

Formal Techniques Assist Protocol-Limited Designs, with a DSP Filter as an Example

R.B. Hughes* J. Southard^λ G. Musgrave[‡]

Abstract Hardware Limited[†]
1, Brunel Science Park,
Kingston Lane,
Uxbridge, Middlesex
UB8 3PQ, UK.

^λ Viewlogic Systems Inc., Fremont, California

[‡] Department of Electrical Engineering and Electronics
Brunel University
Uxbridge, Middlesex UB8 3PH, UK

January 1996

Abstract

This paper examines the application of novel formal methods techniques to assist in the automatic generation of correct RT-level VHDL code. Of particular interest is a class of designs which are “protocol-limited” in their specification, having to meet complex temporal and physical I/O constraints. It is difficult to easily model this in VHDL and a technique which makes this easily realisable is illustrated on an example which is a complex DSP application.

1 Introduction

There has been considerable increase in interest in the last year of formal methods and, in particular, formal verification. In this paper we address the issue of specifying a complex device which has a variety of possible implementations that satisfy the behavioural implementation but not necessarily the performance criteria.

Another factor which is equally as important as performance is physical pin-out limitations which may be present in an implementation, e.g. having 10 data inputs each of 32-bit data would be prohibitive, in many cases, owing to the vast pin requirement of the interface. Guaranteeing

*Email. roger@ahl.co.uk, *Designated contact & Presenter*

[†]Tel. +44 1895 258501, Fax. +44 1895 259437, Telex. 261173G, Email. lambda@ahl.co.uk

conformance with the desired behaviour and being able to examine alternative I/O is a task to which VHDL is not suited as a language.

What is required is a technique whereby the performance criteria can be quickly examined and VHDL with the desired characteristics for the RTL implementation produced. We write here about RT-level VHDL as being the final output because existing synthesis tools can readily synthesise good circuits from well written RTL code.

In this paper we show this technique, which relies on automatic use of theorem proving technology to complete our tasks. The adopted technique is unique and allows for the easy generation of annotated, well-structured and efficient VHDL code, which is easily human readable.

The approach is illustrated by the development of a piece of VHDL which describes a DSP application; in this case the DSP application comes from the image processing domain, an application area which has quite strict performance criteria. The objective is to quickly examine different implementations to see which ones satisfy the performance criteria and to also ensure that the resultant device will have a number of I/O pins within normal fabrication limits. Because the design must satisfy an I/O behaviour and conform to I/O pin fabrication limits, i.e. the design space is constrained to a certain protocol, we call this type of design a "protocol-limited" design. It will be seen that there are many such designs in current industrial practice where the use of the technology illustrated in this paper would be of considerable benefit.

2 Background Material

Writing correct RTL code is very important. Design errors found earlier in the design process are far less costly than design errors found later in the design cycle. Unfortunately, simulation is not adequate alone for illustrating design correctness. The situation is aided if we start at a higher level in the design process, i.e. closer to the human cognitive level of reasoning. For this reason, RTL code is far easier to understand than writing a structural netlist and behavioural code even easier still. However, despite advances in synthesis tools and technology, automatic synthesis from a behavioural level is still quite problematic and it is difficult to decide if the output produced is correct. One of the main reasons for this is that there is a considerable disconnect in the design process, the designer cannot see the relation between the input and output.

Another factor that is of immense importance is that of the suitability of the language for the level of the description provided. We advocate a functional programming style using a language which is able to represent the higher level behavioural concepts far more naturally. We also advocate a specification style which does not initially specify the widths of the inputs and outputs, but reason for any data value being used as an input and an output. Whilst many things are expressed more efficiently in a textual way, the authors believe that a graphical approach to manipulating the specification interactively is a natural way for an engineer to manipulate things.

