Seventh International Workshop on **Designing Correct Circuits 2008**

Budapest, Hungary
29–30 March 2008

A Satellite Event of the ETAPS 2008 group of conferences

Participants' Proceedings

Edited by Gordon J. Pace and Satnam Singh

Preface

This is the seventh edition of the Designing Correct Circuits workshop. Since 2002, after a number of DCC workshops in the 90s, the workshop has been held in conjunction with ETAPS on a biannual basis.

Formal methods have, in recent years, been increasingly present in the design and analysis of circuits. Methods and techniques have been successfully developed, both in academia and industry, to deal with large-scale circuits. And yet, each new result raises new challenges. The DCC informal workshops have brought together researchers in formal methods for hardware design and verification from academia and industry to instigate discussions on new results and challenges about how more effective verification methods can be developed.

The increasing size and complexity of circuits brings in new challenges for more abstract design approaches and more scalable verification techniques. The abstracts and papers collected in these proceedings range over a wide range of current challenges – abstract design approaches, modelling techniques, verification, and new challenges in the design of correct circuits. We look forward to the two days of presentations and discussions about these issues in the informal and relaxed atmosphere of the DCC workshop.

We wish to thank the members of the Programme Committee for their work in selecting the presentations, and to all the speakers and participants for their contributions to the workshop.

Gordon J. Pace and Satnam Singh
February 2008

Programme Committee

Koen Claessen, Chalmers University, Sweden
Mike Gordon, University of Cambridge
Warren Hunt, The University of Texas at Austin
Samin Ishtiaq, ARM
Andy Martin, IBM
Tom Melham, University of Oxford
John O'Leary, Intel
Gordon J. Pace, University of Malta
Tim Sheard, Portland State University
Mary Sheeran, Chalmers University, Sweden
Satnam Singh, Microsoft Research
Walid Taha, Rice University

Table of Contents

Flexible Hardware Design at Low Levels of Abstraction 1
Emil Axelsson (Chalmers University of Technology)

Design and Verification of On-Chip Communication Protocols 15
Peter Böhm and Tom Melham (Oxford University Computing Laboratory)

Describing Hardware with Parallel Programs 30
David Greaves and Satnam Singh (University of Cambridge and Microsoft Research Cambridge)

Efficient Circuit Design with uDL 41
Steve Hoover (Intel)

Some “Real World” Problems in the Analog and Mixed Signal Domains 51
Kevin D Jones, Jaeha Kim, Victor Konrad (Rambus)

Defining Elastic Circuits with Negative Delays 70
Sava Krstić, Jordi Cortadella, Mike Kishinevsky, and John O'Leary (Intel and Universitat Politècnica de Catalunya)

Bridging the Gap between Abstract RTL and Bit-Level Designs 72
Andrew K. Martin (IBM)

Model Checking Transactional Memory 73
John O'Leary, Bratin Saha and Mark R. Tuttle (Intel)

Access to Circuit Generators in Embedded HDLs 74
Gordon J. Pace and Christian Tabone (University of Malta)

Wire-Wise Correctness for Handel-C Synthesis in HOL 86
Juan Ignacio Perna and Jim Woodcock (University of York)

Obsidian: GPU Programming in Haskell 101
Koen Claessen, Mary Sheeran and Joel Svensson (Chalmers University of Technology)

Parametric Verification of Industrial Cache Protocols 117
Murali Talupur, Sava Krstić, John O'Leary and Mark R. Tuttle (Intel)

Flexible Hardware Design at Low Levels of Abstraction

Emil Axelsson (emax@cs.chalmers.se)
Chalmers University of Technology

Experiments with FPGAs have shown that automatic placement can give highly sub-optimal results [1], and that manual placement sometimes gives much better performance [2]. This is particularly the case for regular circuits for which the logical structure also makes a good physical structure. Furthermore, the continued existence of full-custom microprocessor designs [3] shows that this limitation of automatic tools is not just FPGA specific.

The downside of manual placement is that it is time consuming and makes a design harder to maintain and reuse, so naturally there are many methods that try to alleviate this kind of design; from the use of specific attributes in the VHDL source [4], to low-level scripting code for interfacing with CAD tools. Placement support in hardware description languages typically comes in one of the following forms

1. Annotations added to existing structural code [4]
2. Placement combinators with simultaneous structural and geometrical meaning [2]

Method (1) has a loose connection between structure and geometry. This has the advantage of allowing the code to be refined, starting from the purely structural version and adding sufficient geometrical constraints. But it also seems a bit ad-hoc since, for example, there is no guaranteed consistency between the annotations and the netlist. The language Pebble [5] seems to restore this consistency by integrating the placement directives directly into the syntax.

Method (2), used for example in the hardware description language Lava, gives a very clear, consistent connection between structure and geometry, but this also means that geometrical refinement is more intrusive on the code. Furthermore, the combinators have a predefined structural meaning and do, for example, not allow the naming of intermediate signals. This means that an unnatural form of description is sometimes needed in order to fit a combinator's interface.

In this work, we develop a system for expressing placement in a Lava-like language. Just like Lava, our system is implemented in the functional programming language Haskell. The system, called Wired, can be seen as an extension to Lava, but the placement component is general and can be used to express layout of any type of computation. Haskell has very convenient syntactic support for monads [6], which offer a way to emulate side-effects in a structured way. In Wired we use monads to model placement as a side-effect of the structural code and optional placement directives to control the relative placement of blocks. Moreover, thanks to the functional nature of the host language, we get higher-order (placement) combinators for free. This means that Wired combines the advantages of the two methods above in a very flexible way.

The main motivation behind Wired is to be able to control not only the circuit's placement, but also effects of the routing wires, which have a dominating impact on performance in modern chip technologies. We do this by supporting “routing guides”, which specify points that a certain signal should be routed through. These guides are conveniently placed in the same way as other circuit blocks. In the future, Wired should also be equipped with static timing analysis based on wire load estimates for quick feedback during design exploration.

The placed netlist (including guides) can be exported to postscript for visualization or Cadence DEF format which allows us to read the circuit into standard CAD tools for routing and further processing. We should soon (definitely before DCC) be able to report on experiments with real designs (adders and multipliers) in our local CAD flow, and this experience will be the main theme of this talk.

This is the first report on connecting Wired to a realistic design flow. In our previously published work on Wired, the system was based purely on method (2) above, and all wire segments were explicitly placed, which made designing very tedious. In addition, it wasn't clear if the use of detailed routing would be feasible when realistic design rules were considered. The new system is much more flexible, and yet, by including enough guides for the routing, we get almost as good control over the routing. And since we only specify guides for the routing, we leave most of the details about design rules to the automatic router.

References

[1] Michail Romesis, Jason Cong, Min Xie. Optimality and stability study of timing-driven placement algorithms. International Conference on Computer-Aided Design, p. 472, 2003.

[2] Satnam Singh. Death of the RLOC? IEEE Symposium on Field-Programmable Custom Computing Machines, p. 145, 2000.

[3] D. Pham, et al. The design and implementation of a first-generation Cell processor. IEEE Journal of Solid-State Circuits, volume 41, issue 1, p. 179-196, Jan 2006.

[4] Evan Lavelle. Using VHDL to create Xilinx RPMs: an introduction. www.riverside-machines.com/pub2/xilinx/vhdl_rpm/place1.htm

[5] Steve McKeever, Wayne Luk, Arran Derbyshire. Towards verifying parametrised hardware libraries with relative placement information. 36th Annual Hawaii International Conference on System Sciences, p. 279c, 2003.

[6] Philip Wadler. Monads for Functional Programming. Proceedings of the 1992 Marktoberdorf International Summer School. 1993.

Flexible hardware design at low levels of abstraction

Emil Axelsson

Chalmers University of Technology

DCC 2008

Acknowledgment

This project receives Intel-custom funding through the Semiconductor Research Corporation

What is low level (in this talk)?

- A technology mapped cell-level description with some awareness of the physical (non-functional) properties of the circuit's implementation, e.g:
  - Physical location of cells
  - Shape of wires

Why low level?

