Hardware Description Language Based on Message Passing and Implicit Pipelining

Dmitri Boulytchev
St.Petersburg State University
db@tepkom.ru

Oleg Medvedev
St.Petersburg State University
dours@mail.ru, dours@tepkom.ru

Abstract

We present a hardware description language (currently called "HaSCoL") which is based on both reliable and unreliable message passing and implicit pipelining of message handlers. The language consists of a small core and a number of extensions, which cover many features of high level software languages as well as high level hardware description languages (HDLs). These extensions have simple projections into the core language and allow compact and concise description of complex algorithms. The core language in turn can be converted into efficient VHDL. We discuss place-and-route results for some benchmarks implemented both in HaSCoL and VHDL and suggest an optimization which should improve the results significantly and make them close to those for hand-coded VHDL.

1. Introduction

The advances in hardware development technologies made the process of hardware design and implementation more "software-like". For example many algorithms can now be accelerated in orders of magnitude by partial mapping to FPGA (e.g. molecular dynamics simulation [6] or XML filtering [7]). These algorithms can be of various domains and therefore might be tough to implement using conventional hardware development techniques which were historically ad-hoc crafted to utilize rather low-level tools for rather narrow set of problems. In particular, two general problems in the domain of hardware-software codesign — *Design Space Exploration* (DSE) and hardware-software decomposition — can hardly be solved using conventional hardware description language (such as VHDL or Verilog) as the language of primary representation.

The main problem with those languages (let alone their low level) is that they can hardly be operated in a formal, unmanned way. For example, hardware development using VHDL involves a lot of pragmatics, is directed by a number of programming patterns and depends on secret knowledge of implementation details and subtle host platform properties. In the same time the assistance of development tools for exploring the design space by source code transformations with predictable results is extremely desirable.

Another inconvenience originates from the fact that hardware and software are traditionally expressed using *different* languages. Thus the problem of interfacing software and hardware components arises. At present time no drastic solution for this problem is suggested.

We argue that these problems can be solved by describing both hardware and software using the same general-purpose programming language (with hardware description as its proper sublanguage). We call this language "HaSCoL" (**Ha**rdware-**S**oftware **Co**design **L**anguage). This paper presents our current state of the art: a pure hardware description language.

2. HaSCoL in a nutshell

This section informally describes the language by examples.

2.1 Core features

Any specification in HaSCol describes *synchronous* design. Unlike VHDL or Verilog clock and reset (and some auxiliary) signals are not presented explicitly but express themselves by mean of relevant constructs semantics.

The example below demonstrates some features of the language using inline comments (which start with two dashes):

```
— A 19−bit unsigned integer global variable with asynchronous
— read and synchronous write access; initial value 5
— is assigned on each reset.
data count : uint(19) = 5;
— A handler which starts on each cycle when messages arrive
— simultaneously from the channels "chan1" and "chan2".
in chan1(a : Type1), in chan2(b : Type2) {
  — In the first cycle of handler body we put the result of
  -- some computation ("a+b") into intermediate register "c" and
  -- increment the global counter in parallel.
  c = a + b | count := count + 1;
  -- In the second cycle we evaluate a condition ("c > 0")
  -- and perform the first cycle of the "else"-branch or
  -- "then"-branch respectively.
  if c > 0 then
    -- We send the value of register "c" through the channel
    -- "chan3". If no handler is ready to consume the value
    -- the computation is suspended for one cycle.
    send chan3(c)
  else
    -- We asynchronously compute expression "not c" and store
    -- the result in register "d". In parallel we send a message
    -- to the "notify" channel in an unreliable manner (i.e.
    -- the computation continues regardless successful delivery).
    d = not c | inform notify();
    -- We reliably send "d" to the "chan3" on the next cycle.
    send chan3(d)
  fi
}
```

The example above demonstrates the two main features of HaSCoL — stallable pipelining and reliable message delivery. Pipelining means that a handler starts to process a new incoming set of messages on each cycle unless its first cycle is stalled. That is, if it takes more than one cycle to process a set of messages then many sets of messages can potentially be processed simultaneously in a pipelined manner.

