.

Next, depending on the selection signal b, either u or v will be selected by $Select$ to form the input x for the $Insert$ block, which inserts the input a into x such that the output y (the next state) is ordered. $Select$ can be realised as a row of multiplexors mux operating on an interleaved version of u and v:

$$Select \quad = \quad \mathsf{snd}\ zip_n\ \ ; \mathsf{row}_n\ mux$$

while $Insert$ can be implemented as a row of $scell$s which sorts two elements:

$$Insert \quad = \quad \mathsf{row}_n\ scell$$

```
          a   b  c     s0  s1  s2  s3        y0  y1  y2  y3
   0 - <<<8,  0>,100>,<100,100,100,100>> ˜ <8,  100,100,100>  insert : b=0, a=8
   1 - <<<5,  0>,8  >,<8,  100,100,100>> ˜ <5,  8,  100,100>  insert : b=0, a=7
   2 - <<<7,  0>,5  >,<5,  8,  100,100>> ˜ <5,  7,  8,  100>  insert : b=0, a=6
   3 - <<<6,  0>,5  >,<5,  7,  8,  100>> ˜ <5,  6,  7,  8  >  insert : b=0, a=5
   4 - <<<100,1>,5  >,<5,  6,  7,  8  >> ˜ <6,  7,  8,  100>  extract: b=1, c=5
   5 - <<<100,1>,6  >,<6,  7,  8,  100>> ˜ <7,  8,  100,100>  extract: b=1, c=6
   6 - <<<2,  0>,7  >,<7,  8,  100,100>> ˜ <2,  7,  8,  100>  insert : b=0, a=2
   7 - <<<3,  0>,2  >,<2,  7,  8,  100>> ˜ <2,  3,  7,  8  >  insert : b=0, a=3
   8 - <<<100,1>,2  >,<2,  3,  7,  8  >> ˜ <3,  7,  8,  100>  extract: b=1, c=2
   9 - <<<100,1>,3  >,<3,  7,  8,  100>> ˜ <7,  8,  100,100>  extract: b=1, c=3
  10 - <<<100,1>,7  >,<7,  8,  100,100>> ˜ <8,  100,100,100>  extract: b=1, c=7
  11 - <<<100,1>,8  >,<8,  100,100,100>> ˜ <100,100,100,100>  extract: b=1, c=8
```

Figure 5 Simulating the state transition logic $Qstl$.

where $\langle x, y\rangle\ scell\ \langle min(x, y), max(x, y)\rangle$.

At this stage it is desirable to check that the state transition logic $Qstl$ behaves as desired. This can be achieved in two ways: either by formal development, or by simulation using the Rebecca design system which is under development at Imperial College and Oxford University. An example of formal development can be found in [8], and the use of simulation will be described next.

3 Support tools

Rebecca is a system for developing digital designs. It offers various facilities including numerical and symbolic simulation, layout sketching, design analysis, optimising transformations and hardware compilation for designs captured in a variant of the Ruby language. Here we shall illustrate how the Rebecca simulator can be used to explore the behaviour of designs.

A useful feature of our simulator is the capability of carrying out numerical, gate-level and symbolic simulation using the same design description. For instance, if we supply to the simulator the boolean signals FFF, FFT, FTF, FTT in successive cycles for the fulladder $fadd$, we get

```
0 - <F,<F,F>>   ˜   <F,F>
1 - <F,<F,T>>   ˜   <F,T>
2 - <F,<T,F>>   ˜   <F,T>
3 - <F,<T,T>>   ˜   <T,F>
```

The first column displays the cycle number, and the "˜" symbol separates the domain data from the range data. It happens in this case that all the inputs are in the domain. If we now supply the symbolic inputs a,b and c in a single cycle, we get

```
0 - <a,<b,c>> ˜
    <(a and b) or ((a xor b) and c),
     (a xor b) xor c>
```

showing that the outputs are indeed what we expect.

To study the behaviour of the state transition logic $Qstl$, one can simulate the components $IdShl$, $Select$ and $Insert$ separately and then their composition. The simulator can deal with either the non-parameterised or the parameterised version of $IdShl$, given by (1) and (2) respectively; the non-parameterised description is easier to write and understand, but it needs to be altered when the queue size is changed.

To simulate $Qstl$, we supply the inputs a and b and the current state s (Figure 4), and the simulator will return the output c and the next state y; the next state of one cycle is then used as the current state of the next cycle. In the annotated simulation output in Figure 5, ∞ is approximated by 100, and for clarity the input r and the outputs p and q are not included.