The techniques we use also allow the designer to interactively manipulate the resulting implementation (prior to VHDL generation) at the graphical level and allow, as illustrated by the case study, the engineer to alter both the resource allocation and scheduling associated with a particular operation. This interaction naturally allows the designer to make mistakes. This is important. In any design process, the designer will often go through several invalid designs until the overall effect of a sequence of changes produces a valid design. Formal methods, especially those based on theorem proving, usually constrain the designer to only produce valid changes to the design. This sometimes has the effect of restricting what the designer can do at any particular point in time. What we have done in the development of the ViewSchedule tool is to produce a means by which the designer is allowed to make several unproven changes before being forced to go through the automatic proof process again. ViewSchedule is a tool which was produced to handle complex scheduling and allocation problems using formal methods to ensure that the implementation satisfies the original behavioural requirement. The formal technology

that it uses to do the proof is that of the advanced LAMBDA toolset, developed by Abstract Hardware. In fact, it uses the theorem proving core of the LAMBDA technology and ensures that the proof process is totally automatic. When the designer has interactively manipulated the implementation, the design is marked as not valid. At this stage it is impossible to produce any VHDL; important, since the VHDL would not necessarily represent the intended behaviour of the design. The designer then asks for the design schedule (both timing and allocation) to be checked. When this has been validated by some simple checks the designer is then allowed to ask for VHDL to be generated. It is at this stage in the process, just before VHDL generation is carried out, that a full automatic formal proof is made. Deferring the full proof until this point allows the interactive process to be very much quicker than it otherwise would have been. Because various checks are made at each stage in the development of the design, the formal proof always concludes successfully; the exception to this would be a bug in the tool. The important feature of the technology is that the designer is assured by virtue of the formal methods that the VHDL does meet the behavioural specification. So, what exactly are formal methods?

3 Formal Methods

We do not aim to go into detail here, but feel it may be useful to give some background as to what our notion of formal methods is and the basic structure of what is involved. The reader is referred to other published works[1, 2, 3, 4, 5] which cover the material presented here in more mathematical detail. In the LAMBDA theorem proving core, a design is represented by a *rule* in the *formal logic*. This rule holds the relationship between four distinct, but tightly interrelated, elements in the design, viz. the original specification, the partial implementation, the work remaining to be done and environmental constraints introduced as a result of various design decisions made in producing the partial implementation. These elements are represented in a rule as follows:-

Provided **the design SATISFIES the environmental constraints**
and **the design SATISFIES the work remaining to be done**
then the partial implementation plus further work SATISFIES the original specification

This is shown in logic as a rule of the form:-

$$\frac{\begin{array}{l} \Delta \vdash \text{environmental constraints} \\ \Delta \vdash \text{work remaining} \end{array}}{\text{partial implementation, } \Delta \vdash \text{original specification}}$$

The formal approach works quite simply. In formal synthesis a rule is created which is a tautological one stating that if an implementation can be found which satisfies the specification then an implementation exists which satisfies the specification. Such a rule is clearly valid and provided valid rules are applied to it then the resulting rule will be valid. The proof process concludes when all the premises (the items above the line) are discharged to TRUTH and the final rule is a proven theorem which states

“Provided I am in this environment, this final implementation SATISFIES the original specification.”

There are several thousand rules within the theorem proving core of the LAMBDA system. The correctness of them is guaranteed by virtue of the fact that they are all based on eight fundamental axioms of Higher Order Logic (HOL); all other rules have been proven in terms of this fundamental set. The interested reader is referred to the following[6, 7] for further information.

4 The Example - a DSP Application

4.1 Image Processing

We consider a digital signal processing application from the image processing domain as one which is particularly suited to illustrating the protocol-limited aspects of this design type. The design is basically a dedicated module used for a larger image processing application algorithm such as edge-detection or noise smoothing.

Informally, the dedicated module is a pixel comparator. The specification for this unit is that it take a 3x3 grid of pixels and compare the centre pixel value with the surrounding eight pixel values. Values to be computed are the average of the surrounding eight pixels, a boolean indicating if the central pixel value is greater than the mean and a value indicating the absolute difference between the central pixel value and the mean. The pixels are labelled as follows:-

```
-----  
| pTL | pTC | pTR |  
-----  
| pCL | pC  | pCR |  
-----  
| pBL | pBC | pBR |  
-----
```

As part of the informal specification, we are told that the center pixel value is made available 3 cycles after the first outer pixel values are made available, pTL and pBR are fixed to appear at time 0, all the other outer pixels are also available at this time but may be consumed later. We also have some performance criteria that require latency to be minimised (maximum 8 cycles after first input available) and allow a high clock frequency to be used. Throughput of the device is more important than overall latency. Designs should be considered for 8-bit data and 24-bit data, total pin count should be no more than 100 pins.

