Pebble: A Language For Parametrised and Reconfigurable Hardware Design

Wayne Luk and Steve McKeever

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

Abstract. Pebble is a simple language designed to improve the productivity and effectiveness of hardware design. It improves productivity by adopting reusable word-level and bit-level descriptions which can be customised by different parameter values, such as design size and the number of pipeline stages. Such descriptions can be compiled without flattening into various VHDL dialects. Pebble improves design effectiveness by supporting optional constraint descriptions, such as placement attributes, at various levels of abstraction; it also supports run-time reconfigurable design. We introduce Pebble and the associated tools, and illustrate their application to VHDL library development and reconfigurable designs for Field Programmable Gate Arrays (FPGAs).

1 Introduction

Many hardware designers recognise that their productivity can be enhanced by reusable designs in the form of library elements, macros, modules or intellectual property cores. These components are developed carefully to ensure that they are efficient, validated and easy to use. Several development systems based on Java [1], Lola [2], C [3], ML [5], VHDL and Ruby [7] have been proposed. While the languages in these systems have their own goals and merits, none seems to meet all our requirements of:

1. having a simple syntax and semantics;
2. allowing a wide range of parameters in design descriptions;
3. providing support for both word-level design and bit-level design;
4. supporting optional constraint descriptions, such as placement attributes, at various levels of abstraction;
5. including facilities for developing designs reconfigurable at run time.

From our previous work [7] and others, it is also important for design tools to:

6. produce reusable hardware libraries in industrial-standard languages;
7. facilitate multiple means of validation, from formal verification to executing on a hardware platform;
8. enable automatic generation of documentations.

The purpose of this paper is to introduce a language, called Pebble, which is designed to meet the above requirements. Section 2 provides an overview of Pebble, showing how it meets requirements 1–3. Section 3 outlines the development tools for Pebble on which the design flow is based, showing how requirements 6–8 can be satisfied. Section 4 deals with requirement 4: it describes how placement constraints can be captured and how descriptions such as `ABOVE` and `BESIDE` provide a useful abstraction. Section 5 presents an approach for developing reconfigurable designs in Pebble, covering requirement 5. User experience with Pebble is reported in Section 6, while concluding remarks are given in Section 7.

2 Language Overview

Pebble is an alias for *Parametrised Block Language*. The two primary objectives for Pebble are to facilitate the development of efficient and reusable designs, and to support the development of designs involving run-time reconfiguration. Much of our previous work is based on VHDL, which has been used for both library development [7] and simulation of reconfigurable components [11].

The complexity of VHDL and the associated tools, however, has led us to believe that a simpler approach will provide a better foundation on which to build abstractions and tools. A simple language would be both easy to learn and to use. More importantly, it would form a core language satisfying our immediate requirements while amenable to extensions. Moreover, since most VHDL vendors have their own dialect of VHDL, it would be easier to generate vendor-specific VHDL from a single standard library database than to maintain different library databases, one for each VHDL dialect. In any case, the complexity of existing industrial languages such as Verilog or VHDL makes them difficult to include experimental features, such as language support for run-time reconfiguration.

Pebble can be regarded as a much simplified variant of structural VHDL. It provides a means of representing block diagrams hierarchically and parametrically. The basic features of Pebble are outlined below [10].

- A Pebble program is a block, defined by its name, parameters, interfaces, local definitions, and its body.
- The block interfaces are given by two lists, usually interpreted as the inputs and outputs. An input or an output can be of type `WIRE`, or it can be a multi-dimensional vector of wires. A wire can carry integer or boolean values.
- A primitive block has an empty body; a composite block has a body containing the instantiation of composite or primitive blocks in any order. Blocks connected to each other share the same wire in the interface instantiation.
- For hardware designs, the primitive blocks can be bit-level logic gates and registers, or they can, like an adder, process word-level data such as integers or fixed-point numbers; the primitives depend on the availability of corresponding components in the domain targeted by the Pebble compiler.
- The `GENERATE-IF` statement enables conditional compilation, while the `GENERATE-FOR` statement allows the concise description of regular circuits.

Pebble has a simple, block-structured syntax. As examples, Fig. 1 contains a Pebble description of a multiplexor which is a primitive component for Xilinx

```
BLOCK mux [c,x,y:WIRE] [z:WIRE]
BEGIN
END;
```

Fig. 1. A multiplexor description in Pebble, with control input `c`, data inputs `x` and `y` and output `z`. The empty body indicates that it is a primitive block.