To describe the complete priority queue design as a state machine, we need a way of representing latches and sequential circuits in general. This is achieved in Ruby by relations that handle streams, or time sequences, such that the stream version of sq becomes $x\ sq\ y \Leftrightarrow (\forall t.\,(x_t)^2 = y_t)$. It can be shown that the algebraic properties of Ruby are preserved by lifting relations to work on streams. A latch is modelled by a relation $\mathcal{D}$ whose range stream is one time unit behind the domain stream, $x\,\mathcal{D}\,y \Leftrightarrow (\forall t.\,x_{t-1} = y_t)$.

An n-element priority queue can now be expressed as a state machine,

$$Q_0 \quad = \quad \mathsf{loop}\ (Qstl\ ;\ \mathsf{fst}\ (\mathsf{map}_n\ \mathcal{D})).$$

Note that the correct operation of Q_0 requires the feedback latches to be initialised with ∞. Q_0 can also be simulated,

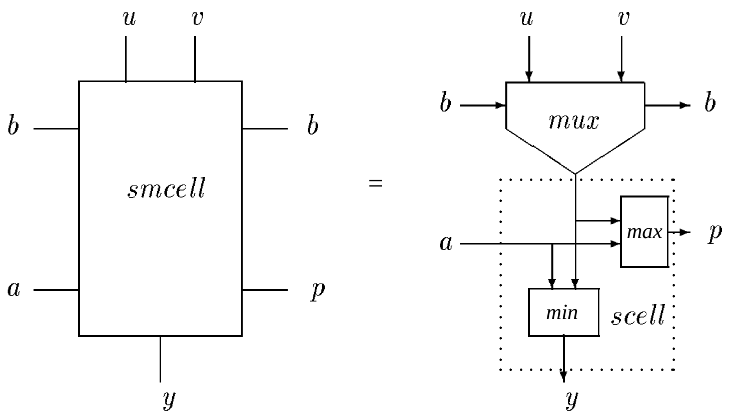

Figure 6 Structure of $smcell$.

although this time only the inputs a and b are required, as the states s and y are hidden.

The Rebecca system includes an automatic data refinement facility for producing gate-level descriptions. These descriptions can then be used to generate netlists in a variety of formats, including Xilinx XNF and Algotronix CFG formats, for automatic place and route tools. A route to VHDL is also available. We can use these tools to produce an FPGA implementation for Q_0 directly from its word-level description.

However, some designers may choose to produce bit-level circuits by hand, especially when they are dealing with repeating units in structured designs such as systolic arrays. The Rebecca simulator can also be used for checking user-defined bit-level circuits: there are functions that transform word-level data (such as integers) to various bit-level data (such as two's complement representation) and vice versa, so it is straightforward to compare the hand-developed bit-level design and the word-level specification. These data transformations are also useful for revealing high-level behaviour when an optimised bit-level circuit is being adapted for another purpose.

4 Word-level optimisations

In this section we illustrate the systematic transformation of Q_0 to increase regularity and to produce systolic implementations. At present Q_0 is expressed as a single state machine consisting of three blocks and a single bank of feedback latches. It is desirable to decompose this state machine into a cascade of simple state machines, for two reasons: first, if the blocks can be merged to form a single repeating structure, then one only needs to optimise a relatively small repeating unit by hand or using automatic tools; second, the collection of state machines may be pipelined in various ways so that the clock speed is largely independent of the number of processors.

While an experienced designer can probably obtain reasonably optimised designs without spending too much time, we include the three steps – block fusion, state distribution and pipelining – here as an example of transforming designs and a proof of correctness. The automation of these steps is being investigated. Readers aiming at an overview of our method may skip the algebraic details below; the result of these transformations will be shown in diagrams.

The first step, block fusion, involves recasting the state transition logic $Qstl$ in the form of a row of elements. This can be achieved by expressing $IdShl$ as

$$\mathsf{row}_n\, fcell \;;\; \mathsf{fst}\, zip_n^{-1}$$

where $\langle a, a\rangle\ fcell\ \langle\langle a, b\rangle, b\rangle$ and $zip_n^{-1}\ ;\ zip_n = id$. Algebraic laws in Ruby such as

$$\begin{aligned}(\mathsf{snd}\, R \;;\; A) \updownarrow (B \;;\; \mathsf{fst}\, Q) &= A \updownarrow (B \;;\; \mathsf{fst}\,(Q \;;\; R))\\ (\mathsf{row}_n\; Q) \updownarrow (\mathsf{row}_n\; R) &= \mathsf{row}_n\,(Q \updownarrow R)\end{aligned}$$