The formal specification, which could also specify the latency of particular outputs (although we have not done so in this case to allow greater freedom in the design space), is quite easily entered either as an equation which is then automatically displayed as a graph, or directly as a graph itself. We firstly concentrate on the behaviour required, which is simply to add the surrounding eight values, divide by eight (a simple shifting operation) and then calculate the absolute difference between the two values. Various operators are defined within the system. The designer sets the required I/O behaviour by pre-scheduling the pTL and pBR inputs to occur at time 0 and pC to occur at time 2. (In the tool these pre-schedules appear in red.) The resulting data-flow specification diagram appears as follows:-

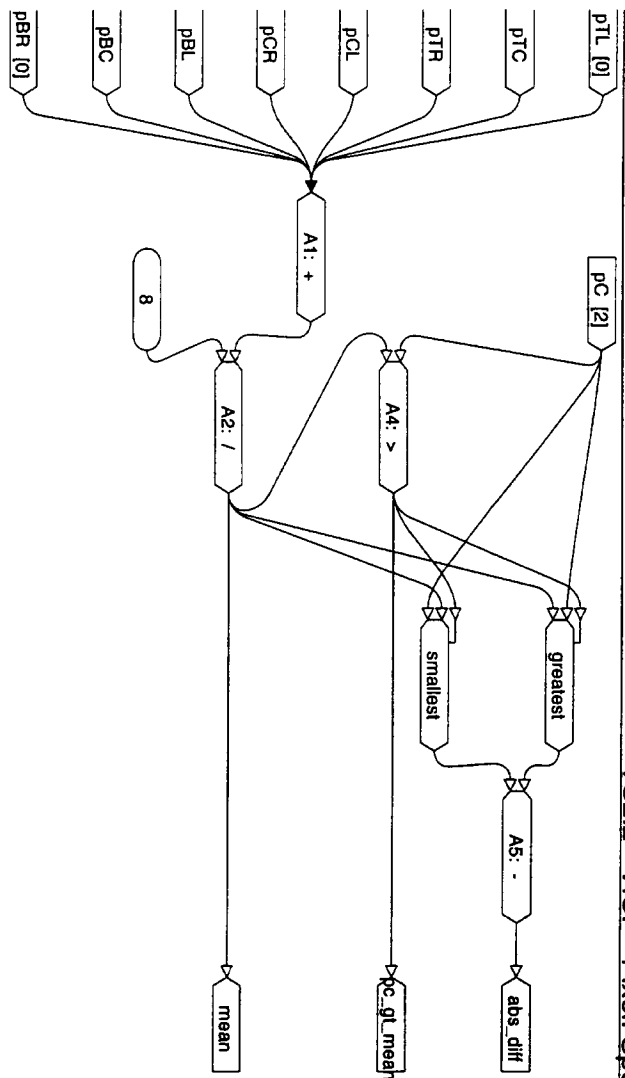
The figure is shown on a following page

The designer then quickly asks for one automatically generated implementation and is given a view of the hardware resources used (minimum component hardware is the automatic response). The parameters returned to the designer are the overall latency of the design, the repeat time of the design (that is the time at which the next set of inputs may be supplied, i.e. we may have a partially or fully pipelined design) and some frequency estimates for the design in terms of maximum estimated clock frequency. This design is shown in 4.1.

The figure to be shown could not be included at time of going to press, it will be shown during the presentation as a Viewfoil

This design has a latency of 8 clock cycles and a repeat rate of 9 clock cycles, i.e. the next inputs can be fed in after 9 cycles. The maximum clock frequency is 322MHz and the data rate for new inputs is 35.8MHz. We have also chosen to take all the surrounding pixels in time slice 0.