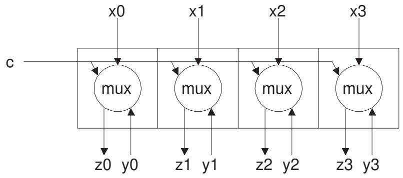

Fig. 2. An array of multiplexors described by the Pebble program in Fig. 3.

6200 FPGAs, while Fig. 3 describes the multiplexor array in Fig. 2, provided that the size parameter `n` is 4.

In more complex descriptions, the parameters in a Pebble program can include the number of pipeline stages or the pitch between neighbouring interface connections [7]. Different network structures, such as tree- or butterfly-shaped circuits, can be described parametrically by indexing the components and wires.

Pebble supports the use of annotations and constraint descriptions. Annotations contain optional information that does not affect the functional behaviour of Pebble programs. The use of annotations for guiding the Pebble compiler and for automatic documentation generation will be described in Section 3. The use of constraint descriptions to provide, for instance, abstract and concrete placement information will be presented in Section 4.

The semantics of Pebble depends on the behaviour of the primitive blocks and their composition in the target technology. Currently a synchronous circuit

```
BLOCK muxarray (n:GENERIC) [c:WIRE, x,y:VECTOR (n-1..0) OF WIRE]
                           [z:VECTOR (n-1..0) OF WIRE]
      VAR   i
BEGIN
  GENERATE FOR i = 0..(n-1) DO
    mux [c,x(i),y(i)] [z(i)]
END;
```

Fig. 3. A description of an array of multiplexors (Fig. 2) in Pebble. The external input `c` is used to provide a common control input for each mutiplexor.

model is used in our tools (Section 3), and special control components for modelling run-time reconfiguration are also supported (Section 5). However, other models can be used if desired. Indeed Pebble can be used in modelling any block-structured systems, not just electronic circuits.

Advanced features of Pebble include support for modules which improves reusability and facilitates interface to components in other languages, including behavioural descriptions. Discussions about these features are beyond the scope of this paper.

3 Development Tools and Design Flow

We have developed a compiler for Pebble which can produce either a flattened netlist for simulation, or a parametrised description in structural VHDL. Pebble programs can be compiled into the netlist format for the Rebecca simulator, which can be used for cycle-accurate numerical or symbolic simulation at word-level, bit-level, or a mixture [7]. Automatic mapping between word-level and bit-level blocks is under development. Pebble descriptions can also be translated into formats suitable for verification systems such as HOL [4].

Pebble programs can be compiled into parametrised VHDL while preserving their hierarchy and parametrisation. The resulting VHDL code may contain compiler-generated names, but they can be replaced by user-specified names annotated in the Pebble source code. Section 6 includes more details about the parametrised VHDL libraries generated from Pebble; users of these VHDL libraries do not need to know Pebble.

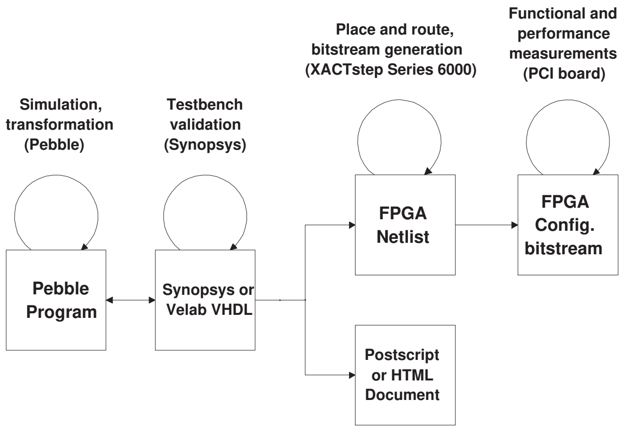

Fig. 4. Design flow for our Pebble-based system. Synopsys and Velab are VHDL tools, and XACTstep Series 6000 is the implementation tool for Xilinx 6200 FPGAs.

Fig. 4 shows the major elements in our design flow. Synopsys is a well-known industrial system which deals with VHDL synthesis; Velab and XACTstep Series

6000 are implementation tools for Xilinx 6200 FPGAs; and a PCI-based platform [9] for evaluating designs. We have also developed a Library Documentation Tool (LDT), which automatically produces library documentation in various formats such as Postscript and HTML [7]. The documentation is generated from information annotated in a specific format in the Pebble source; this method reduces the number of files to be maintained.

4 Constraint Description and Abstraction

It is often useful to have the ability to include information about layout or timing in a hardware description. Such information provides a means of guiding design tools to produce an optimised design, or to generate improved estimates of design properties such as critical path delay or reconfiguration time.