can be used to transform $Qstl$:

$$\begin{aligned}Qstl &= ((\mathsf{row}_n\, scell) \updownarrow (\mathsf{snd}\, zip_n \;;\; \mathsf{row}_n\, mux))\\ &\qquad \updownarrow (\mathsf{row}_n\, fcell \;;\; \mathsf{fst}\, zip_n^{-1})\\ &= ((\mathsf{row}_n\, scell) \updownarrow (\mathsf{row}_n\, mux))\\ &\qquad \updownarrow (\mathsf{row}_n\, fcell \;;\; \mathsf{fst}\; (zip_n^{-1} \;;\; zip_n))\\ &= ((\mathsf{row}_n\, scell) \updownarrow (\mathsf{row}_n\, mux))\\ &\qquad \updownarrow (\mathsf{row}_n\, fcell)\\ &= \mathsf{row}_n\; Qcell\end{aligned}$$

where (Figure 6)

$$\begin{aligned}Qcell &= smcell \updownarrow fcell,\\ smcell &= scell \updownarrow mux.\end{aligned}$$

An optimised $Qcell$, the repeating unit, can now be produced by hand or by automatic tools. Note that since the expression $\mathsf{row}_n\ R$ can be considered as a loop in conventional languages, this process is analogous to loop fusion carried out by optimising compilers.

The second optimisation step is to decompose the state machine Q_0. We need the identity

$$\mathsf{row}_n\ Q\ ;\ \mathsf{fst}\,(\mathsf{map}_n\ R) \quad = \quad \mathsf{row}_n\,(Q\ ;\ \mathsf{fst}\ R),$$

so that

$$\begin{aligned} Q_0 &= \mathsf{loop}\ (\mathsf{row}_n\ Qcell\ ;\ \mathsf{fst}\,(\mathsf{map}_n\,\mathcal{D})) \\ &= \mathsf{loop}\ (\mathsf{row}_n\,(Qcell\ ;\ \mathsf{fst}\,\mathcal{D})) \end{aligned}$$

The theorem for state machine decomposition,

$$\mathsf{loop}\ (\mathsf{row}_n\ R) \quad = \quad (\mathsf{loop}\ R)^n,$$

can now be applied to give $Q_0 = Qcell0^n$, where $Qcell0 = \mathsf{loop}\ (Qcell\ ;\ \mathsf{fst}\,\mathcal{D})$ (Figure 7). The correctness proof of the state machine decomposition theorem is given in [8].

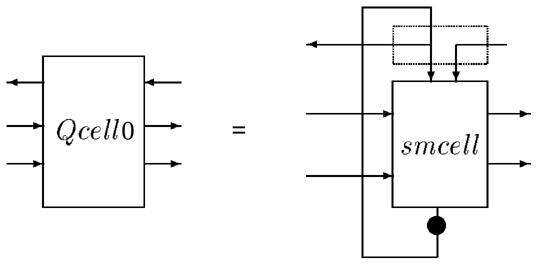

Figure 7 Design $Qcell0$. The dotted box identifies $fcell$, and the black disk corresponds to a latch.

The final step in optimising our word-level priority queue design is to pipeline Q_0 by including registers between adjacent $Qcell0$s. This can be done in several ways, and two of them will be shown here. Given that $n = km$, the first way is to pipeline Q_0 by theorems similar to

$$\mathcal{D}^m\ ;\ R^{km} \quad = \quad (\mathcal{D}\ ;\ R^k)^m$$

(provided that $\mathcal{D}\,;R = R\,;\mathcal{D}$) to give

$$\begin{aligned} Q_1 &= [[\mathcal{D},\mathcal{D}],\mathcal{D}]^m\ ;\ Q_0 \\ &= [[\mathcal{D},\mathcal{D}],\mathcal{D}]^m\ ;\ Qcell0^{km} \\ &= ([[\mathcal{D},\mathcal{D}],\mathcal{D}]\ ;\ Qcell0^k)^m \\ &= (Qcell1\ ;\ Qcell0^{k-1})^m \end{aligned}$$

where

$$\begin{aligned} &Qcell1 \\ &= [[\mathcal{D},\mathcal{D}],\mathcal{D}]\ ;\ Qcell0 \\ &\ \ \mathsf{loop} \\ &\ \ \ ((\mathsf{fst}\,[\mathcal{D},\mathcal{D}]\ ;\ smcell) \updownarrow (fcell\ ;\ \mathsf{fst}\ (\mathsf{fst}\,\mathcal{D}))) \end{aligned}$$

(Figure 8). In other words, Q_1 is made up of m repeating units, where $mk = n$ and $k \geq 2$. Each repeating unit consists of a linear array of a $Qcell1$ and $(k-1)$ $Qcell0$s. An example of Q_1 is shown in Figure 9.