VS2.2 - VIUF - Pixel: Specification



4.2 Alternative Designs

We see from the first design that the number of physical data inputs is too large. pTL occurs on one input and pBR on another because they must occur at the same time. This is not the case for pC, we could re-use the input used for pTL for pC because it is free at this time. In fact we will need to do some significant re-use of inputs if we are doing a design for 24-bit colour signals and still wish to keep our pin count down to 100 pins. The current design is acceptable for 8-bit data but not for 24-bit if and only if we are constrained by this I/O pin limitation. Let us first ignore this limitation and instead concentrate on improving the performance of the 8-bit design which does not suffer from pin limitation problems.

Firstly, we see that the design uses just one ALU to perform the large sequence of additions that need to take place to compute the sum prior to computing the mean. We can select the addition operator in the specification and then select *Unallocate* from the menu. This will remove all the additions from this ALU leaving it to carry out just the other tasks. At this stage the large green tick symbol, showing that the design is valid, is replaced by red crossed thorns to indicate that the design is no longer necessarily valid. Dragging the addition operation into the hardware resource area automatically introduces additional ALUs to carry out the addition operations, but it is intelligent enough to only introduce as many as are required so as to allow the addition to be performed as a tree operation. Although additional ALUs appear in the hardware resource list in the top left window of the display, the operations are not shown to occur at any particular point in time; this is because they have not yet been scheduled. Pressing the *Schedule* button forces some aspects of a proof process to take place and the design is shown as validated on successful completion of the scheduling. The resultant latency, repeat time and clock frequencies are 8 and 3 cycles and 112MHz respectively. A new set of inputs can thus be taken at a rate of 37.4MHz. This is not a major improvement. Why?

The scheduling operation preserved the previous scheduling that was optimised for just one ALU. We can now re-schedule everything. To do this we can un-schedule all operations (excluding, as we wish, the pre-schedules, such as pTL) and then once again press "schedule". This time we get results of 3 for the latency and 3 for the repeat time and 112MHz for the clock frequency. This circuit is better for real-time use owing to its much shorter latency; however, *throughput* of data is exactly the same. This is not the end of the optimisation process. The designer can quickly examine which period contains the most combinatorial circuitry and move some of the operations to other cycles and/or hardware resources to try to improve the maximum frequency of the design by minimising the maximal period. The design decisions are checked simply by pressing the *Schedule* button again. This time the latency is 8 (as before) and the repeat time is 4 but the maximum clock frequency has been improved to 322MHz (with the corresponding data input rate at 80.6MHz).

All the different designs carried out here were done in the space of a few minutes. What saving does this give the designer who is writing VHDL? Quite a significant one. The resulting VHDL is shown in the next section.

For the severely "protocol-limited" design of the 24-bit version, we need to restrict ourselves to two physical inputs and three physical outputs ($2 * 24$, 1 bit control) in order to give us a total of 97 ($4 * 24 + 1$) pins or something similar to this. To illustrate the point, we have also done the design with three physical inputs and two physical outputs, i.e. both the mean and the absolute difference appearing on the same physical output but time multiplexed. This is shown in figure 4.2. We can once again develop for minimum hardware, the single ALU case, where we produce a 322MHz clock design with a repeat and latency of 9 cycles, only using three input ports. The mean occurs at time slice 8 and the absolute difference output at time slice 9 on the same output port. If we develop the design for a total of 5 ALUs (as earlier) and we improve the design, we achieve the following:

a 24-bit, 100 I/O pin compliant design with the mean at 8 cycles after the first input (the boolean control output also occurs at this time) and the absolute difference on the same output port at 10 clock cycles. This time we have an overall latency of 10 cycles, a partially pipelined

design with a repeat time of just 3 cycles and a maximum clock frequency of 454MHz with a maximum input data rate of 151MHz. VHDL (this time compatible with Viewlogic tools) is shown in fastp3.vhdl.