Reliable message delivery means that each "send" operator may stall a handler if no handler is ready to accept the message being sent. The stall is propagated in opposite to the message flow direction so that no message is ever lost. This reliability mechanism may completely block a handler. Since reliable delivery is not always desirable the language also has "inform" construct, which never stalls but may loose messages.

The language provides many high-level constructs by mean of syntax extensions (see section 3). Below we demonstrate the most hardware specific one — processor instruction description:

```
-- The statement expresses a register-to-register assignment.
-- It matches an instruction code in "cmd" with a binary pattern
-- of the instruction and performs assignment.
 match cmd with
  -- The instruction has two parameters -- numbers of general
  -- purpose registers of the processor.
  insn (src, dst : uint(3))
    -- Instruction semantics.
    does { regs[dst] := regs[src] }
    -- Binary pattern: specified bit strings and binary
    -- representation of the parameters have to be
    -- concatenated to produce a binary representation of
    -- the instruction (e.g. 0x6F000000).
    coded {0b01 src dst 0x000000 }
    -- Assembler syntax pattern: specified strings and
    -- string representation of the parameters have to be
    -- concatenated to produce assembler text for the
    -- instruction (e.g. "r5 := r7").
    looks {"r" dst ":=" "r" src}
end
```

2.2 Structural design and communication

A typical design in HaSCoL consists of a hierarchy of blocks. Blocks are interconnected by channels, which are used to transmit messages between handlers.

A single HaSCoL block is a set of handlers. Each handler receives messages from some channels, processes received data and sends messages in a reliable or non-reliable manner through other channels.

Some of blocks may be written directly in VHDL if one needs to instantiate some technology-specific VHDL components (e.g. a dual-port on-chip RAM).

A situation when two different messages are sent through the same channel in the same cycle is rather common for complex designs. HaSCoL provides a convenient way to resolve such a conflicts by introducing the notion of *ports*, which can be specified for a channel (in fact, if no ports are explicitly specified then one default port is assumed; strictly speaking a message can only be sent through some port of channel, not a channel itself). All ports of the same channel have different priorities assigned. Sending different values through the same port of the same channel at the same cycle results in unpredictable clash of data, while sending through different ports does not.

There are two priority assignment disciplines — *static* and *fair*. Static priorities are assigned once and for all; fair priorities are reassigned in a circular manner each time a message is consumed from the channel.

3. Language extension hierarchy

An important HaSCoL feature, which facilitates various analyses and transformation into VHDL, is that HaSCoL is not a single language but a *hierarchy of languages*. Each hierarchy level is a syntax extension of the previous one. The base level is easily convertible into efficient VHDL. The hierarchy is shown on Fig. 1.

The core language level consists of constructs to express only structural design, message delivery, straight-line multi-cycle code and one-cycle conditionals (no other control flow, no global variables etc.) This level can be directly converted into synthesizable VHDL.

The next level extends core language with global variables. It is converted to the core language by representing each global variable as a handler which holds assigned value by permanently resending it to itself in

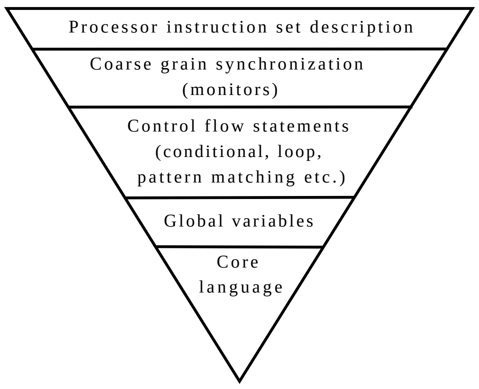
Processor instruction set description
Coarse grain synchronization (monitors)
Control flow statements (conditional, loop, pattern matching etc.)
Global variables
Core language

