In *Proc. IEE Colloquium on Field Programmable Gate Arrays – Technology and Applications*, Ref. 1993/037, pp. 9.1–9.4, IEE, February 1993.

HARDWARE COMPILATION FOR FPGAs: IMPERATIVE AND DECLARATIVE APPROACHES FOR A ROBOTICS INTERFACE

Ian Page, Wayne Luk[1] and Henry Lau[2]

Recent improvements in the performance of Field-Programmable Gate Arrays provide an opportunity for rapid construction of special-purpose hardware accelerators. There is little support at present, however, for implementation in such accelerators of algorithms expressed in high-level languages: in most cases users must translate their algorithms into circuit diagrams or state machines before using any computer-based tools. For complex designs these methods become tedious and error-prone; they are incapable of exploiting the structure of a high-level program for parametrising and transforming designs. Moreover, while graphical representations may offer useful visual feedback, they can be tedious to create or to modify.

We advocate a language-based approach for developing hardware accelerators. Two languages are briefly covered in this paper: occam [1] and Ruby [2]. Occam is a simple imperative language, with commands like assignment, sequential composition, conditional and iteration which are found in most imperative languages. In addition, it has commands for parallel composition of program fragments and for synchronised communication between them. Ruby, on the other hand, is a declarative language for expressing designs as functions and relations; a single Ruby program can often be used to produce a variety of architectures with different trade-offs in size and performance [3].

In the following, we illustrate the use of occam and Ruby in developing a shaft encoder interface. In the joint of a robot arm, a shaft encoder measures the angle of each shaft by reading two output signals generated from photo-sensitive detectors. The light input to these detectors is interrupted by a fine pattern of transparent and opaque regions on a glass disc. Rotating the encoder disc results in two digital pulse streams, and the shaft encoder interface must deduce from these streams the direction of rotation and position. While special-purpose devices such as the Texas Instrument THCT2000 can be used in the interface, the resulting system has a low bandwidth and a high chip-count. Our task is to develop a new interface, based on FPGAs, with a higher speed of operation, higher accuracy, additional functionality, smaller physical size, lower development cost, reduced development time, and with increased flexibility.

<u>Occam solution</u>. The following is a fragment of an occam module which converts the two-bit values from the shaft encoder into a number which represents the angular position of the shaft. To make it easier to read, we have slightly changed the syntax of occam.

[1]Programming Research Group, Oxford University Computing Laboratory, 11 Keble Road, Oxford
[2]Robotics Research Group, Department of Engineering Science, Oxford University, Parks Road, Oxford

```
SEQ
  $encoder$ ? $current$
  IF $current \neq previous$
    THEN
      IF $current$ [0] = $previous$ [1]
        THEN
          $angle$ := $angle$ $+1$
        ELSE
          $angle$ := $angle$ $-1$
  $previous$ := $current$
```

This program fragment does three things in sequence. First, it takes the current two-bit output value from the shaft encoder device and stores it in a variable called $current$. Second, it determines whether the encoder has moved since it was last looked at by comparing the value in $current$ with the value in the variable $previous$. If it has moved, then the next comparison determines if the shaft has moved one unit clockwise or anticlockwise, and this is recorded by incrementing or decrementing the value in the variable $angle$. The third and final action is to record the $current$ value in the variable $previous$ for the next iteration. Provided that these three statements are repeatedly executed much faster than the shaft encoder could ever move, we can be certain that the $angle$ variable is a true representation of the physical position of the robot arm.

This module, together with the rest of the program, is compiled automatically into a network of gates and registers, following the strategy expounded in [5]. Occam differs from most imperative languages – such as C – in that it obeys a collection of algebraic laws, so that the compilation process can be seen as applying these laws systematically to turn a user program into a Normal Form program. The Normal Form program that corresponds to the occam description for encoder handling is given below; note that "$S_1 \lhd\ b\ \rhd S_2$" means "if b then S_1 else S_2".

```
WHILE $\neg$ $finished$
  PAR
    $previous$ := $current \lhd c2 \vee c3 \vee c4 \rhd previous$
    $current$  := $encoder \lhd (start \vee c0) \wedge encoder\_rdy \rhd current$
    $angle$    := $(angle{+}1 \lhd incr \rhd angle{-}1) \lhd decr \vee incr \rhd angle$
    $c0$       := $(start \vee c0) \wedge \neg\ encoder\_rdy$
    $c1$       := $(start \vee c0) \wedge encoder\_rdy$
    $c2$       := $decr$
    $c3$       := $incr$
    $c4$       := $c1 \wedge \neg\ changed$
    $finished$ := $c2 \vee c3 \vee c4$
    $start$    := 0
WHERE
    $bitdiff$  = $current[1] \neq previous[0]$
    $changed$  = $current \neq previous$
    $incr$     = $(c1 \wedge changed) \wedge bitdiff$
    $decr$     = $(c1 \wedge changed) \wedge \neg\ bitdiff$
```