- High-performance designs require low-level optimizations
  - Main reason: Routing wires are dominating performance-stoppers [1]
- Automatic synthesis tools, lacking deep understanding of the design, often give sub-optimal results [2,3]
- Processors from e.g. Intel and IBM are to a large extent designed manually with lots of low-level optimizations [4]

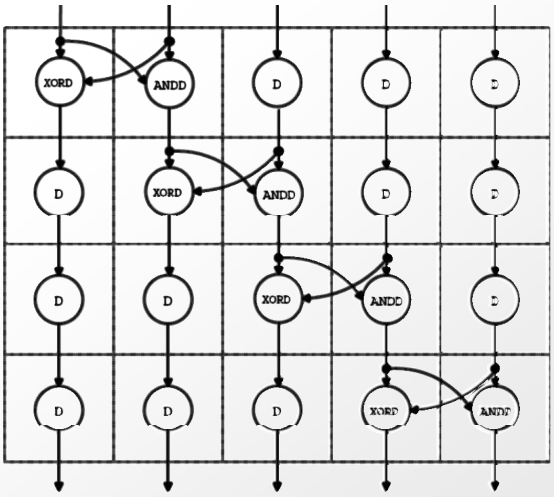
Physical awareness in HDLs

- Simple form of physical awareness: ***Placement***
  - Allows reasonable estimation of area and wire loads/delays
- Two main methods:
  - Relative/absolute placement ***annotations*** added to structural code [5,6]
  - Relative ***placement combinators*** with simultaneous structural and geometrical meaning [7]

Placement annotations (in Pebble [6])

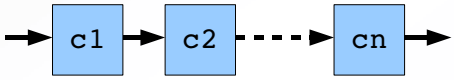
- Pebble: Simple VHDL dialect with placement support
- Example: Pipelined incrementer

```
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)]
  ) ;
```

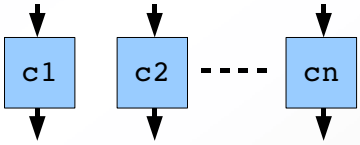


Placement combinators a la Lava

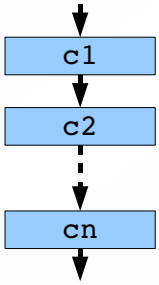
```
serBeside [c1,c2 ... cn]
```



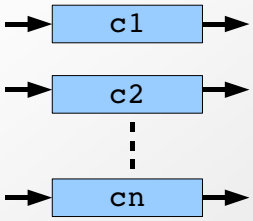

```
parBeside [c1,c2 ... cn]
```




```
serBelow [c1,c2 ... cn]
```




```
parBelow [c1,c2 ... cn]
```



Incrementer with placement combinators

```
increment as = serBelow (map incr1 [0 .. length as-2]) as
  where
    incr1 n as = as'
      where
        dels   = parBeside (repeat del)
        ha     = (xor2D `parBeside2` and2D) . fork
        theRow = dels `parBeside2` ha `parBeside2` dels

        (bs, c1:c2:cs)          = splitAt n as
        ((bs',(c1',c2')),cs')   = theRow ((bs,(c1,c2)),cs)
        as'                     = bs' ++ c1':c2':cs'
```

Interesting part

Plumbing

Comparison

- Annotations
  - Separation between data-flow and geometry
  - Non-intrusive
  - Flexible
- Combinators
  - Powerful, reusable
  - But:
    - Predefined data-flow (may require plumbing)
    - Intrusive

This work: Wired

- An extension to Lava which
  - combines annotation-style and combinator-style placement
  - targets semi-custom VLSI designs (not exclusively)
  - goes further than other HDLs by providing awareness of routing wires
- Wire-awareness:
  - User-provided guiding points (≈ partial routing)
  - Quick feedback from static timing analysis based on wire length estimates

Monads in Haskell [8]

- Haskell functions are pure
- Side-effects can be “simulated” using monads

```
add a = do
    m <- get
    put (m+a)
    return (m+a)
```

Syntactic sugar, expands to a pure program with explicit state passing

```
*Main> print $ evalState (add 1) 10
11
```

Monads can also be used to model e.g. IO, exceptions, non-determinism etc.

Monad combinators

- Haskell has a general and well-understood combinator library for monadic programs

```
mapAdd = mapM add [10,10,10,10]
```

```
*Main> print $ evalState mapAdd 0
[10,20,30,40]
```

```
*Main> print $ evalState ((add >=> add) 1) 10
22
```

Wired descriptions

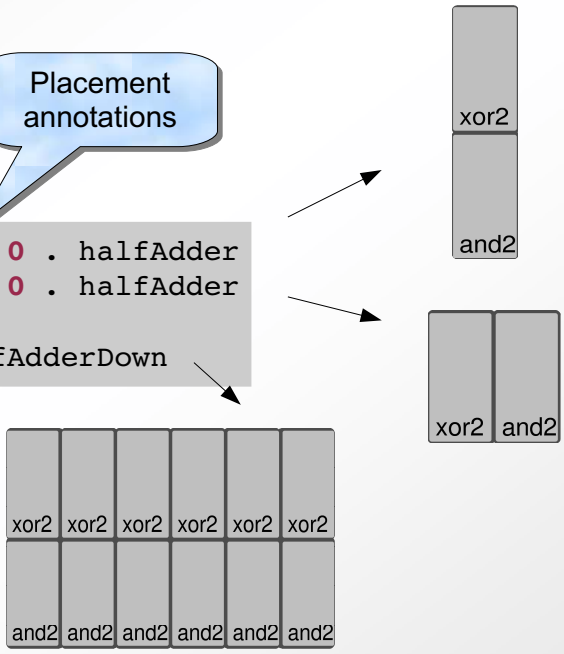
```
halfAdder (a,b) = do
    s  <- xor2 (a,b)
    co <- and2 (a,b)
    return (s,co)
```

Wired descriptions

```
halfAdder (a,b) = do
    s  <- xor2 (a,b)
    co <- and2 (a,b)
    return (a,co)
```

Placement annotations

```
halfAdderDown  = downwards  0 . halfAdder
halfAdderRight = rightwards 0 . halfAdder
mapHalfAdder   =
    rightwards 0 . mapM halfAdderDown
```

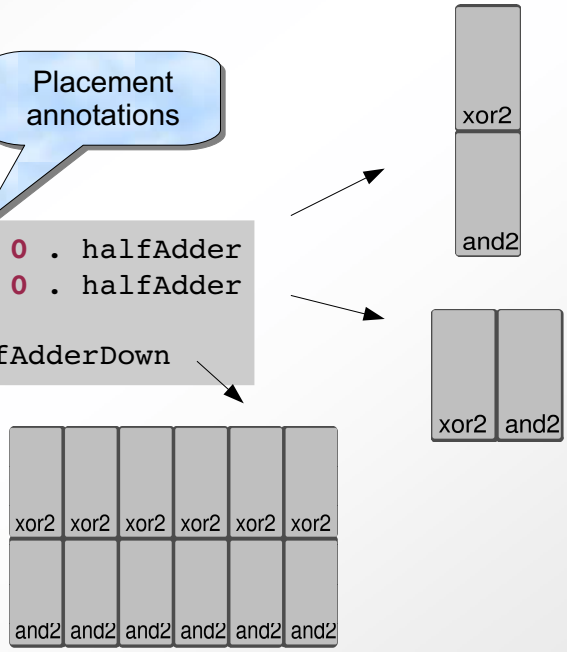


Wired descriptions

```
halfAdder (a,b) = do
    s  <- xor2 (a,b)
    co <- and2 (a,b)
    return (a,co)
```

Placement annotations

```
halfAdderDown  = downwards  0 . halfAdder
halfAdderRight = rightwards 0 . halfAdder
mapHalfAdder   =
    rightwards 0 . mapM halfAdderDown
```



Reusing do notation and monadic combinators

Incrementer in Wired

- Purely structural version:

```
increment as =
    serial (map incr1 [0 .. length as-2]) as
  where
    incr1 n as = do
        let (bs, c1:c2:cs) = splitAt n as
        bs' <- mapM delay bs
        c1' <- xor2D (c1,c2)
        c2' <- and2D (c1,c2)
        cs' <- mapM delay cs
        return (bs' ++ c1':c2':cs')
```

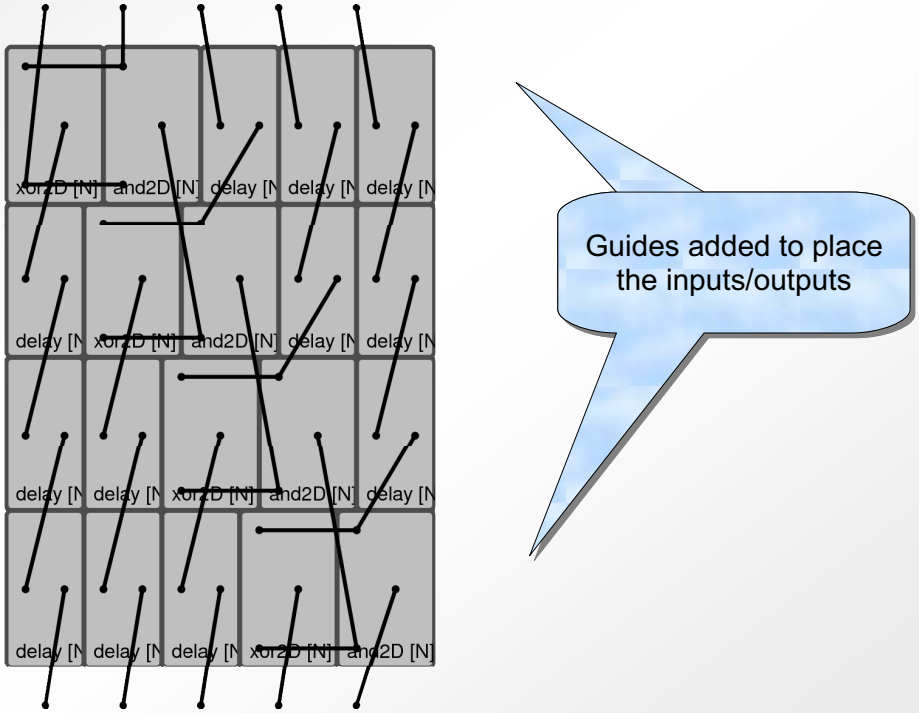
Incrementer in Wired

- Adding placement annotations:

```
increment as = downwards 0 $
    serial (map incr1 [0 .. length as-2]) as
  where
    incr1 n as = rightwards 0 $ do
        let (bs, c1:c2:cs) = splitAt n as
        bs' <- mapM delay bs
        c1' <- xor2D (c1,c2)
        c2' <- and2D (c1,c2)
        cs' <- mapM delay cs
        return (bs' ++ c1':c2':cs')
```

Incrementer in Wired

Showing wires



Search for optimal prefix networks

- Mary Sheeran has developed a successful algorithm for finding optimal prefix networks based on a simplistic delay model
- I will show results from our experiment on using Wired to make this search more accurate
- I will demonstrate the use of routing guides to let the user control parts of the wiring (though we have indications that this may not work well in practice)
- I will also show how the resulting circuit is exported and read in to our local CAD flow (Cadence Soc Encounter) for routing and more accurate timing analysis

Summary

- Flexible low-level design enabled by
  - Combination of placement annotations and combinators
  - Simple form of wire-awareness through routing guides and static timing analysis
- Wired now connects to standard CAD flows
- Much more experiments needed
  - Can Wired handle the many details found in more realistic designs (e.g. sizing, buffering)?
  - Are routing guides really useful other than for visualization?
  - ...