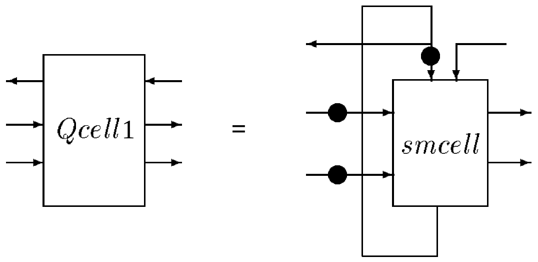

Figure 8 Design $Qcell1$.

Notice that the parameter k controls the degree of pipelining: the most pipelined design is obtained when $k = 2$ and $m = n/2$, while the least pipelined one corresponds to the case when $k = n$ and $m = 1$. Table 1 summarises how different values of k affects the resource/performance trade-off – the estimation of clock periods assumes that wire delays are included in cell delays. If m is not a factor of n, then Q_1 can be implemented by having another unit made up of a $Qcell1$ and $k' - 1$ $Qcell0$s, where $0 < k' < k$ and $n = mk + k'$.

The second way of pipelining Q_0 involves a method known as *slowdown* (see [6],[7],[11]). A 2-slow version of Q_0, Q_0', can be obtained by replacing every latch in Q_0 by two latches in series; as a result Q_0' handles two independent computations in successive cycles. Q_0' can be fully pipelined to give Q_2:

$$\begin{aligned} Q_2 &= [[\mathcal{D},\mathcal{D}],\mathcal{D}]^n\ ;\ Q_0' \\ &= [[\mathcal{D},\mathcal{D}],\mathcal{D}]^n\ ;\ (\mathsf{loop}\ (fcell\ ;\ \mathsf{fst}\ \mathcal{D}^2))^n \\ &= Qcell2^n \end{aligned}$$

where (Figure 10)

$$\begin{aligned} &Qcell2 \\ &= [[\mathcal{D},\mathcal{D}],\mathcal{D}]\ ;\ \mathsf{loop}\ (fcell\ ;\ \mathsf{fst}\,\mathcal{D}^2) \\ &= \mathsf{loop}\ ((\mathsf{fst}\ [\mathcal{D},\mathcal{D}]\ ;\ smcell\ ;\ \mathsf{fst}\,\mathcal{D}) \\ &\qquad \updownarrow (fcell\ ;\ \mathsf{fst}\,(\mathsf{fst}\,\mathcal{D}))). \end{aligned}$$

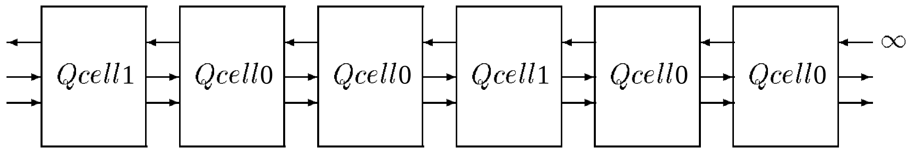

Figure 9 An instance of Q_1 ($n = 6$, $k = 3$, $m = 2$). The left-hand output provides the extracted result. The upper left-hand input controls insertion and extraction, and records for insertion should be placed at the lower left-hand input.

Table 1 Comparison of n-element priority queue designs for records of ℓ bits wide. T_s is the delay of an $scell$ and T_m is the delay of a mux (Figure 6).

	Minimum clock period	Number of latches in array	Slowdown factor
Design Q_1	$max(kT_s + T_m, 2T_s + 2T_m)$ $(n \geq k \geq 2)$	$\frac{n(k\ell + \ell + 1)}{k}$	1
Design Q_2	$T_s + T_m$	$n(3\ell + 1)$	2

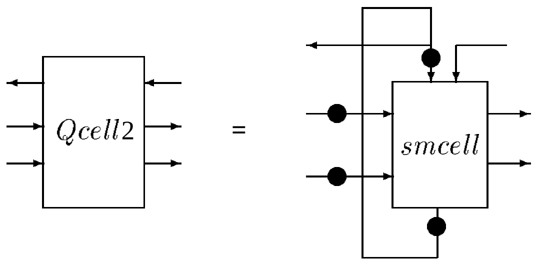

Figure 10 Design $Qcell2$.

