High-Level Hardware Specifications
Keynote Talk

DJ Greaves
Tenison EDA, University of Cambridge*

Abstract

Programmers are good at writing single-threaded functional and imperative code. Everybody is good at writing and understanding rules. Why do hardware engineers persist in writing massively parallel code using little more than RTL?

I argue that mainstream automatic generation of hardware from software designs and formal specifications is inevitable, but we ask why does the road seem so hard ?

1 Design Style Review

Programmers are good at writing single-threaded functional and imperative code. Everybody is good at writing and understanding rules. It is well known that concurrent systems are hard to think about, so why do hardware engineers persist in writing massively parallel code using little more than RTL?

The current holy grail for hardware and SoC development is a tool flow to allow stepwise refinement from very-high level, formal or behavioural models. These high-level models may accurately reflect the contents of RAM memories and 'user' registers in terms of the interactions between hardware and software, or possibly be higher still, taking us into the zone known as codesign.

Codesign of embedded software and custom application-specific instruction set processors (ASIPs) is today a reality for certain tasks that were previously handled with standard DSP processors, but this methodology has far from permeated the mainstream of SoC design.

One thing that hardware designers need from their design entry language is very-fine control down to the net and gate level of synthesis. But they need this only rarely. The success of Verilog and VHDL is that this level of coding is supported, as well as higher levels, including mainstream RTL.

*djg@cl.cam.ac.uk

Designers commonly use the behavioural elaboration components of Verilog and VHDL, where an imperative program (using blocking assigns (Verilog) or variable assigns (VHDL)) describes their intent. But the normal synthesisable subset limits them to updating each variable with eactly one thread, and introducing each thread manually to generate a small number of gates (say 500 typical max). Finally, each thread is enabled once per clock cycle and the tool chain gives zero support for synchronised data transfer between threads. The only non-RTL feature normally available is pausing in the middle of the process loop and waiting for a clock edge. Handle-C from Celoxica is more powerful in some ways, because it has blocking interprocess communication along channels [8], but again the exact number of clock cycles used and overall timing of the system are simply syntax driven, with hardly any scheduling being done by the tool. In my own department, SAFL [10] is broadly similar, but uses a subset of ML as the hardware description language.

A mainstream commercial venture from Synopsys was the Behavioural Compiler product. Here, the much-loved synthesis semantics of VHDL and Verilog are slightly **changed** to provide new functions. A neat and very desireable feature was the free-floating port groups, where the compiler could insert its own pipeline delays, provided it preserved the relative timing of control signals within a nominated set of signals that make up the port. Of course, overloading a much-loved language to give parts of it a new meaning is hard to sell idea. It's why people have referred to C++ a write-only language. To be fair, the tool also greatly expanded the synthesisable subset beyond what was supported before, but here I think they ran into a problem encountered by other high-level synthesis providers: uncertain expectations. The users can no longer be sure what the tool will generate or how to debug it. The CEO of one company offering a C to Hardware tool complained to me that his tool always generated a chip the size of a baseball field. Nonetheless, these objections echo the remarks of the neanderthal 1960's programmers who were frightened of abandoning predictable and comfortable assembly programming and moving to FORTRAN.

A number of C to Verilog compilers do exist. I have written one. This will accept basically all of C, except for floats and pointers that cannot be resoloved to a particular array at compile time. Multiple user threads are supported, but one is not forced to use them everywhere, as in Verilog and VHDL. Instead any variable can be updated any thread and I even got as far as adding a test-and-set for semaphores.

I think all of these compilers operate by performing a symbolic evaluation of the whole program as far as they wish to go, and the writing out the result as a single logical cycle of the system. In my compiler, I stopped symbolic evaluation at user-inserted barrier instructions or where the equality of array subscripts could not be determined at compile time. The logical cycle is then mapped to physical time-space resources such as ALUs and RAMs using various heuristics and user inputs. In my compiler, the user can decide on the number of RAMs and port functions on each RAM and then map each variable or array into a chosen RAM as desired. Without a mapping, the rule was that each array became a single-ported SSRAM and each scalar a number of flip-flops.

Bringing such a product to the EDA market place is not easy, as the experience of a small number of notable start-ups has shown. Indeed, there are both technical and commercial barriers to adoption. With a 90nm mask sets at $1.2M a throw, changing the design flow is always a risk, and one must still make provision for legacy and third party IP. In my view, those who are currently adopting SystemC for architectural exporation will inevitably start using small amounts of hardware synthesis from C in the future.

2 Bridging the Behavioural Design Gap

VTOC is a tool from Tenison EDA that converts an RTL hardware design in Verilog or VHDL to an executable C++ model. It is mainly being used for rapid-prototyping, so that programming teams for SoC are able to test out their software without depending on any hardware: neither a tapeout or other emulator.

The users of VTOC aspire to the behavioural modelling and stepwise refinement methodology, but today most of the conversion from spec to RTL is done by hand using real engineers. However, the use of SystemC for architectural exploration is in trial at a number of companies and certain have used VTOC to generate a SystemC model from the RTL to plug back up into the architecural model for proof of concept or design iteration.