A Normal Form program consists of a single simultaneous assignment embedded in a loop. Everything on the left-hand side of the assignment can be interpreted as registers, and everything on the right-hand side can be interpreted as a set of logic gates. In other words, a Normal Form program contains the same information as a circuit diagram; this enables the smooth transition from the world of computer programs to the world of electronic circuits with a guarantee that no errors are introduced by crossing this boundary.

The Normal Form program above has been slightly massaged to make it more readable. Normally this program would not be seen by a human; its purpose is to provide a behavioural interpretation for the netlist produced by the compiler. The netlist is then turned into FPGA programming information using automatic place-and-route software. Thus the entire procedure from user program to FPGA implementation can, in principle, be made fully automatic and provably correct.

A new shaft encoder interface board has been prototyped to test the result of the compilation. The board consists of little more than a Xilinx 3090 FPGA, a chip to communicate directly with the control computer, and a few non-digital components necessary for each shaft encoder. It took only a few hours to design this board, whereas the previous one took over six weeks. One benefit of using FPGAs is that the physical construction of the hardware can be started – and may even be completed – before the full specification of the system is available. This allows us to defer many design decisions, and to achieve a high-degree of product flexibility. In our case, the same encoder interface board has been re-used a number of times with different interface circuits without resoldering a single wire!

Ruby solution. There are no assignment statements in a Ruby program; instead, one considers functions and relations on the state and on the inputs and the outputs of a system. Computations are described by relations in the form $x\ R\ y$, so for example an incrementer is given by $x\ inc\ (x+1)$, while a decrementer is given by $dec = inc^{-1}$, since $x\ R^{-1}\ y \Leftrightarrow y\ R\ x$. Projection relations, such as π_2, can be used to extract a component from a sequence: $\langle x, y\rangle\ \pi_2\ y$. There are also relations, like lsh (left shift), for rearranging the hierarchy in a sequence: $\langle\langle x, y\rangle, z\rangle\ lsh\ \langle x, \langle y, z\rangle\rangle$.

Ruby has a number of operators for structuring designs. An operator that we shall require later is a conditional construct, given by $\langle b, x\rangle\ (\mathsf{cond}\ Q\ R)\ y \Leftrightarrow (x\ Q\ y) \lhd b \rhd (x\ R\ y)$ (recall that $X \lhd b \rhd Y$ means if b then X else Y). The construction of composite designs is facilitated by the composition operator: $x\ (Q;R)\ y \Leftrightarrow \exists s.\ (x\ Q\ s)\ \&\ (s\ R\ y)$. An array of n copies of R formed by repeated composition is given by R^n; this operator can be defined by recursion, $R^1 = R$ and $R^{i+1} = R^i\ ;\ R$.

The first task of a Ruby user is to express a design in relations, using operators like composition and conditional as required. As an illustration, given that b_0 and b_1 are respectively the conditions "$current \neq previous$" and "$current$ [0] = $previous$ [1]", and that a and a' correspond to the values of the $angle$ variable in the occam code before and after the assignment, then a state-transition relation for the encoder interface would be

$$\langle\langle b_0, b_1\rangle, a\rangle\ count\ a' \quad \Leftrightarrow \quad (a' = a) \lhd b_0 \rhd (a' = a+1 \lhd b_1 \rhd a' = a-1).$$