Placement information is particularly important for FPGAs for two reasons. First, optimal resource usage is often necessary in FPGA design, since the density and speed of FPGAs are much less than those of custom integrated circuits in similar technologies. Second, precise control over the placement of components is required to minimise reconfiguration time, since components at identical locations common to two successive configurations do not need to be reconfigured.

A number of design languages, such as Lola [2], VHDL [7] and Lava [12], include mechanisms for specifying placement information. Pebble provides a facility similar to these languages. For example, a halfadder containing an `xor2` gate beside an `and2` gate can be described by the Pebble program in Fig. 5.

```
BLOCK hadd (x,y:GENERIC) [a,b:WIRE] [cout,sum:WIRE]
BEGIN
  xor2 [a,b] [sum ] MAP rloc IS "X,x,Y,y,";
  and2 [a,b] [cout] MAP rloc IS "X,(x+1),Y,y,"
END;
```

Fig. 5. A Pebble program describing a halfadder with an `xor` gate on the left of an `and` gate. The values `x` and `y` denote the (`x`,`y`) co-ordinates of a block; for instance if `x`=8 and `y`=3, then the `xor` gate will be placed at (8,3) and the `and` gate at (9,3).

While placement information helps to optimise the layout, it is usually tedious and error-prone to specify. Pebble provides high-level descriptions for placement constraints, abstracting away the low-level details. These descriptions are compile-time directives for the Pebble compiler to project co-ordinates onto designs, generating a tree representing placement possibilities. The two main descriptions are `BESIDE`, which places two or more blocks beside each other, and `ABOVE`, which places blocks vertically. These descriptions allow blocks to be placed relatively to each other, without the user providing the coordinates of their locations. Using them, the halfadder example in Fig. 5 becomes the one in Fig. 6(a), while the multiplexor array in Fig. 3 becomes the program in Fig. 6(b).

```
(a) BLOCK hadd [a,b:WIRE] [cout,sum:WIRE]
    BEGIN
      BESIDE (xor2 [a,b] [sum ],
              and2 [a,b] [cout])
    END;

(b) BLOCK muxarray (n:GENERIC) [c:WIRE, x,y:VECTOR (n-1..0) OF WIRE]
                               [z:VECTOR (n-1..0) OF WIRE]
          VAR   i
    BEGIN
      BESIDE FOR i = 0..(n-1) DO
          mux [c,x(i),y(i)] [z(i)]
    END;
```

Fig. 6. (a) A Pebble program using BESIDE to describe the halfadder shown in Fig. 5. (b) A Pebble program describing an array of multiplexors placed beside one another, as shown in Fig. 2. The only alteration to the Pebble description in Fig. 3 is to replace the reserved word GENERATE by BESIDE.

To illustrate further how ABOVE and BESIDE abstract from placement details, consider the description in Fig. 7(a) which specifies that blockC will be placed above blockA and blockB. Without using ABOVE and BESIDE, to place blockC one needs to calculate the width of blockA and the larger of the height of blockA and blockB as in Fig. 7(b). The calculations may involve the generic parameters to these blocks, hence Fig. 7(a) provides a significant simplification.

```
(a)   ABOVE ( blockC [cin] [cout],
              BESIDE ( blockA [ain] [aout],
                       blockB (n) [bin] [bout] ) );

(b)   blockA     [ain] [aout] MAP rloc IS "X,x,Y,y,";
      blockB (n) [bin] [bout] MAP rloc IS "X,(x+widthA),Y,y";
      GENERATE
        IF heightA >= heightB THEN
          blockC [cin] [cout]  MAP rloc IS "X,x,Y,(y+heightA)" END
        IF heightA <  heightB THEN
          blockC [cin] [cout]  MAP rloc IS "X,x,Y,(y+heightB)" END;
```

Fig. 7. (a) A description with nested ABOVE and BESIDE. (b) An alternative with explicit co-ordinates. heightA and widthA denote the height and width of blockA.

A more complex example is a pipelined incrementer [7]. A fully-pipelined incrementer, with a rectangular layout suitable for Xilinx 6200 FPGAs, is shown in Fig. 8. In this design, the core array of halfadders are placed along the diagonal of the block, with triangular-shaped arrays of registers for signal re-alignment placed above and below the halfadder cells.