Wired history

- Previously published work on Wired [9]
  - was based purely on placement combinators
  - forced the designer to describe the routing in detail
  - did not permit partial placement/routing
  - was *very* inflexible
  - is not further developed
- The current version has the same motivation as before, but is quite different in nature
- Old vs. new version:
  - Minor difference in accuracy (detailed routing vs. guides)
  - Major difference in flexibility

References

[1] Chappell Brown. SoC Interconnect Crisis: Path Delays Cancel Speed Increase. http://www.eetimes.com/story/OEG20030620S0028. 2003.

[2] Michail Romesis, Jason Cong, Min Xie. Optimality and stability study of timing-driven placement algorithms. *ICCAD' 03: International Conference on Computer-Aided Design*, p. 472, 2003.

[3] Satnam Singh. Death of the RLOC? *IEEE Symposium on Field-Programmable Custom Computing Machines*, p. 145, 2000.

[4] D. Pham, et al. The design and implementation of a first-generation Cell processor. *IEEE Journal of Solid-State Circuits*, volume 41, issue 1, p. 179-196, Jan 2006.

[5] Evan Lavelle. Using VHDL to create Xilinx RPMs: an introduction. http://www.riverside-machines.com/pub2/xilinx/vhdl_rpm/place1.htm

[6] Steve McKeever, Wayne Luk, Arran Derbyshire. Towards verifying parametrised hardware libraries with relative placement information. *36th Annual Hawaii International Conference on System Sciences*, p. 279c, 2003.

[7] Koen Claessen, Mary Sheeran, Satnam Singh. The Design and Verification of a Sorter Core. *CHARME' 01: Correct Hardware Design and Verification Methods*, volume 2144, p. 355-369. 2001.

[8] Philip Wadler. Monads for Functional Programming. *Proceedings of the 1992 Marktoberdorf International Summer School*. 1993.

[9] Emil Axelsson, Koen Claessen, Mary Sheeran. Wired: Wire-Aware Circuit Design. *CHARME' 05: Correct Hardware Design and Verification Methods*, volume 3725, p. 5-19. 2005.

Design and Verification of On-Chip Communication Protocols

Peter Böhm
Oxford University Computing Laboratory,
Wolfson Building, Oxford, OX1 3QD, England
Email: peter.boehm@comlab.ox.ac.uk

Tom Melham
Oxford University Computing Laboratory,
Wolfson Building, Oxford, OX1 3QD, England
Email: tom.melham@comlab.ox.ac.uk

***Abstract*—Modern computer systems rely more and more on on-chip communication protocols to exchange data. To tackle performance requirements these protocols have become highly complex, which makes their formal verification usually infeasible with reasonable time and effort.**

We present an initial case study for a new approach towards the design and verification of on-chip communication protocols. This new methodology combines the design and verification processes together, interleaving them in a hand-in-hand fashion.

In our initial case study we present the design and verification of a simple arbiter-based master-slave communication system inspired by the AMBA High-performance Bus architecture. Starting with a rudimentary, sequential protocol, the design is extended by adding pipelining and burst transfers. Both extensions are realized as transformations of a previous version such that the correctness of the former leverages the verification of latter. Thus, we also argue about the correctness of both extended designs.

I. Introduction

Modern computer systems rely more and more on highly complex on-chip communication protocols to exchange data. The enormous complexity of these protocols results from tackling high-performance requirements. Protocol control can be distributed, and there may be non-atomicity or speculation. Moreover, different system components may have separate clocks or adjustable clock frequencies, requiring asynchronous communications. These complexities arise in many important circuit design areas, such as multicore architectures, system-on-chip, or network-on-chip designs.

Whereas the efforts of chip manufacturers to validate or even formally verify their designs has increased over the last years, the complexity of communication protocols makes their formal verification usually infeasible within reasonable time and effort.

We present a new approach towards the design and formal verification of on-chip communication protocols. The approach can be summarized as follows. We start with a design model for a basic protocol that can be formally verified with reasonable effort; this is then extended with advanced features step-by-step to meet performance demands, handle asynchronous communication, or improve fault-tolerance. These extensions are realized by mathematical transformations of a previous design, rather than constructing a new design. The correctness of an extended design is obtained from the correctness of the previous version and either the transformation itself (*correctness-by-design*) or a refinement or simulation relation between the two versions.

Using this approach, the verification is interleaved with the protocol design process in a hand-in-hand fashion. The task of obtaining the next, more advanced design then splits into three main challenges: (i) is there an algebraic model to derive the extended design from the previous one? (ii) how does the refinement or simulation relation between them look? (iii) how does the correctness statement need to be modified during the design steps?

Verification by stepwise refinement is, of course, not

new (cf. Section I-A). The novelty of our approach lies in the application of this methodology to protocol verification, and in the technical details of the specific optimising transformations we propose to investigate. Given a clean algebraic model for the three parts mentioned above, this leads to a new methodology combining design and verification into a single process. This approach is new, to the best of our knowledge, with respect to high-performance on-chip communication protocol design and verification.

This paper presents an initial case study demonstrating the extension of a simple, rudimentary communication protocol with pipelining and burst transfer features. The basic protocol is a sequential arbiter-bases master-slave communication protocol inspired by the AMBA High-performance Bus (AHB) architecture [1] but reduced to its basics. It supports sequential master-slave communication, including possible slave-side wait-states for memory system operations.

We formally specify a design model realizing the protocol using the Isabelle/HOL [2] theorem prover. Masters and slaves are specified at gate-level as state machines using transition functions. The arbiter is abstracted to a function providing bus grant signals to the masters and obeying a simple fairness property.

The verification is performed using a combination of interactive proofs with Isabelle/HOL and the open-source model checker NuSMV [3]. The NuSMV model checker is used via the oracle-based, domain-reducing interface IHaVeIt [4]. To argue about the behavior over time within the theorem prover, we had to transfer the model-checked temporal logic properties to cycle-accurate statements. This was done using a formalization of an execution trace semantics which relates a hardware cycle to a current state.

Finally, we describe transformations to the design model realizing pipelining and burst transfers of variable but known length. We formulate local and global correctness properties for the new designs and argue about their validity. The correctness is obtained from the correctness of the previous design and reasoning about the applied transformation. Hence, we show transformations that conserve correctness properties from its input design and provide *correctness-by-design* properties.

Even though our technical approach in this case study is modelling in higher order logic, and a combination of theorem proving with Isabelle and model checking with NuSMV, this is not necessarily our plan for future development of this work. We are primarily interested in capturing the right collection of primitive definitions and proof structures to support our planned refinement approach. Therefore, we are deliberately not starting with a pre-conceived notion of what language this future work will happen in, or what tool support we need to provide. Within the overall project, we are focusing on the right basic structuring principles for protocol descriptions with the specific features we are looking at. Choice of language and tools will come after, once we know what we need.

Organization of the paper: Next we discuss related work. In Section II we present the basic overall structure of our communication system and introduce notation used. The design and verification of the basic, sequential design is detailed in Section III. Afterwards, we present the transformation and correctness for pipelining in Section IV and for burst transfers in Section V. Finally, we conclude and outline future work in Section VI.

A. Related Work

Most existing work n this area of formal verification addresses the verification of specific protocols. For example, Roychoudhury *et al.* [5] present a formal specification of the AMBA protocol. They use an academic protocol version and verify design invariants using the SMV model checker. In [6] Amjad verifies latency, arbitration, coherence, and deadlock freedom properties on a simplified AMBA model. Schmaltz *et al.* present initial work [7] of a generic network on chip model as a framework for correct on-chip communication. They identify key constraints on architectures and show protocol correctness provided these constraints are satisfied.

All these approaches rely on a post-hoc protocol verification, which is a key difference to the methodology presented here. Even the framework presented in [7] relies on a post-hoc verification of protocol properties against their constraints. This verification approach becomes more and more infeasible, due to the complexity

of modern communication protocols.

In [8] Müffke presents a framework for the design of communication protocols. It provides a dataflow-based language for protocol specification and decomposition rules for interface generation. These rules relate dataflow algebra and process algebra. Aside from noting that correct and verified protocol design is still an unsolved problem, Müffke does not address the verification aspect in general. He claims that the generated interfaces are correct by construction in the sense that if the generated interface is implemented correctly than the behavior of the components complies with the protocol specification. But he neither addresses the protocol correctness itself nor the verification of the implementation against the specification.