Figure 1. Language extension hierarchy

Table 1. Evaluation results

	FFT		polynomial	
	VHDL	HaSCoL	VHDL	HaSCoL
Lines of code	531	290*	95	18
4-input LUTs	566	2320	129	162
RAMB16s	40	40	0	0
DSP48s	16	16	18	18
Clock, MHz	147	150	120	122

* The FFT benchmark is written in HaSCoL except for the FPGA-specific block RAM instantiation code, which is in VHDL. The HaSCoL part is 138 lines, the VHDL part is 152 lines

each cycle. Read and write operations are expressed by sending and receiving dedicated messages.

The control flow level adds all conventional structural control flow primitives (including pattern matching) to the previous one. It is translated back to the previous level by emulating control flow with data flow.

Synchronization level enriches the language with the notion of monitors which can be used to prevent conflicting handlers from simultaneous execution. Again, each monitor is expressed in the previous language using similar properties of reliable message delivery.

Finally, instruction set level introduces instruction description construct which we already observed in the Section 2.1. Conversion into the previous level replaces instruction description with regular pattern matching.

A fact that the language is separated into several levels demonstrates that most language constructs are *orthogonal* to each other and to the basic primitives. We believe that this is good by itself.

This fact also allows to implement HaSCoL to VHDL translation as a set of separate passes. This simplifies design and testing of the translator.

4. Evaluation

Currently HaSCoL is supported by the following tools: parser and type checker; VHDL generator which generates synthesizable VHDL; assembler generator which generates a binutils-compatible[1] assembler from a processor instruction set description. In this section we present evaluation of these tools on some benchmarks.

First, we implemented a pipelined cubic polynomial evaluator and a Fast Fourier Transform with a pipelined butterfly. Table 1 shows results of synthesis and place-and-route for a Virtex-4 xc4vlx15-12sf363 device. Implementations written in HaSCoL and in pure VHDL are compared for each benchmark.

For a case of the polynomial the difference in performance is rather minor.

Analysis of mapping results for the FFT explains the major difference in LUT numbers. The reason is an incompleteness of the VHDL generator. It generates a huge amount of logic to support reliable message passing even for the case, where only unreliable one is used (like in FFT). This defect can be fixed with an appropriate code analysis and we will do this soon.

Note that the difference in *description simplicity* is **major** for both cases. The FFT is written almost directly like in a general purpose software programming language, using "`if`" and "`while`" constructs. An ability to integrate VHDL code as "external" processes is used to interface with the Virtex on-chip RAM to store coefficients and data arrays.

Second, we implemented a "real-world-size" design only in HaSCoL. It is a 32-bit general purpose processor. It is integrated into GRLIB[2] environment as an AMBA bus master and is developed to run an embedded Linux distribution. We implemented it using automatical assembler generation, which significantly speeded up the development. The processor occupies 700 lines of code, which include a register file, an ALU, an AHB bus master interface, interrupts and exceptions support, an instruction queue, and a memory management unit. A debug support unit occupies another 155 lines.

The processor has a non-trivial architecture with out-of-order execution. Currently it runs on 70MHz in a GRLIB environment with a DDR2 memory controller on a xc5vlx50t-2ff1136 device. The processor with a debug support unit occupies about 4000 LUTs, 2000 flip-flops, and 4 DSP48 blocks.

5. Related work

Many efforts have been applied to develop a language for high-level algorithm description which could also allow efficient hardware synthesis.

In ROCCC 2.0 [8] project a C-like language is used for algorithm description which allows parallelization

[1] http://www.gnu.org/software/binutils/

[2] http://www.gaisler.com

of loop nests for a higher performance. The parallel versions of loops are then mapped to an FPGA. This project is aimed at high-performance computing, not general hardware development. For example, the language doesn't have a notion of a cycle, so one just cannot express arbitrary hardware with this language.