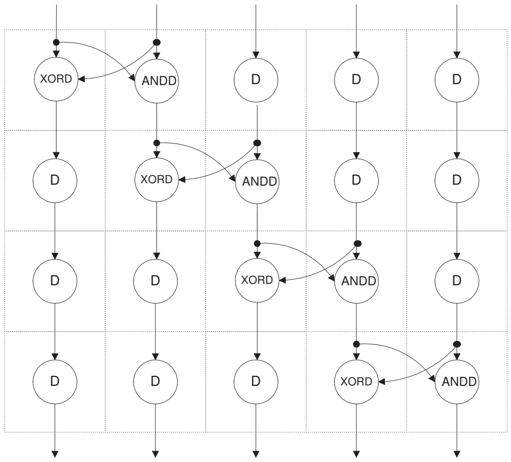

Fig. 8. A fully-pipelined 4-bit incrementer. The components `XORD` and `ANDD` correspond to `xor` and `and` gates with a latched output.

```
ABOVE FOR i = 0..(n-1) DO
  BESIDE (
    BESIDE FOR j = 0..(i-1) DO
      D [w(i,j)] [w(i+1,j)],
    XORD [w(i,i),w(i,i+1)] [w(i+1,i)],
    ANDD [w(i,i),w(i,i+1)] [w(i+1,i+1)],
    BESIDE FOR j = (i+2)..n DO
      D [w(i,j)] [w(i+1,j)] ) ;
```

Fig. 9. Pebble description of the pipelined incrementer in Fig. 8, where `n` is 4. `w(0,0)..w(0,n)` are the input wires for the top row of halfadders, and `w(0,0)` is the carry-in. `w(n,0)..w(n,n)` are the outputs at the bottom, and `w(n,n)` is the carry-out.

The corresponding Pebble code (Fig. 9) contains an `ABOVE` loop, the body of which contains a `BESIDE` of the four components found on each row of the array of cells. The first component is itself a `BESIDE` loop of registers `D`, whose size increases from zero for the top row of cells to `n-1` for the bottom row. On the right of this `BESIDE` loop, there are the `xor` and `and` gates with a register at each of their outputs, which correspond to `XORD` and `ANDD` in Fig. 8. The fourth component in a row of cells is another `BESIDE` loop of registers, whose size decreases from `n-1` for the top row of cells to zero for the bottom row. For clarity, the `clock` and `clear` signals for registers are not shown. The corresponding code with explicit co-ordinates is too large to be included here.

It is possible to extend this design so that the number of pipeline stages can be controlled by a parameter [7]. The resulting description can be used to produce implementations with different trade-offs in resource usage and performance.

5 Support for Reconfiguration

Pebble supports the development of run-time reconfigurable circuits based on the model for reconfigurable designs in [8]. In this model, a component that can be configured to behave either as A or as B is described by a network with A and B connected between two control blocks. The control blocks, RC_DMux and RC_Mux, route the data and results from the external ports x and y to A or B depending on the value of *cond* (Fig. 10). Each control block will be mapped either into a real multiplexor or a demultiplexor to produce a single-cycle reconfigurable design, or into virtual ones which model the control mechanisms for replacing one configuration by another. If the reconfiguration sequence is known at compile time, then control blocks which model the run-time selection of components in a particular sequence can be used [6]. At present Pebble descriptions are translated into the EDIF format, for which a set of tools has been developed to produce reconfigurable designs [8]. We are exploring language support for reconfiguration by having, for instance, a `RECONFIGURE-IF` statement (Fig. 11).

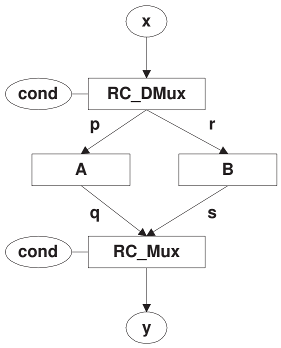

Fig. 10. A design that can behave either as A or as B, depending on the value of the input wire *cond* connected to the control blocks RC_DMux and RC_Mux.

6 Experience

Pebble has been in development since early 1997. In the following we report our experience with Pebble in three applications: hardware library development,

```
(a)   BLOCK AB [x,cond:WIRE] [y:WIRE]
      BEGIN
         RECONFIGURE IF cond THEN
            A [x] [y]  END
         ELSE
            B [x] [y]  END
      END;

(b)   BLOCK AB [x,cond:WIRE] [y:WIRE]
      VAR p, q, r, s: WIRE
      BEGIN
         RC_DMux [cond,x] [p,r];
         A [p] [q];
         B [r] [s];
         RC_Mux [cond,q,s] [y]
      END;
```

Fig. 11. (a) A Pebble block showing how the `RECONFIGURE IF` statement captures the circuit in Fig. 10. (b) An alternative Pebble block describing the same design, with variables p, q, r and s representing the internal wires.

implementation of video-processing hardware, and student design projects.