The basic idea behind our approach is similar to Intel's integrated design and verification (IDV) system [9]. The IDV system justifies its transformations by a *local proof* using simple equivalence checking. We expect that focusing on transformations tailored specifically for high-performance on-chip communication protocols result in more intricate refinement steps than can be handled by equivalence checking.

Verification by refinement or simulation relations is also used in many related areas. In [10] Aagaard *et al.* present a framework for microprocessor correctness statements based on simulation relations. Chatterjee *et al.* [11] verify a memory consistency protocol against weak memory models using refinement via model-checking. In [12] Datta *et al.* present a framework to derive verified security protocols using abstraction and refinement steps. They introduce a similar approach to ours but nevertheless neither of them deals with on-chip communication or even complexities arising from high-performance demands.

II. Basics

In this Section we present the overall structure of our communication system and introduce basic notation used throughout the paper.

The presented design is an arbiter-based master-slave system inspired by the AHB architecture. A number of masters (specified by NS) are interconnected with a number of slaves (specified by NS) via a communication bus. Bus access on the master-side is granted by an arbiter. The bus itself consists of a ready signal plus separated control and data parts. Additionally, the masters are interconnected with the arbiter via signals to request and grant the bus.

The masters are connected to the bus using multiplexers controlled by the arbiter. The bus signals generated by the slaves are obtained via large or-trees. The bus is defined in Definition 1.

Definition 1 (Master-Slave Communication Bus) *The master-slave communication bus is defined as the tuple of signals between the master and slaves:*

$$\begin{aligned} bus^t &= (rdy^t, trans^t, wr^t, addr^t, wdata^t, rdata^t) \\ &\in (\mathbb{B}, \mathbb{B}, \mathbb{B}, \mathbb{B}^{ad}, \mathbb{B}^d, \mathbb{B}^d) \end{aligned}$$

where the components are:

- rdy *is the afore mentioned bus ready signal.*
- $trans$ *is the signal indicating either a idle transfer* (0) *or a data transfer* (1)*.*
- wr *denotes if a data transfer is a read* (0) *or write* (1) *transfer.*
- $addr$ *denotes the address bus of width* ad*. The address consists of a local address part (the lower* sl *bits) specifying the memory address within a slave and a device address (the upper* $ad - sl$ *bits) specifying the currently addressed slave.*
- $wdata$ *denotes the write data bus of width* d*.*
- $rdata$ *denotes the read data bus of width* d*.*

We refer to the last two components as data bus *and to the second to fourth components as* control bus*.*

Since the generation of the different bus signals has to be slightly modified during the presented transformations, they are defined in detail within the corresponding sections.

The basic, pivotal protocol characteristics can be summarized as: (i) every transfer consists of an address and a data phase, (ii) the end of each phase is defined by the bus signal rdy, and (iii) the bus is granted to some master at all times.

The first two properties are illustrated in Fig. 1. Every transfer starts with an address phase during which the signals on the control bus need to be generated. The

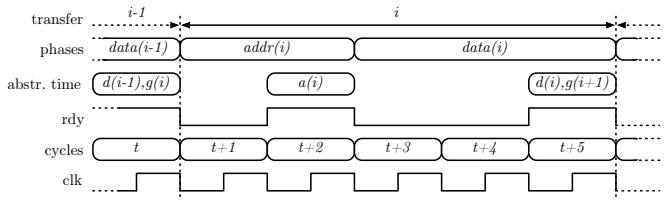


Fig. 1. Sequence of Sequential Transfers

address phase ends with an active rdy signal. At that time, the control bus signals have to be valid.

After the address phase, the data phase starts. It is again completed by an active rdy signal. In case of a write transfer, the master has to provide valid data on the $wdata$ bus during this phase. In case of a read transfer, the slave has to provide valid data on the $rdata$ bus at the end of this phase when the rdy signal is active.

The third characteristic is detailed as follows: the arbiter always grants the bus to some master. In case no master has requested bus access, the bus is granted to a default master $defM$. If a master is granted the bus but there is no data to transmit, the master has to initiate an $idle$ transfer. This transfer is distinguishable from a data transfer and a slave has to acknowledge with a zero wait-state response.

A. Notation

Throughout the next sections we use natural numbers $u \in [0 : NM - 1]$ and $v \in [0 : NS - 1]$ to refer to the u-th master and v-th slave and denote them with $m[u]$ and $s[v]$, respectively. When the number of the master or slave is irrelevant, we omit the u or v and use simply m or s to refer to any of them. We assume that $NM = 2^k$ for $k \in \mathbb{N}$ and $NS = 2^{ad-sl}$.

In order to specify the design they are related to, we index them with either seq for sequential, $pipe$ for pipelined, or $burst$ for burst design. When we refer to an arbitrary design, we omit the index.

We denote a bit vector $x \in \mathbb{B}^n$ with $x[n-1:0]$ and refer to its a-th bit with $x[a]$ where $a \in \{0, \ldots, n-1\}$. For a bit vector $x \in \mathbb{B}^n$ we use the predicate $unary(x)$ to specify that x is a unary coded bit vector, i. e. $x[i] = 1 \iff x[j] = 0$ for all $j \neq i$. Moreover, we denote the value of x interpreted as a unary number by $\langle x \rangle_u$ and define it as $\langle x \rangle_u = a \iff x[a] = 1$. Analogously, we denote the value of a bit vector $x \in \mathbb{B}^n$ interpreted as a binary number by $\langle x \rangle$. It is defined by $\langle x \rangle = \sum_{i=0}^{n-1} x[i] \cdot 2^i$. We denote the binary representation of a unary coded bit vector $x \in \mathbb{B}^n$ with $n = 2^k$ by $bin_u(x) \in \mathbb{B}^k$ and define it as $y = bin_u(x) \iff \langle y \rangle = \langle x \rangle_u$.

Finally, we define a signal sig as a function from clock cycles to Boolean values. Thus, $sig : \mathbb{N} \to \mathbb{B}^n$ for some $n \in \mathbb{N}$. We refer to the value of a signal with sig^t. In order to refer to the value of a signal during a time interval, we use the notation $sig^{[a:b]} = x$ with $x \in \mathbb{B}^n$ as a shorthand for $sig^a = \ldots = sig^b = x$. Similarly, we use $sig^{[a:b]} = sig'^{[a:b]}$ for a signal sig' as a shorthand for $sig^a = sig'^a \wedge \ldots \wedge sig^b = sig'^b$.

III. Sequential Design

In this Section we present a simple, sequential realisation of the protocol without any support of advanced functionality such as pipelining or burst transfers. We start by introducing the concept of an *abstract transfer*. Afterwards, we specify the main parts of the communication system, i. e. arbiter, salve, and master, by an implementation-close description and reason about local correctness. Finally, we define the bus components in the sequential design and argue about global correctness in Section III-D.

The following definition of an abstract transfer relies on the basic protocol property that there is always on master who is granted the bus (cf. Section II). This definition is crucial to the reasoning throughout this paper. The definition is illustrated in Fig. 1.

Definition 2 (Abstract Sequential Transfer) *The i-th abstract sequential transfer $tr_{seq}(i)$ is defined in terms of three cycle-accurate time points, a corresponding grant signal vector $gnt(i) \in \mathbb{B}^{NM}$, and a single bit $id(i) \in \mathbb{B}$ indicating if the transfer is a idle transfer or not. The first time point $g(i) \in \mathbb{N}$ is the point in time when the bus is granted to the master $\langle gnt(i) \rangle_u$. The second time point $a(i) \in \mathbb{N}$ is the point in time when the address phase ends, i. e. when the rdy signal is active the first time after $g(i)$. The third time point $d(i) \in \mathbb{N}$ denotes the time when the data phase of transfer i ends, i. e. when the rdy signal is active for the second time after $g(i)$.*

$$\begin{aligned} tr_{seq}(i) &= (gnt(i), id(i), g(i), a(i), d(i)) \\ &\in \mathbb{B}^{NM} \times \mathbb{B} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \end{aligned}$$

The components are defined as

$$\begin{aligned} gnt(i) &= arb.grant^{g(i)} \\ id(i) &= trans^{a(i)} \\ g(i) &= \begin{cases} 0 & : i = 0 \\ d(i-1) & : \textit{otherwise} \end{cases} \\ a(i) &= \min\{t > g(i) \mid rdy^t\} \\ d(i) &= \min\{t > a(i) \mid rdy^t\} \end{aligned}$$

where $arb.grant$ denotes the arbiter configuration component specifying the current grant vector. It is specified in detail in the next section.

When talking about transfers, we often need to refer to the time point when the host of the granted master requested this transfer. This is done via the *transfer request time* $trt(i)$ of transfer i.

Definition 3 (Transfer Request Time) *Given a transfer $tr(i)$ we define the transfer request time $trt(i)$ as the time point where the master $\langle gnt(i)\rangle_u$ received the corresponding $startreq$ signal. Let $x = \langle gnt(i)\rangle_u$, then $trt(i)$ is defined by:*

$$trt(i) = \begin{cases} \max\{j \leq g(i) \mid m[x].startreq^j\} & : id(i) \\ \infty & : \neg id(i) \end{cases}$$

Note that the second case specifies $trt(i)$ for an idle transfer. This is only done for reasons of completeness and is not used within the proofs as we only refer to $trt(i)$ in case of a data transfer.