The features of Q_2 are summarised in Table 1. While Q_2 has a higher clock speed than the fastest version of Q_1 (with k=2), it requires twice as many latches as Q_1. Moreover, since Q_2 is 2-slow, each operation takes two cycles to complete – twice as many cycles as Q_1 would take. The behaviour and performance of Q_1 and Q_2 can also be validated using the Rebecca system. A design similar to Q_2 has been outlined in [7], but that paper does not include the details of cell structure or performance comparison.

The designs presented above are just two examples that can be obtained from the initial Q_0 description; other possibilities include various forms of transposed [9] and serialised designs [11].

5 Bit-level optimisations

An incremental method should support optimisation down to the lowest level. There are many situations in which a little extra effort on the part of the designer may yield an implementation better than any automatic tools can produce. It is important, however, to ensure that the optimised version preserves the intended functional behaviour.

One way of optimising the performance of a circuit is to use hard macros, which are highly-optimised technology-specific library cells. For instance hard macros, if available, can be used for the mux, max and min blocks in $smcell$ (Figure 6). However, this method often leads to wiring congestion between the connected blocks, which is especially undesirable for FPGAs. A better solution is to develop a bit-level cell which can be replicated in a column to form a $Qcell0$. Such a cell is shown in Figure 11, and can be described in around 10 lines of Ruby code; another 5 lines complete the description of the entire priority queue design.

With the help of the Rebecca system, it only took a short time to compile this bit-level description into hardware and to validate it against the word-level design – this is partly due to the concise notation that we adopt, and partly due to the powerful simulation facilities, such as functions converting between high-level and low-level data. One can also use algebraic transformations to verify that the bit-

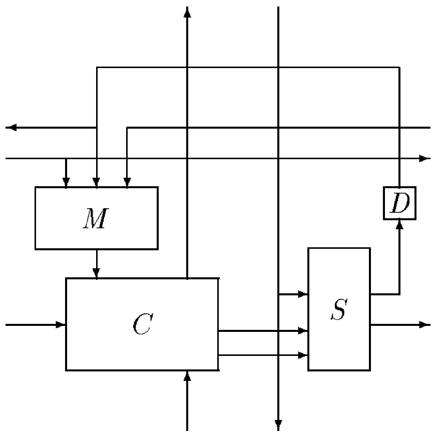

Figure 11 Bit-level priority queue cell. M is a mulitplexer, C is a comparator, S is a conditional swapper and D is a latch.

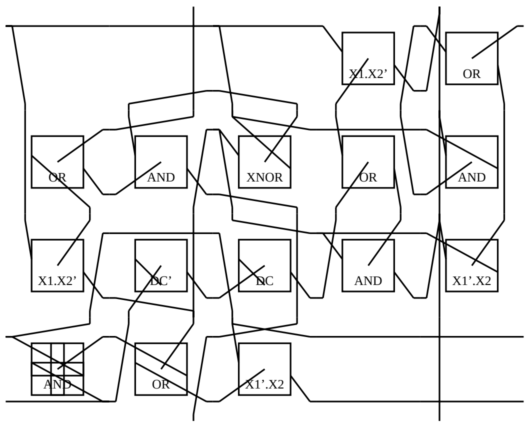

Figure 12 CAL implementation of a bit-level priority queue cell. The inputs to a gate are connected to its two sides, while its output emerges from its centre.

level design is a faithful implementation of the word-level description.

The highest performance of a design can be obtained by exploiting device-specific features. Although this is probably not profitable to do for random logic implementations, it is often desirable for structured designs in which inefficiency in a repeating unit will be amplified many times.

Bit-level designs are relatively straightforward to implement using fine-grained FPGAs with a symmetric architecture, such as Algotronix CAL devices. Figure 12 shows a compact CAL-based implementation of the bit-level circuit given in Figure 11. A parameterised description of this design has been captured in a device-specific version of Ruby known as OAL [10]. The CAL implementation took less than a day to develop, given the bit-level circuit diagram (Figure 11). Although our CAL devices were acquired 5 years ago, we found that they should still be capable of implementing a priority queue design with a speed of over 10MHz when fully pipelined.

6 Concluding remarks

The benefits of parameterised descriptions, hierarchical design and systematic development are well-known (see [4], [14]). A declarative approach shares these benefits, while providing an incremental route for design validation based on numerical and symbolic simulation as well as algebraic transformations.