A figure was to be shown here, but could not be included at time of going to press, it will be shown as a Viewfoil at the presentation

4.3 Resulting VHDL

The VHDL can be produced either for entry into Synopsys tools or Viewlogic tools. There are two basic forms: scheduled only, which contains only the timing information of when operations are to occur, and scheduled and allocated, which contains the hardware resource information as well. For brevity in this paper we show both types of VHDL output for the result of the first design and the scheduled only VHDL for the two differing 24-bit designs below. It is in the process of generating the VHDL that we select what the bit-width of the various busses will be. This keeps our design general until we want to specialise it.

```
fastp3.vhdl -----  
  
-----  
-- VIEWSCHEDULE: Schedule Only VHDL  
--  
--  
-- Date:   Fri Feb  2 16:28:48 1996  
--  
-- File:   /staff/roger/Doc/Pubs/Camera_R/fastp3.vhdl  
--  
-- Entity: Untitled  
--  
-- Target: ViewArchitect (* note: could have been Synopsys - but different output*)  
--  
  
(* text deleted for brevity *)  
  
-----  
-- Libraries  
-----  
library ieee;  
use      ieee.std_logic_1164.all;  
  
-----  
-- Entity declaration for Untitled  
-----  
  
entity Untitled is  
  port (  
    CLK           : in  STD_ULOGIC;  
    RST           : in  STD_ULOGIC;  
    Input1        : in  NATURAL range 0 to 8388607;  
    Input4        : in  NATURAL range 0 to 8388607;  
    Input7        : in  NATURAL range 0 to 8388607;  
    Output2       : out STD_ULOGIC;  
    Output1       : out NATURAL range 0 to 8388607  
  );  
end Untitled;  
  
-----  
-- Architecture declaration for Untitled
```

```

-----
architecture SCH_ONLY of Untitled is
(* signal declarations deleted for brevity *)
-----
-- Type and signal defined for internal state.
-----
type Untitled_stateType is (VsVHDLstate0,VsVHDLstate1,VsVHDLstate2);
signal VsVHDLstate : Untitled_stateType;

begin

CLOCKED:
  process
    -----
    -- This process contains all signal storages.
    --
    -- In ViewSchedule, storages are shown as dataflow arrows
    -- crossing from one time-slot to another.
    -----

  begin
    wait until RST = '1';

    RESET_LOOP: loop

      -----
      -- Initialise VsVHDLstate asynchronously on reset.
      -----
      if RST = '1' then
        VsVHDLstate <= VsVHDLstate0;
      end if;

      -----
      -- Make assignments on rising clock edge (with repeat of 3)
      -----

      -----
      -- Transition at time-slots 0 -> 1, 3 -> 4, etc. due to repeat of 3
      -----

      wait until (CLK'EVENT and CLK = '1') or RST = '1';
      next RESET_LOOP when RST = '1';

      -----
      -- Assignments at this transition
      -----
      A5_Stored <= A5_ToStore;
      A2_Stored <= A2_ToStore;
      A1_S5_Stored <= A1_S5_ToStore;
      A1_S4_Stored <= A1_S4_ToStore;
      A1_S2_Stored <= A1_S2_ToStore;
      A1_S1_Stored <= A1_S1_ToStore;
      pBR_Stored <= pBR_ToStore;
      pCR_Stored <= pCR_ToStore;
      pTL_Stored <= pTL_ToStore;
      VsVHDLstate <= VsVHDLstate1;

(* other transitions deleted for brevity *)

    end loop;
  end process CLOCKED;