A. Arbiter

The arbiter grants bus access to a master $m[u]$ by activating the corresponding bit $grant[u]$ in the grant bit vector. In case no master requests the bus, the arbiter grants the bus to a default master $defM$ as defined in Section II. In the scope of this project, we abstract the arbiter to a combinatorial circuit relying on an abstract function af generating a new grant vector given the current one and the current request vector. Thus, $af(grant, req)$ returns a new grant vector.

The inputs of the arbiter are a request signal from every master, thus a bit vector $req_{in} \in \mathbb{B}^{NM}$, and the rdy signal of the bus. The arbiter updates the $grant$ bit vector at the end of an address phase, i.e. at $a(i)$. In order to store the information whether an active rdy signal represents the end of an address phase or the end of a data phase, we introduce a singe bit register $aphase$.

However, if the $grant$ vector is updated at the end of the address phase, the old vector is still required during the data phase to select the correct $wdata$ output from a transmitting master. Therefore, the old grant vector is stored as a delayed grant vector in the register $dgrant$.

Moreover, a request vector req has to be maintained in order to store which masters requested the bus.

Thus, the configuration of the sequential arbiter is:

$$\begin{aligned} arb^t_{seq} &= (grant^t, dgrant^t, req^t, aphase^t) \\ &\in \mathbb{B}^{NM} \times \mathbb{B}^{NM} \times \mathbb{B}^{NM} \times \mathbb{B} \end{aligned}$$

The arbiter is specified by the following update rules: Let $upd^t = aphase^t \wedge rdy^t$ and $u \in [0 : NM-1]$, then:

$$\begin{aligned} grant^{t+1} &= \begin{cases} af(req^t, grant^t) & : upd^t \\ grant^t & : \text{otherwise} \end{cases} \\ dgrant^{t+1} &= \begin{cases} grant^t & : upd^t \\ dgrant^t & : \text{otherwise} \end{cases} \\ req^{t+1}[u] &= \begin{cases} 1 & : req_{in}[u] \\ 0 & : af(req^t, grant^t)[u] \\ & \quad \wedge\ upd^t \\ req^t[u] & : \text{otherwise} \end{cases} \\ aphase^{t+1} &= \begin{cases} \neg aphase^t & : rdy^t \\ aphase^t & : \text{otherwise} \end{cases} \end{aligned}$$

Note that $aphase$ is initialized with 0.

Lemma 1 (Sequential Arbiter Correctness) *Let $u \in [0 : NM-1]$ be a master index. Then, the local arbiter correctness is described by the following:*

$$\begin{aligned} &unary(grant^t) \wedge (grant^t \neq grant^{t+1} \implies rdy^t) \\ &\wedge\ (aphase^t \iff t \in [g(i)+1 : a(i)] \textit{ for some } i) \\ &\wedge\ (req_{in}[m]^t \implies grant^{t'}[m] \textit{ for some } t' > t) \end{aligned}$$

Finally, the arbiter provides the outputs $grant^t$ to the masters, i. e. $grant[u]$ for $u \in [0 : NM-1]$ to master m_u, as well as to the control bus multiplexers. Additionally, it provides $dgrant^t$ to the write data bus multiplexer.

B. Slave

The task of a slave is to perform read or write accesses to an attached memory system mem. Regarding the bus, the slave has the inputs $sel \in \mathbb{B}$ indicating that the slave is currently addressed, the rdy signal, the $wdata$ component, and the control bus. The $addr$ signal is reduced to the local address $addr[sl-1:0] \in \mathbb{B}^{sl}$. the upper part of the address bus, namely $addr[ad-1:sl]$,

is used to generate the select signal sel by an address decoder (cf. Section II).

In case a slave is currently addressed, indicated by an active sel input at the end of the address phase, it has to sample the control bus data in case of a data transfer. Afterwards, the actual memory system access is performed during the data phase. Within that access the memory system can activate a memory busy signal $mem.busy$. The requested data is delivered by the memory system in the cycle when $mem.busy$ is inactive for the first time after the start of the request. We assume that the memory is busy for at most k cycles.

At the end of the memory request, the slave activates the rdy_{out} output and provides the read data on the $rdata_{out}$ output in case of a read access.

As we will see during pipelining in Section IV, the sequential slave is a little bit more complex than the pipelined one. This results from the fact that the sequential slave has to generate the rdy signal indicating the end of the address phase, $a(i)$, additionally to the rdy signal indicating $d(i)$. As the address data can be sampled during one cycle, a unit delay register rdy' is used to delay an active rdy single by one cycle. Then, if rdy' and sel is active, the rdy_{out} output is enabled to generate the $bus.rdy$ signal at $a(i)$.

Moreover, in case of an $idle$ transmission, the slave just produces a rdy_{out} signal in the next cycle (at $d(i)$).

The slave configuration is defined as the tuple

$$s^t_{seq} = (state^t, wr^t, addr^t, wdata^t, mem^t)$$

where

- $state \in \{idle, req\}$ denotes the automaton state.
- $wr \in \mathbb{B}, addr \in \mathbb{B}^{sl}, wdata \in \mathbb{B}^d$ denote the registers to sample the corresponding bus signals.
- $mem : \mathbb{B}^{sl} \to \mathbb{B}^d$ denotes the local memory.

The slave is realized in a straight forward way according to the above description. We omit details here.

The local correctness statement reads as follows.

Lemma 2 (Sequential Slave Correctness) *Given that $sel \wedge rdy'$ holds at time t on sequential slave s_{seq}. Let $t' = \min(k > t \mid \neg mem.busy^k)$, then:*

$$\begin{aligned} & rdy^t_{out} \wedge (\neg trans^t \implies rdy^{t+1}_{out}) \\ & \wedge (trans^t \implies \\ & \quad \neg rdy^{[i+1:t'-1]}_{out} \wedge rdy^{t'}_{out} \wedge \\ & \quad (\neg wr^t \implies rdata^{t'}_{out} = mem^{t'}[addr^t]) \wedge \\ & \quad (wr^t \implies mem^{t'}[addr^t] = wdata^{t'})) \end{aligned}$$

C. Master

The master provides the interface between the communication system and an attached host system. Within the scope of this case study, the master is our main interest as we present transformations to add advanced functionality to it.

The task of the master is to handle host requests to transfer data. Thus the master has inputs from the host denoted $startreq \in \mathbb{B}$, indicating a transfer request, and host data signals denoted $hwr \in \mathbb{B}$, $haddr \in \mathbb{B}^{ad}$, and $hwdata \in \mathbb{B}^d$ for the respective transfer data. The master has to perform a bus request to the arbiter in case it is not granted the bus. It has to transfer the data according to the sequential schedule, hence the next time it is granted the bus.

Additionally, in case there is no data to transmit but the master is granted the bus, it has to initiate an idle transfer in order to meet the protocol requirements.

The inputs to master $m[u]$ for $u \in [0 : NM - 1]$ regarding the bus are the signals $rdy \in \mathbb{B}$, $grant[u] \in \mathbb{B}$, and $rdata \in \mathbb{B}^d$ as defined in Section II.

As outputs the master provides the signals $trans_{out} \in \mathbb{B}$, $wr_{out} \in \mathbb{B}$, $addr_{out} \in \mathbb{B}^{ad}$, and $wdata_{out} \in \mathbb{B}^d$ needed for the corresponding bus signals. It provides a request signal $req \in \mathbb{B}$ to the arbiter and a busy signal $busy \in \mathbb{B}$ as well as a signal $rdata_{out} \in \mathbb{B}^d$ for the read data to the host. The purpose of the $busy$ signal is the following: the correct transmission of a host request is shown if the master is not busy while the transfer is initiated ($startreq^t \implies \neg busy^t$).

The configuration of the master consists of a $state$ component, a set of registers to sample a host request, and a register to sample data from the $rdata$ bus at the end of a read request. In detail, the configuration of a sequential master m_{seq} is defined as the tuple

$$m^t_{seq} = (state^t, vreq^t, lwr^t, laddr^t, lwdata^t, lrdata^t)$$

where the components are:

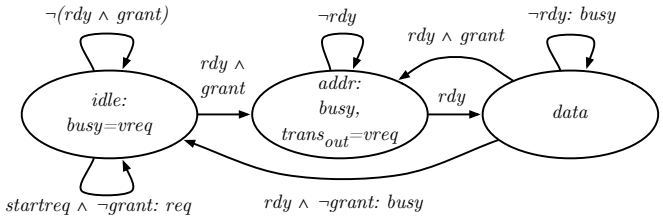


Fig. 2. Sequential Master Control Automaton

- $state \in \{idle, aphase, dphase\}$ denotes the automaton state: $idle$ is the idle state, $aphase$ the state denoting the address phase, and $dphase$ the state denoting the data phase, respectively.
- $vreq \in \mathbb{B}$ denotes that a valid request is currently processed.
- $lwr \in \mathbb{B}$ denotes the local copy of the hwr input. It is written when a host request is sampled.
- $laddr \in \mathbb{B}^{16}$ denotes the local copy of the address analogously to wr.
- $lwdata \in \mathbb{B}^{32}$ denotes the local $hwdata$ copy.
- $lrdata \in \mathbb{B}^{32}$ denotes the register used to sample the $rdata$ bus.