Our aim is to exploit the concise notation and the simple reasoning framework of declarative languages for fast, modular and flexible design, possibly in concert with other languages such as VHDL, C and occam [13]. We have shown that much of the design hierarchy from architectural to gate-level descriptions can be expressed in a declarative manner, facilitating multi-level design optimisation and re-use of design expertise. It is not even necessary to understand our language in order to use the result of our derivation: given the performance characteristics of the available computational elements such as multiplexers and comparators, a designer or a tool can simply substitute the appropriate data in formulae like the ones in Table 1 to work out the smallest circuit of a particular speed, or the fastest circuit of a particular size. In any case declarative languages such as Ruby should come into more wide-spread use, as the associated tools begin to reach maturity, and as enthusiasm for declarative languages is growing fast in academic and industrial institutions [15].

The work described in this paper can be extended in several ways. First, we have shown that it is possible to describe dynamically reconfiguring computations using Ruby [11], but more experience needs to be gained in this area. Next, we are developing a better integration of our tools with other tools for supporting an evolutionary approach to custom computing. Finally, we are exploring incremental methods for producing systems with both hardware and software components [12]; declarative techniques should become an indispensable part of our framework.

Acknowledgements

Thanks to Jonathan Saul for comments on an earlier draft. Many colleagues and students have contributed to the Rebecca system; I am particularly grateful to Tim Cheung, Shaori Guo, Cedric Hui, Quentin Miller, Corin Pitcher and Teddy Wu for their efforts. The support of Oxford University Hardware Compilation Research Group, Oxford Parallel Applications Centre, ESPRIT OMI/HORN (P7249) project, Scottish Enterprise and Xilinx Development Corporation is gratefully acknowledged.

References

[1] J.M. Arnold and D.A. Buell, “VHDL programming on Splash2,” in *More FPGAs*, W. Moore and W. Luk (eds.), Abingdon EE&CS Books, 1994, pp. 182–191.

[2] R. Bird and P. Wadler, *Introduction to Functional Programming*, Prentice-Hall International, 1988.

[3] S. Guccione, *List of FPGA-based Computing Machines*, World-Wide Web document, September 1994.

[4] B.K. Fawcett, “Tools to speed FPGA development,” *IEEE Spectrum*, Nov. 1994, pp. 88–94.

[5] F. Hoeg, N. Mellergaard and J. Staunstrup, “The priority queue as an example of hardware/software codesign,” *Proc. Third International Workshop on Hardware/Software Codesign*, IEEE Computer Society Press, 1994, pp. 81–88.

[6] G. Jones and M. Sheeran, “Circuit design in Ruby,” in *Formal Methods for VLSI Design*, J. Staunstrup (ed.), North-Holland, 1990, pp. 13–70.

[7] C.E. Leiserson, “Systolic priority queues,” in *Proc. Caltech Conf. on VLSI*, Caltech Computer Science Dept., 1979, pp. 199–214.

[8] W. Luk and G. Brown, “A systolic LRU processor and its top-down development,” *Science of Computer Programming*, Vol. 15, No. 2–3, December 1990, pp. 217–233.

[9] W. Luk, “Optimising designs by transposition,” in *Designing Correct Circuits*, G. Jones and M. Sheeran (eds.), Springer-Verlag, 1991, pp. 332–354.

[10] W. Luk and I. Page, “Parametrizing designs for FPGAs,” in *FPGAs*, W. Moore and W. Luk (eds.), Abingdon EE&CS Books, 1991, pp. 284–295.

[11] W. Luk, “Systematic serialisation of array-based architectures,” *Integration*, Vol. 14, No. 3, 1993, pp. 333-360.

[12] W. Luk and T. Wu, “Towards a declarative framework for hardware-software codesign,” *Proc. Third International Workshop on Hardware/Software Codesign*, IEEE Computer Society Press, 1994, pp. 181–188.

[13] I. Page and W. Luk, “Compiling occam into FPGAs,” in *FPGAs*, W. Moore and W. Luk (eds.), Abingdon EE&CS Books, 1991, pp. 271–283.

[14] R.G. Shoup, “Parameterized convolution filtering in an FPGA,” in *More FPGAs*, W. Moore and W. Luk (eds.), Abingdon EE&CS Books, 1994, pp. 274–280.

[15] S. Thompson and P. Wadler (eds.), “Functional programming in education,” *Journal of Functional Programming*, Vol. 3, No. 1, 1993.