VTOC is cycle-accurate at the register level, which implies it is also accurate at the bus transaction level (BTL) and memory view level (MVL), but it is actually these second two aspects that the programmers want, since they have no knowledge of the internal registers anyway. In the future, VTOC may abandon register-level modelling but preserve BTL and MVL. This is a research topic inside Tenison.

With Alan Mycroft, I have coined the term ‘erosion’ to mean the opposite of refinement. What Tenison is doing is eroding the RTL details away so that the programmers get a completely accurate model that should conform to the design spec if the RTL coders got it right.

3 Declarative Design

Most of the hardware designs I have worked on are used to ship data. Quite frequently, in a buggy RTL design, the last bit of a packet may get corrupted, or perhaps the byte after a frame-alignment boundary may get duplicated. These are data conservation errors. A basic rule for data handling is that the data is preserved by default, regardless of its actual value. The use of such rules leads us to consider a declarative design system.

A lot of real hardware design involves stitching together sub-modules, bought in as external IP. Common bus structures for SoC, such as AHB [7] from ARM, can help, but really we should use a design language where components can be joined to each other with flow control and bus width conversion as primitive, automatic operations. A small standard library of protocol conversions will indeed handle most common situations, including FIFO buffering and crossing clock domains. Note: such a language will only be successful if all existing design methodologies are also easily available (i.e. gate level, RTL and behavioural elaboration). New research on behavioural type systems to describe the terminals of structural components for software systems [6] can be adapted to automatic synthesis of structures for interconnecting ports of various types.

In recent years, a number of papers have presented synthesis from formal methods [1, 3, 4, 2]. Although this seems to be the right track, there is much work to do, including all aspects of integration with existing design flows. For success we need fine-grained mixing.

We might go further and think of a whole gammit of useful features to pile into a system design language. These include advanced abstract data types such as sets, and complex algorithms such as stable-marriage scheduling. After all, most of system design comes down to trading off time and space, by which I mean, generating a schedule of the available resources (space) that runs with the lowest latencey (time).

So I have argued that a small number of appropriate threads with upcalls and so on are much easier to think about (i.e. staff who can deal with them are less costly) than today’s massive user-level parallelism in today’s HDLs. But I further believe that declarative programming with rules is potentially easier (provided all other, prior methodologies are

also seamlessly available!). The good thing about rules is that they do not come in any order and rules from disparate sources are easily aggregated. The resulting system is either unbuildable (rules inconsistent) or generatable as a counter-example from a model checker or other planning engine. I have done a small amount of work on this under the title Orangepath [9]. For instance, I directly used the output from a SAT solver to generate the programming bit-stream for a fictional FPGA. The resulting hardware did the job, but only the inner core of the SAT solver perhaps knew how, why or which bits of the resulting design were totally unneccessary!

Right now, I am experimenting with a system design language (called H2) where the use can describe various facets, such as protocols, interfaces, structural compositions and basic nodes. A basic node might be a flip-flop or ALU, that can only serve one purpose at a time. Facets can be specified in a wide variety of ways, from direct imperative programming, through regular expressions, to macro generation from the results of an integer linear programming. The user brings together a number of such facets and a model checker (or other planning tool) generates a schedule of their use which is compatible with all the pertaining declarative rules.

4 Conclusion

In summary, I believe that automated reasoning (AR) will play a much greater part in tomorrow's system design flow. Today it is limited to theorem proving for (semi-)automatic checking of hand-engineered designs, but while silicon capacity remains cheaper than brainpower there is no reason why systems should not be completely generated by AR planners. Although these techniques apply to codesign decisions at the highest level, there is also a major role for them to play at intermediate levels of abstraction, such as generating schedules.

References

[1] Behavioural Transformation for Algorithmic Level IC Design IEEE Trans CAD Vol 8 No 10, 1998.

[2] DSS: A Distributed High-Level Synthesis System. J Roy, N Kumar, R Duta, R Vemuri. IEEE Design and Test of Computers June 1992.

[3] An Approach to the Synthesis of HW and SW in Codesign. V Carchiolo, M Malgeri, G Mangioni. Proceedings IEEE/IFIP/ACM 5th International Workshop on Hardware/Software Codesign, Braunschweig (Germania), 24-26 March 1997.

[4] Prototyping of VLSI Components from a Formal Specification Roderick McConnell. Inria Report PI-865.

[5] Confluence tutorial and reference manual. Tom Hawkins. http://www.launchbird.com.

[6] Behavioral Types for Component-Based Design. Edward A. Lee and Yuhong Xiong. Memorandum UCB/ERL M02/29. EECS, Berkeley.

[7] AHB bus AMBA Specification. Arm Limited.

[8] Page I. and Luk W. Compiling Occam into Field-Programmable Gate Arrays. Seemingly unpublished, available from ftp://ftp.comlab.ox.ac.uk/pub/Documents/techpapers/Ian.Page/ hw-comp.ps.gz

[9] Greaves, D.J. Generic System Synthesis for Eternal and Ubiquitous Systems. Computer Laboratory SRG Talk. Slides http://www.cl.cam.ac.uk/users/djg/wwwhpr/optalk/obj/index.html

[10] Mycroft, A. and Sharp, R.W. Higher-Level Techniques for Hardware Description and Synthesis. International Journal on Software Tools for Technology Transfer, 2003 (to appear).