The control automaton is shown in Fig. 2 illustrating the update of the $state$ component and the output signal generation. The bus outputs different from $trans_{out}$ are straight forward the respective local components.

The other configuration components are updated according to the following specification.

$$vreq^{t+1} = \begin{cases} 1 & : (idle^t \wedge startreq^t \wedge \neg busy^t) \\ & \quad \vee (data^t \wedge \ grant^t \wedge rdy^t) \\ 0 & : data^t \wedge rdy^t \\ & \quad \wedge \neg(startreq^t \wedge grant^t) \\ vreq^t & : \text{otherwise} \end{cases}$$

$$lrdata^{t+1} = \begin{cases} rdata^t & : data^t \wedge rdy^t \wedge \neg lwr^t \\ lrdata^t & : \text{otherwise} \end{cases}$$

$$x^{t+1} = \begin{cases} hx^t & : startreq^t \wedge \neg busy^t \\ x^t & : \text{otherwise} \end{cases}$$

where $x \in \{lwr, laddr, lwdata\}$ and $idle^t$, $data^t$ denote that the automaton is in the corresponding state.

Next we argue about the local correctness of the master. We split this argumentation into two parts: (i) local correctness with respect to the host interface, and (ii) local correctness with respect to the bus.

We call a master *locally correct with respect to the host interface* iff it requests the bus upon a valid host request, i. e. a request initiated when the master has not been busy. Therefore, the master shall activate the $busy$ signal during a transfer. Moreover, every time the master is granted the bus, the busy signal has to be inactive for at least on cycle to enable the host to initiate a transfer.

Lemma 3 (Master Correctness wrt. Host Interface) *Given a master m_{seq}, a transfer $tr_{seq}(i)$, and the corresponding transfer request time $trt(i)$. If $startreq^{trt(i)} \wedge \neg busy^{trt(i)}$ holds, then:*

$$\begin{aligned} &(req^{trt(i)} \vee grant^{trt(i)}) \wedge busy^{[trt(i)+1:d(i)-1]} \\ &\wedge (grant^{d(i)} \implies \neg busy^{d(i)}) \\ &\wedge (\neg grant^{d(i)} \implies busy^{d(i)} \wedge \neg busy^{d(i)+1}) \end{aligned}$$

Proof: The proof is obtained by simple reasoning on the control automaton of the master and can be checked automatically. ■

The second correctness part is the more interesting one. We say that the master is *locally correct with respect to the bus* iff the following properties hold: (i) the master reacts to an active rdy and $grant$ signal by starting a transmission; either a $idle$ or a $data$ transmission, (ii) it keeps the bus control signals stable during the address phase, (iii) in case of a write transfer it keeps the write data stable during the data phase, and (iv) in case of a read transfer it samples the $rdata$ bus correctly at the end of the data phase.

Next we define a predicate called $lcorr$ formulating the above properties. Afterwards, we state and prove the correctness lemma for the sequential master.

Definition 4 (Local Master Correctness wrt. Bus) *Given a master m, a transfer $tr(i)$, and a host request $hreq^{trt(i)} = (hwr, haddr, hwdata)$. Then, the local master correctness predicate $lcorr(m, tr(i), hreq^{trt(i)})$ is defined by: $lcorr(m_{seq}, tr(i), hreq^{trt(i)})$ holds iff either $m \neq \langle gnt(i) \rangle_u$ or $m = \langle gnt(i) \rangle_u$ and*

$$\begin{aligned} &trans_{out}^{[g(i)+1:a(i)]} = d(i) \\ &\wedge\ id(i) \implies (wr_{out}^{[g(i)+1:a(i)]} = hwr^{trt(i)} \\ &\qquad \wedge\ addr_{out}^{[g(i)+1:a(i)]} = haddr^{trt(i)})) \\ &\wedge\ (id(i) \wedge hwr^{trt(i)}) \implies \\ &\qquad wdata_{out}^{[a(i)+1:d(i)]} = hwdata^{trt(i)} \\ &\wedge\ (id(i) \wedge \neg hwr^{trt(i)}) \implies lrdata^{d(i)+1} = rdata^{d(i)} \end{aligned}$$

Lemma 4 (Local Master Correctness wrt. Bus) *Given a sequential master m_{seq}, a transfer $tr_{seq}(i)$, and a corresponding host request $hreq_{seq}^{trt(i)}$. Then*

$lcorr(m_{seq}, tr_{seq(i)}, hreaq_{seq}^{trt(i)})$ *holds.*

Proof: The proof is obtained in two steps: (i) by reasoning about the control automaton of the master, a similar statement is proven which argues about the register contents at time $g(i)$ instead of the host inputs at time $trt(i)$, (ii) afterwards, the claim is obtained with Lemma 3. ■

D. Global Correctness

In this Section we argue about the global correctness of the sequential communication system. First, we define the values of the bus components in detail. Afterwards, we argue about the bus signal generation and introduce some global invariants which ease the argumentation about global correctness. Finally, we state the global correctness theorem of the sequential design and argue about its validity.

Definition 5 (Sequential Bus Components) *Let* $u = \langle arb.grant^t \rangle_u$ *and* $du = \langle arb.dgrant \rangle_u$. *Then the components of the bus from Definition 1 in the sequential design are defined by:*

$$\begin{aligned} rdy^t &= \bigvee_{v=0}^{NS-1} s[v].rdy_{out}^t \\ trans^t &= m[u]_{seq}.trans_{out}^t \\ wr^t &= m[u]_{seq}.wr_{out}^t \\ addr^t &= m[u]_{seq}.addr_{out}^t \\ wdata^t &= m[du].wdata_{out}^t \\ rdata^t &= \bigvee_{v=0}^{NS-1} s[v].rdata_{out}^t \end{aligned}$$

As mentioned in Section II, the control bus signals are obtained from the master currently granted the bus and selected by $bin_u(grant)$ using multiplexers. Analogously, the $wdata$ bus is obtained from the master outputs by a multiplexer controlled by $bin_u(dgrant)$.

In the following, we introduce global properties and invariants for the sequential communication system. The first invariant argues about the set of masters. It states that there is exactly one master that is not idle during a transfer. It is exactly the master who is granted the bus.

Invariant 1 (Master Uniqueness) *Given a transfer* $tr_{seq}(i)$. *Then:*

$$\begin{aligned} \exists! u \in [0 : NM-1]. \quad & m[u].state^{[g(i)+1:d(i)]} \neq idle \wedge \\ & m = \langle gnt(i) \rangle_u \end{aligned}$$

Proof: The validity of this invariant follows from the following three properties: (i) the $unary(grant)$ property of the arbiter, (ii) the fact that the $grant$ signal only changes after the active rdy signal at the end of the address phase, and (iii) the fact that a master only leaves the $idle$ state if the $grant$ and rdy signals are active. ■

Given the master uniqueness we show that the bus signals generated by the masters are obtained correctly.

Lemma 5 (Master to Bus Correctness) *Given a sequential transfer* $tr_{seq}(i)$ *and let* $u = \langle gnt(i) \rangle_u$. *Then:*

$$\begin{aligned} trans^{[g(i)+1:a(i)]} &= m[u].trans_{out}^{a(i)} \wedge \\ wr^{[g(i)+1:a(i)]} &= m[u].wr_{out}^{a(i)} \wedge \\ addr^{[g(i)+1:a(i)]} &= m[u].addr_{out}^{a(i)} \wedge \\ wdata^{[a(i)+1:d(i)]} &= m[u].wdata_{out}^{d(i)} \end{aligned}$$

Proof: The lemma is obtained from Invariant 1, the local master correctness from Lemma 4, and the arbiter correctness from Lemma 1. ■

Next, we formulate an invariant similar to Invariant 1 but for the set of slaves. It states that all slaves are in $idle$ state during the address phase and there is exactly one slave that is not in the $idle$ state during the data phase. This is exactly the slave addressed by the master.

Invariant 2 (Slave Uniqueness) *Given a sequential transfer* $tr_{seq}(i)$. *Let* $u = \langle gnt(i) \rangle_u$ *be the granted master. Then:*

$$\begin{aligned} & \forall v \in [0 : NS-1]. \quad s[v].state^{[g(i)+1:a(i)]} = idle \\ \wedge\ & \exists! v \in [0 : NS-1]. \quad s[v].state^{[a(i)+1:d(i)]} \neq idle \wedge \\ & \qquad\qquad v = \langle m[u].addr[ad-1 : sl] \rangle \end{aligned}$$

Proof: The invariant is obtained in three steps: (i) the address decoder correctness ensures that $unary(sel)$ holds and $\langle sel \rangle_u = addr[ad-1 : sl]$, (ii) the master correctness (Lemma 4) and Lemma 5 entail that the address bus is stable during the whole address phase, and (iii) reasoning on the slave control automaton shows that the only time interval the idle state is left is between the rdy signal at time $a(i)$ and the next one ($d(i)$). ■

A property similar to Lemma 5 regarding the rdy and $rdata$ signals of the bus can be formulated. It is obtained from Invariant 2 and the slave correctness in Lemma 2.