Our previous work on parametrised hardware libraries involved VHDL87 from Synopsys [7]. Since then Xilinx introduced Velab, a fast VHDL93 elaborator which is issued free of charge. Because of the differences in the two VHDL dialects, many of our libraries have to be rewritten. Following the design flow in Fig. 4, over 30 libraries have now been recast in Pebble and most can target both Synopsys and Velab.

Pebble libraries have been used in various applications, particularly for video processing. To enable video experiments, a real-time video interface has been built for a PCI-based board with a Xilinx 6200 FPGA and two megabytes of memory [9]. Case studies include linear and non-linear filtering, edge detection, image rotation, colour identification and motion detection.

Pebble has also been used in many student projects. Because of the simplicity of the language and the tools, students usually master the basic techniques rapidly and complete complex designs much faster than using VHDL. As an example in a group project for first-year undergraduates, a convolver which took more than four weeks to design in VHDL was finished in Pebble within the first week of the project.

7 Concluding Remarks

Pebble has served as a focus for our research on languages and tools for developing hardware in general and reconfigurable circuits in particular. Its simplicity facilitates design construction, parametrisation and validation. It supports circuit descriptions at various levels of abstraction, allowing designers to selectively control a design step such as placement when desired. Designs can be captured and analysed at different levels of detail, since they can be expressed using non-implementable components such as RC_Mux (Section 5) and converters between word-level and bit-level data. Current and future work includes support for passing blocks as parameters and for polynomial constraints, optimisations such as retiming and partial evaluation, interface to high-level tools and run-time environments, and backends for various FPGAs and custom VLSI implementations.

Acknowledgements

Our thanks to the many Pebble developers, users and advisors, particularly Florent Dupont-De-Dinechin, Douglas Grant, Patrick Mackinlay, Richard Sandiford, Seng Shay Ping, Nabeel Shirazi, Dimitris Siganos, David Stacey and Markus Weinhardt, for their help and suggestions. The support of Xilinx, Interval Research, Hewlett Packard Laboratories Bristol and the UK Engineering and Physical Sciences Research Council (Grant Number GR/L24366, GR/L54356 and GR/L59658) is gratefully acknowledged.

References

1. P. Bellows and B. Hutchings, "JHDL – an HDL for reconfigurable systems", in *Proc. FCCM98*, IEEE Computer Society Press, 1998.
2. S. Gehring and S. Ludwig, "The Trianus system and its application to custom computing", in *Field-Programmable Logic, Smart Applications, New Paradigms and Compilers*, LNCS 1142, Springer, 1996.
3. M. Gokhale and E. Gomersall, "High-Level compilation for fine-grained FPGAs", in *Proc. FCCM97*, IEEE Computer Society Press, 1997.
4. M.J.C. Gordon, "Why Higher-Order Logic is a good formalism for specifying and verifying hardware", in *Formal Aspects of VLSI Design*, G. Milne and P.A. Subrahmanyam (eds.), North Holland, 1986.
5. Y. Li and M. Leeser, "HML: an innovative hardware description language and its translation to VHDL", in *Proc. CHDL'95*, 1995.
6. W. Luk, "Systematic serialization of array-based architectures", *Integration, the VLSI Journal*, 14(3), February 1993.
7. W. Luk, S. Guo, N. Shirazi and N. Zhuang, "A framework for developing parametrised FPGA libraries", in *Field-Programmable Logic, Smart Applications, New Paradigms and Compilers*, LNCS 1142, Springer, 1996.
8. W. Luk, N. Shirazi and P.Y.K. Cheung, "Compilation tools for run-time reconfigurable designs", in *Proc. FCCM97*, IEEE Computer Society Press, 1997.
9. W. Luk, N. Shirazi, S. Guo and P.Y.K. Cheung, "Pipeline morphing and virtual pipelines", in *Field Programmable Logic and Applications*, LNCS 1304, Springer, 1997.
10. W. Luk, S. McKeever and M. Weinhardt, *A Tutorial Introduction to Pebble*, Technical Report, Imperial College, 1998.
11. P. Lysaght and J. Stockwood, "A simulation tool for dynamically reconfigurable field programmable gate arrays", *IEEE Trans. VLSI*, 4(3), September 1996.
12. S. Singh, *Lava*, http://www.dcs.gla.ac.uk/~satnam/lava/main.html.