Let $condid = \mathsf{cond}\ inc\ dec$. The relation $count$ can then be expressed using Ruby constructs as follows:

$$\begin{aligned}
\langle\langle b_0, b_1\rangle, a\rangle\ count\ a' &\Leftrightarrow (a' = a) \lhd b_0 \rhd (a\ inc\ a' \lhd b_1 \rhd a\ dec\ a') \\
&\Leftrightarrow (a' = a) \lhd b_0 \rhd (\langle b_1, a\rangle\ condid\ a') \\
&\Leftrightarrow (\langle b_1, a\rangle\ \pi_2\ a') \lhd b_0 \rhd (\langle b_1, a\rangle\ condid\ a') \\
&\Leftrightarrow \langle b_0, \langle b_1, a\rangle\rangle\ (\mathsf{cond}\ \pi_2\ condid)\ a' \\
&\Leftrightarrow \langle\langle b_0, b_1\rangle, a\rangle\ (lsh\ ;\ \mathsf{cond}\ \pi_2\ condid)\ a',
\end{aligned}$$

hence $count = lsh; \mathsf{cond}\ \pi_2\ condid$.

With practice, one can write down a Ruby expression reasonably quickly without going through a detailed derivation like the one above. A Ruby interpreter, which can handle mixed numerical and symbolic simulation, can be employed to check the behaviour of Ruby programs. A program with the desired behaviour can be optimised by correctness-preserving transformations, using strategies such as pipelining and serialisation, to achieve a higher speed or to reduce the circuit size [3].

As an example of a Ruby transformation, provided that $Q;R = R;Q$, one can use induction on n to prove that $Q;R^n = R^n;Q$. This result can then be used to derive a distributive theorem for repeated composition (Figure 1):

$$(Q\ ;\ R)^n \quad = \quad Q^n\ ;\ R^n.$$

Q
R
Q
R
Q
R
(a)
Q
Q
Q
R
R
R
(b)

Figure 1 (a) $(Q\ ;\ R)^n$, (b) $Q^n\ ;\ R^n$, with $n = 3$.

This theorem applies when, for instance, Q and R are operations that add or multiply the input by a constant, since addition and multiplication are commutative. In a more complex circuit model, one can regard Q as a combinational circuit and R as a register; then the left-hand side of the above equation can be considered as a pipelined version of the expression on the right-hand side.

We have developed simple rewriting engines to carry out Ruby transformations automatically. There are also prototype compilation tools [4] to generate, from a Ruby expression, configuration data for Xilinx, Algotronix and Concurrent Logic devices.

While FPGAs are getting larger, sometimes it is still useful to be able to control the layout of a design manually rather than leaving it to the automatic placement and routing software. There is a variant of Ruby, called OAL, which allows users to specify the exact placement and routing of components in a generic manner [4]. OAL has been used in implementing two 32-bit counters and the associated decision logic in an Algotronix CAL1024 FPGA for the shaft encoder interface described earlier; in contrast, a THCT2000 has two 8-bit cascadable counters. A Ruby to OAL compiler is currently under development.

Summary. Let us review the common features and the differences of the two compilation approaches reported in this paper. Both occam and Ruby are designed to enable the clear structuring of designs and reasoning about them. Both of them are based on a sound theoretical framework, which can be used in proving the correctness of the algebraic transformations carried out in the compilation process. Most programs in occam and in Ruby can also be executed – sometimes with symbolic data – to provide a useful feedback of their behaviour.

Occam is more widely used than Ruby, because of its association with the transputer and because of its similarity with conventional sequential programming languages. Moreover, it is possible to compile an occam program partly into hardware for the FPGA, and partly into object code running on a transputer. Occam also has a more flexible synchronisation scheme for communication, although as a result it has a more complicated semantics than Ruby. There are designs, such as those involving data-dependent iterations, which can be cumbersome to describe in Ruby but pose no such problem in occam. On the other hand, architectures with a uniform structure, like systolic systems, can often be captured succinctly in Ruby and implemented efficiently using OAL. The ability to control the layout of circuits without lengthy execution of placement and routing software with the Ruby/OAL approach is also very attractive. We are currently studying various methods of combining the two languages in a coherent manner to get the best of both worlds.

Acknowledgements.

Thanks to Mike Spivey for comments on an earlier draft. The support of Esprit OMI MAP and OMI HORN projects, Scottish Enterprise, Algotronix Limited and Concurrent Logic, Inc. is gratefully acknowledged.

References.

[1] G. Jones, *Programming in occam*, Prentice Hall International, 1987.

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

[3] W. Luk, “Transformation techniques for serial array design,” in *Proc. International Conference on Application-Specific Array Processors*, J.A.B. Fortes et. al. (eds.), IEEE Computer Society Press, 1992, pp. 574–588.

[4] 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.

[5] 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.