Lemma 6 (Slaves to Bus Correctness) *Given a transfer* $tr_{seq}(i)$ *and let* $v = \langle addr^{a(i)}[ad-1 : sl] \rangle$ *be the addressed slave. Then:*

$$\begin{aligned} rdy^{[a(i):d(i)]} &= s[v].rdy_{out}^{[a(i):d(i)]} \wedge \\ rdata^{d(i)} &= s[v].rdata_{out}^{d(i)} \end{aligned}$$

Finally, the overall correctness theorem reads as follows.

Lemma 7 (Overall Correctness Sequential System) *Let $hreq_m^k$ denote a host request at time k with $startreq^k$ on master m_{seq}. Moreover let*

$$w = s[\langle haddr^k[ad-1:sl]\rangle].mem(haddr^k[sl-1:0])$$

denote the addressed memory word. Then:

$$\begin{aligned} \neg busy_m^k \implies & \exists i.\ (k = trt(i) \\ & \wedge\ tr(i) = (gnt(i), id(i), g(i), a(i), d(i)) \\ & \wedge\ gnt(i) = m_{seq} \wedge id(i) \\ & \wedge\ (hwr^k \implies w^{d(i)} = hwdata^k) \\ & \wedge\ (\neg hwr^k \implies lrdata^{d(i)+1} = w^{d(i)})) \end{aligned}$$

Proof: The first two clauses follow from the arbiter correctness in Lemma 1 and the correctness of the master with respect to the host interface (Lemma 3). The third follows from the transfer definition in Definition 2 and again the arbiter correctness. The last three clauses are given by the master to bus correctness in Lemma 5, the slave to bus correctness in Lemma 6, and the local correctness of both (Lemmas 4 and 2). ■

Note that one would need a much more evaluated arbitration system in order to provide the read data from time k instead of time $d(i)$.

IV. Pipelined Protocol

In this section, we derive a communication system supporting pipelined data transfer from the previous sequential design based on a correctness-preserving transformation. The idea behind the pipelining of this protocol is to execute the address phase of a transfer i in parallel with the data phase of address $i-1$. This is possible because of the separated control and data buses. A sequence of pipelined transfers is shown in Fig. 3.

The bus for the pipelined system stays exactly the same as for the sequential system.

$$bus^i = (rdy^i, trans^i, wr^i, addr^i, wdata^i, rdata^i)$$

As we focus on the transformation applied to the master, we do not go into details regarding arbiter and slaves here. The arbiter is obtained by ignoring the $aphase$ bit register from the sequential arbiter and updating the $grant$ vector each time the rdy signal is active. Since the $grant$ vector is updated every time rdy is active, we have to introduce a third grant register denoted $ddgrant$ which is updated with the data from $dgrant$ in exactly the same way as $dgrant$ with $grant$.

This results from the following: The master for the next transfer is specified by the $grant$ vector at time $g(i)$. In contrast to the sequential arbiter, the grant vector is also updated at time $g(i)$. Thus the control parts of the bus during the address phase are not specified by the $grant$ but by the $dgrant$ vector. But then, we need even one more copy of the former grant vector to control the data multiplexer during the data phase.

The slave is obtained by removing the unit delay rdy'. The pipelined slave only generates a rdy signal after the memory access, thus at the end of the data phase.

Finally, we adopt the abstract transfer definition.

Definition 6 (Abstract Pipelined Transfer) *The i-th abstract pipelined transfer is defined analogous to the sequential transfer from Definition 2 as the tuple $tr_{pipe}(i) = (gnt(i), id(i), g(i), a(i), d(i))$ where the components are defined as:*

$$\begin{aligned} gnt(i) &= grant^{g(i)} \\ id(i) &= \langle gnt(i)\rangle_u.trans_{out}^{a(i)} \\ g(i) &= \begin{cases} 0 & : i = 0 \\ a(i-1) & : otherwise \end{cases} \\ a(i) &= \min\{t > g(i) \mid rdy^t\} \\ d(i) &= \min\{t > a(i) \mid rdy^t\} \end{aligned}$$

The new definition is illustrated in Fig. 3. Note that the only difference to the sequential definition is that $g(i)$ is now recursively defined over $a(i-1)$ instead if $d(i-1)$. This represents the pipelining in the abstract transfer and results in the additional properties $a(i) = d(i-1)$ for $i > 0$ and $g(i) = d(i-2)$ for $i > 1$.

Next we specify the transformation for the master followed by the local correctness argumentation. Finally, we reason again about the global correctness of the pipelined communication system.

A. Master

Our gaol is to obtain a master supporting pipelined transfers from the master only supporting sequential transfers. We derive the new master in an algebraic way such that we can use the correctness of the former

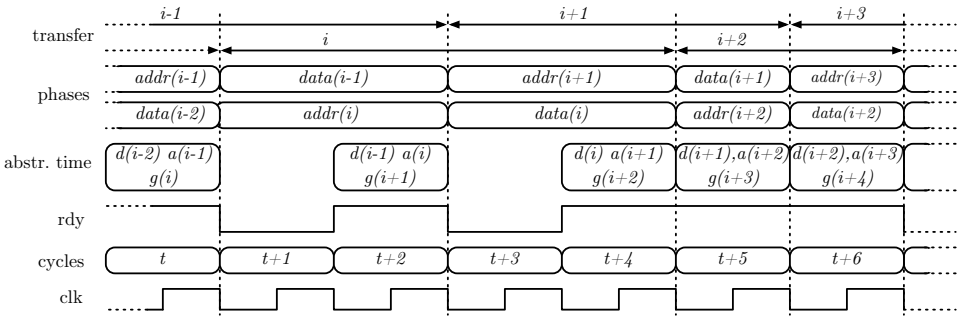


Fig. 3. Sequence of Pipelined Transfers

master to argue about the new one. The basic idea behind the transformation is to execute two sequential masters in parallel. Then we restrict the behavior of the parallel system by excluding some *bad* executions. In the following, we denote the two sequential masters with $m1$ and $m2$, hence $m_{pipe} = (m1, m2)$.

These behavioral constraints are obtained by modifying the inputs to the *internal* sequential masters. This is realized by a logic denoted $InpTrans$ splitting the inputs into two sets of inputs for $m1$ and $m2$. Additionally, we need to combine the outputs of the two internal masters to generate the outputs of a single master. This combinatorial function is called $TransOut$ in the following.

If we execute two sequential masters in parallel, the state space of $m_{pipe}.state$ is equal to the cartesian product of the state components of the two sequential masters.

$$m_{pipe}.state \in m_{seq}.state \times m_{seq}.state$$

The key question is now which behavior has to be excluded from the purely interleaved execution of both control automatons. We aim to obtain a behavior where $m1$ dominates $m2$ in the sense that $m2$ only becomes active in situations where the sequential master could not executed the required behaviour. We have the following two key properties which have to be maintained: (i) $m2$ never requests the bus and (ii) $m2$ only leaves the idle state in case $m1$ is at the end of the address phase and an additional, parallel transfer from m_{pipe} is required. The first property is required to prohibit the situation that the bus is granted to m_{pipe} and the master would have to 'decide' which of the sequential masters requested the bus (and has stored the data to transmit). Moreover, the arbiter cannot handle two requests from one master in the sense to be able to grant the bus twice to that master.

These properties are realized in the following way: The first one is obtained by restricting the $startreq$ input of $m2$ and adjusting the $busy$ output to the host. The latter has to be done in order to preserve the property that a transfer is handled correctly if the $busy$ signal is inactive. The second property is achieved by restricting the $grant$ input of $m2$.

Next we define the $InpTrans$ function which splits the inputs of the master and applies the described restrictions. Then we specify the pipelined master in terms of a transition function using the $InpTrans$ function.

Definition 7 (Input Transformation) *Given a master m_{pipe}. Let $inp = (startreq, hreq, rdata, rdy, grant)$ denote its inputs with $hreq = (hwr, haddr, hwdata)$. Then we define the function $InpTrans$ by:*

$$\begin{aligned} &InpTrans\ inp = let \\ &\quad startreq_2 = startreq \wedge grant \wedge (m1.state = addr) \\ &\quad grant_2 = grant \wedge (m1.state = addr) \\ &in \\ &\quad (inp, (startreq_2, hreq, rdata, rdy, grant_2)) \end{aligned}$$

Definition 8 (Master for Pipelined Transfers) *Given a pipelined master $m_{pipe} = (m1, m2)$ and let dM_{seq} denote the transition function from the sequential master as defined in Section III-C. Moreover, let $inp = (startreq, hreq, rdata, rdy, grant)$ denote the inputs of m_{pipe} with $hreq = (hwr, haddr, hwdata)$. Then, the pipelined master is defined by the following transition function.*

$$\begin{aligned} &dM_{pipe}\ m_{pipe}\ inp = let \\ &\quad (inp_1, inp_2) = InpTrans\ inp \\ &in \\ &\quad (dM_{seq}\ m1\ inp_1, dM_{seq}\ m2\ inp_2) \end{aligned}$$

Finally, the outputs are obtained in a straight forward way. The sequential master which is currently in the $addr$ or $data$ state provides the corresponding bus outputs. Additionally, the req_{out} signal to the arbiter is only triggered by $m1$.