Our approach is rather different: we want to have a true high-level hardware description language, which could be used as a target for a compiler from an abstract algorithm description language. This would seriously simplify such a compiler.

SpecC [5] [3] is an extension of C programming language for hardware development. Communication between concurrent entities is based on events. The time interval semantics of the event delivery resembles that of VHDL. Statements with arbitrary control flow and explicit parallel and pipeline blocks are allowed.

On the contrary the semantics of our language divides time into cycles. This allows to express *reliable message delivery* as well as static and fair priorities for a channel trough which several messages may be sent simultaneously (i.e. in the same cycle).

Handel-C [2] (which is based on Occam [9] which is in turn based on CSP [3]) is another extension of C. It is aimed mainly at producing synchronous designs and its semantics of channels and reliable message passing resembles ours.

Nevertheless our language have then following differences:

1. A message can be received and re-sent to another channel within the same cycle. This is really desirable sometimes — for example, when we want to write an asynchronous stub between two predefined components.
2. Many handlers may try to read from the same channel in the same cycle.
3. A handler may refuse to consume a message.

A structural decomposition support in Handel-C also seems not to be as convenient as in HaSCoL or VHDL.

Bluespec [4; 1] allows a system to be described as a set of modules communicating by sending commands to one another and by accessing read-only ports of each other. An internal description of a module is based on a concept of guarded atomic actions. These actions operate on the module's local data. Each action has a predicate that says whether it can be fired on this cycle. A programmer may assume that actions are executed atomically and sequentially. In order to execute some ations in parallel to improve efficiency while preserving semantics a special dynamic scheduler of them is generated.

[3] http://www.cecs.uci.edu/˜specc/

Our language doesn't hide that much from a designer and allows one to make cycle-accurate descriptions.

SystemC [10] allows hardware to be described at various levels of abstraction. It is intended mostly for system level simulation, though there is a possibility to describe hardware at the RTL level. SystemC is not a language but rather a set of idioms for the C++ programming language. We believe that this "programming in terms of idioms" approach makes introduction of some desirable language extensions (e.g. control flow) almost impossible.

An important feature of HaSCoL that lacks in all the above-mentioned languages is a convenient pipeline description with automatic creation of intermediate buffers.

6. Future Work

The main direction for a future work is to improve VHDL generator and make it use advanced analysis and optimization techniques to provide more efficient hardware implementations. This would allow us to generate competitive designs.

Another direction is to develop a number of refactoring and transformation tools to provide a way for human-driven automated hardware optimization.

References

[1] Arvind, R. S. Nikhil, D. Rosenband, and N. Dave. High-Level Synthesis: An Essential Ingredient for Designing Complex ASICs. In *Proceedings of ICCAD*, 2004.

[2] Celoxica. *Handel-C Language Reference Manual.* 2005.

[3] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, 1985.

[4] J. Hoe. *Operation-Centric Hardware Description and Synthesis.* PhD thesis, Dept. of EE&CS, MIT, 2000.

[5] R. D. Jianwen Zhu and D. D. Gajski. Syntax and Semantics of the SpecC Language. In *Proceedings of the Synthesis and System Integration of Mixed Technologies 1997*, 1997.

[6] Y. G. Martin. High Performance Molecular Dynamics Simulations with FPGA Coprocessors. RSSI, 2009.

[7] A. Mitra, M. R. Vieira, P. Bakalov, V. J. Tsotras, and W. A. Najjar. Boosting XML filtering through a scalable FPGA-based architecture. In *CIDR*. www.crdrdb.org, 2009.

[8] W. Najjar and J. Villareal. Reconfigurable Computing in the New Age of Parallelism. SAMOS Workshop, 2009.

[9] I. Page and W. Luk. Compiling Occam into FPGAs. *"FPGAs", Abingdon EE&CS books*, pages 271–283, 1991.

[10] G. M. Thorsten Grötker, Stan Liao and S. Swan. *System Design with SystemC.* Kluwer Academic Publishers, 2002.