```

```

UNCLOCKED:
  process(Input1, Input4, Input7, VsVHDLstate,
    A5_Stored, A2_Stored, A1_S5_Stored, A1_S4_Stored,
    A1_S2_Stored, A1_S1_Stored, pBR_Stored, pCR_Stored,
    pTL_Stored, A4_Stored, A1_S6_Stored, A1_S3_Stored,
    pBC_Stored, pCL_Stored, pTR_Stored, smallest_Stored,
    greatest_Stored, pC_Prev_Stored, A1_A1_Stored, pC_Stored,
    pBL_Stored, pTC_Stored)
-----
    -- This process contains all combinational logic (C/L).
    --
    -- In ViewSchedule, combinational logic is shown as operator
    -- lozenges in particular time-slots.
    -----

    -- Internal variables
    -----
    variable Alu1, greatest, smallest, A2,
      pC_Prev, Alu2, Alu3, A1_S4,
      A1_S5 : NATURAL range 0 to 8388607;
    variable Alu1_Output2 : STD_ULOGIC;

begin

  case VsVHDLstate is

    when VsVHDLstate0 =>
      -----
      -- C/L required for time-slots 0, 3 etc. due to repeat of 3
      -----
      Alu1 := greatest_Stored - smallest_Stored;
      A5_ToStore <= Alu1;
      A2 := A1_A1_Stored / 8;
      A2_ToStore <= A2;
      A1_S5 := pBC_Stored + pBR_Stored;
      A1_S4 := pCR_Stored + pBL_Stored;
      Alu3 := pTR_Stored + pCL_Stored;
      Alu2 := pTL_Stored + pTC_Stored;
      A1_S5_ToStore <= A1_S5;
      A1_S4_ToStore <= A1_S4;
      A1_S2_ToStore <= Alu3;
      A1_S1_ToStore <= Alu2;
      pBR_ToStore <= Input1;
      pCR_ToStore <= Input4;
      pTL_ToStore <= Input7;

    (* other cases deleted for brevity *)

  end case;
end process UNLOCKED;

end SCH_ONLY;

```

It will have been seen that the VHDL is indeed quite simple to follow and quite efficient. Hopefully, the reader will also appreciate that the “scheduled-only” VHDL is almost behavioural, in that within each cycle the order of operations is not defined, it being left as an optimisation for the following synthesis tool.

5 Benefits

The resulting designs all implemented the general behavioural specification but several alternative performances were shown. The resulting RTL VHDL would have been quite cumbersome to

get right the first time using conventional techniques. The hidden benefit is that the VHDL is formally proven to satisfy the original behavioural specification, i.e. the design has been formally verified via a full logical mathematical proof without the user ever having to see any awkward proof engine.

6 Conclusion

With the introduction of techniques such as outlined in this paper, VHDL writing can become more productive and a larger class of problems can be more efficiently addressed than before. We have shown how ViewSchedule, a tool developed by Abstract Hardware and sold by Viewlogic, directly addresses these issues and brings formal methods within the easy grasp of the average practising engineer

The design example which we chose in the time-critical DSP domain is one of many industrial applications which can be said to be “protocol-limited”. Writing VHDL for such systems has always been a difficult process owing to there being no mechanism within VHDL to readily reason or describe the behaviours that we have illustrated. Formal techniques have allowed us to bridge that gap by starting at a higher conceptual level and using design reasoning, at a level where the engineer is good at it, prior to generating RTL code for further synthesis.

References

- [1] M. Bombana, P. Cavallaro, S. Conigliaro, R.B. Hughes, G. Musgrave, and G. Zaza. Design-flow and synthesis for asics: a case study. In *Proceedings of 32nd ACM/IEEE Design Automation Conference*, pages 292–297, San Francisco, California, June 1995. IEEE.
- [2] R.B. Hughes and G. Musgrave. Design-Flow Graph Partitioning for Formal Hardware/Software Codesign. In J.W. Rozenblit and K. Buchenrieder, editors, *Codesign: Computer-Aided Software/Hardware Engineering*, chapter 10. IEEE Computer Society Press, September 1994.
- [3] G. Musgrave, S. Finn, M. Francis, R. Harris, and R.B. Hughes. Formal Methods and Their Future. In *Proceedings of EUROCAST'93 - the third International Workshop on Computer Aided Systems Theory and Technology*, Las Palmas, Canary Islands, February 1993. Springer-Verlag.
- [4] M. Gordon. Why Higher-Order Logic is a good language for specifying and verifying hardware. In G. Milne and P.A. Subrahmanyam, editors, *Formal Aspects of VLSI Design*. North-Holland, 1986.
- [5] H. Busch and G. Venzl. Proof-aided design of verified hardware. In *Proceedings of 28th ACM/IEEE Design Automation Conference*, San Francisco, California, June 1991. IEEE.
- [6] S. Finn, M. Fourman, and G. Musgrave. Interactive synthesis in higher order logic. In *Proceedings of the 1991 International Workshop on the HOL Theorem Prover and its Applications*, Davis, California, August 1991.
- [7] Gordon and Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.