The only non-obvious computation is the $busy$ signal. Obviously, the pipelined master has to be busy if both sequential masters are busy. Additionally, the second sequential master is not allowed to request the bus. Therefore we have to enable the busy signal in cases where the first master is busy and the second master would have to request the bus after a $startreq$ signal. There are three of those cases, namely (i) both sequential masters are in the $idle$ state and the first master is waiting for a $grant$ signal, (ii) $m1$ is in the address phase and the bus is no more granted to the master, and (iii) $m1$ is in the data phase and neither the bus is granted anymore nor the rdy signal is active.

Definition 9 (Output Transformation) *Given a master m_{pipe}. Then its outputs are obtained from the outputs of the sequential masters $m1$ and $m2$ in the following way:*

$$\begin{aligned}
req^t_{out} &= m1.req^t_{out} \\
x^t_{out} &= \textit{if } (m2.state^t = addr) \\
&\quad \textit{then } m2.x^t_{out} \textit{ else } m1.x^t_{out} \\
wdata^t_{out} &= \textit{if } (m2.state^t = data) \\
&\quad \textit{then } m2.wdata^t \textit{ else } m1.wdata^t \\
busy^t_{out} &= (m1.busy^t \wedge m2.busy^t) \\
&\quad \vee\ (m1.busy^t \wedge m1.state^t = idle \wedge \\
&\qquad m2.state^t = idle) \\
&\quad \vee\ (m1.state^t = addr \wedge \neg grant^t) \\
&\quad \vee\ (m1.state^t = data \wedge \neg(grant \wedge rdy))
\end{aligned}$$

where $x \in \{trans, addr, wr\}$.

Recall that the $grant$ vector only changes in a cycle when rdy is active. Thus if the the bus is granted to a master this holds until the next active rdy signal and vice versa.

Next we argue about the local correctness of the pipelined master. Similar to the local correctness of the sequential master, we split cases into host interface correctness and bus correctness. The correctness with respect to the host interface reads similar to for the sequential system. The only difference is that we also have to ensure that there is a cycle when $busy$ is inactive before the data phase (i. e. at $a(i)$) if the bus is still granted. This is required to allow pipelined data transfers. The proof is slightly more complex because one has to argue about two consecutive transfers instead of one as they are executed partially in parallel.

The correctness with respect to the bus is also the same as for the sequential system. This maybe surprising fact results from the pipelined transfer definition which encapsulates the parallel execution property. This leads to clean correctness statements but has to be considered during the proof.

Lemma 8 (Local Master Correctness wrt. Bus) *Given a master m_{pipe}, a pipelined transfer $tr_{pipe}(i)$, and a corresponding host request $hreq^{trt(i)}$. Then the local correctness predicate for the sequential system from Definition 4 also holds for the pipelined master, thus $lcorr(m_{pipe}, tr_{pipe}(i), hreq^{trt(i)})$.*

Proof: This lemma is obtained from: (i) the correctness properties for $m1$ and $m2$, (ii) $m1$ always executes *in advance* to $m2$, i. e. for a sequence of transfers for which the master is constantly granted the bus, $m1$ always executes the odd transfers within this sequence and $m2$ the even ones. (iii) induction on the length of the sequence of consecutive granted transfers.

■

B. Global Correctness

The global correctness of the pipelined system is obtained completely analogous to the sequential argumentation. The two invariants have to be slightly modified.

Invariant 1 (master uniqueness) needs to be adopted to take the parallel execution of address and data phase into account. Moreover, the new state space has to be considered and whether a master granted the bus for consecutive transfers.

In Invariant 2 (slave uniqueness) not all slaves are in the idle state during address phase since the address phase of a transfer is equal to the data phase of the previous transfer. Hence, it has to be modified to state that during the address phase of a transfer, the only slave not in the $idle$ state is the slave address by the master specified by the old grand vector $dgrant$ of the arbiter at time $a(i)$.

Finally, the global correctness statement reads the same as for the sequential design in Lemma 7. Similarly to the local correctness, this results from the abstract transfer definition but additionally from the fact that the properties regarding the address and data phase in the global correctness statement are formulated completely separately. The difficulties arising from the transfer pipelining has already been dealt with in the previous local correctness properties and the invariants.

V. Burst Protocol

In this Section we specify a transformation providing burst transfers applicable to either the sequential or the pipelined system. The presented transformation is general enough that there are only few cases where one has to distinguish whether a sequential or a pipelined design is extended. Therefore, we present the general transformation and accept the slightly increased complexity at some places.

A burst transfer of size $(bsize + 1)$ is a read or write transfer supposed to start a some base address $baddr$ transferring all data from address $baddr$ to $baddr+bsize$. We support burst transfers of arbitrary but fixed length up to a maximum denoted BM. The actual length of a transfer is specified during the corresponding host request. We assume that $BM = 2^b - 1$ for some $b \in \mathbb{N}$ and thus we use $bsize \in \mathbb{B}^b$.

The communication bus between masters and slaves stays as before, but to realize burst transfers, we need to introduce two additional signals from the master to the arbiter. Hence, besides the req signal, the master provides the outputs $bst \in \mathbb{B}$ and $bsize \in \mathbb{B}^b$ to the arbiter. These signals are used to signal a burst transfer request to the arbiter that has to ensure that the grant signal stays constant during a burst transfer.

As in the previous section, we focus on the master transformation and therefore, we only describe the arbiter and slave shortly.

The arbiter is obtained from the corresponding previous arbiter, i. e. sequential or pipelined, by adding two additional registers for every master to sample burst request data. That is upon an active $startreq$ signal from some master, the corresponding flag indicating a burst request is set if the bst bus signal is active. At the same time, the $bsize$ signal is sampled. When the bus is granted to a master with a pending burst request, the arbiter keeps the $grant$ vector stable for $bsize$ may transfers.

The slave is the same as in the corresponding sequential or pipelined system.

Finally, we adapt the transfer notation to support burst. These changes are more complex than the changes from sequential to pipelined transfers. Again, the basic principle is to add a component $bst(i)$ to indicate a burst transfer together with a field $bsize(i)$. The end of the address phase is now not only a single time point but a partial function assigning a address phase end time to every *sub-transfer* $n \in [0 : bsize(i) - 1]$.

Definition 10 (Abstract Burst Transfer) *The i-th abstract burst transfer $tr_{burst}(i)$ within the burst communication system is defined as the tuple*

$$\begin{aligned} &tr_{burst}(i) \\ &\quad = (gnt(i), id(i), g(i), a(i), d(i), bst(i), bsize(i)) \\ &\quad \in \mathbb{B}^{NM} \times \mathbb{B} \times \mathbb{N} \times (\mathbb{N} \to \mathbb{N}) \times \mathbb{N} \times \mathbb{B} \times \mathbb{B}^b \end{aligned}$$

where the components are defined as:

$$\begin{aligned} gnt(i) &= grant^{g(i)} \\ id(i) &= m[\langle gnt(i)\rangle_u].trans_{out}^{a(i)} \\ g(i) &= \begin{cases} 0 & : i = 0 \\ a(i-1, bsize(i-1)) & \\ \qquad : i > 0 \wedge (\textit{m pipelined}) & \\ d(i-1) & : i > 0 \wedge (\textit{m sequential}) \end{cases} \\ a(i,n) &= \begin{cases} \min\{t > g(i) \mid rdy^t\} & \\ \qquad : n = 0 \vee \neg bst(i) & \\ \min\{t > a(i, n-1) \mid rdy^t\} & \\ \qquad : n \in [1 : bsize(i)] \wedge bst(i) & \\ \textit{undefined} : \textit{otherwise} \end{cases} \\ d(i) &= \begin{cases} \min\{t > a(i, 0) \mid rdy^t\} & \\ \qquad : \neg bst(i) & \\ \min\{t > a(i, bsize(i)) \mid rdy^t\} & \\ \qquad : bst(i) & \end{cases} \\ bst(i) &= m[\langle gnt(i)\rangle_u].bst^{a(i,0)} \\ bsize(i) &= m[\langle gnt(i)\rangle_u].bsize^{a(i,0)} \end{aligned}$$

The main difference between this definition and the previous is the case split on actual burst transfers. In case of a non-burst transfer the above definition resolves to one of the previous transfer definitions depending on

the kind of master we are extending by burst support.

A. Master

In the following we present the transformation to add burst transfer support to one of the previous masters. The basic idea is simple: We add a counter for the *burst sub-transfers* and *simulate* a sequence of $bsize$ many standard transfers. Thereby, the arbiter correctness ensures that the bus is granted during the whole burst transfer. We specify the whole transformation again in terms of an input transformation $InpTrans$ and an output transformation $TransOut$.

The interface from host to master has to be extended by two new signals: Upon a host request, $hbst \in \mathbb{B}$ signals that the current transfer is a burst request of size $hbsize \in \mathbb{B}^b$. Additionally, we introduce a signal to the host called $bdataupd \in \mathbb{B}$. It is used to signal that the data in $wdata$ has to be updated for a burst write access or a data chunk from a burst read access can be read from the $rdata$ bus. Hence, we require that the host updates the $wdata$ register at the time $bdataupd$ is active for a burst write access and the host hast to read the data from $rdata$ in the cycle after $bdataupd$ was active. We define $bdataupd$ in detail within the specification of $TransOut$.

Note that this host interface requirement could be discharged by extending the master with a data memory of address width b. This would not complicate the subsequent correctness argumentation significantly but lengthen the specification of the transformations. Therefore we omit it here.

We also have to extend the configuration of the master by three new components to handle burst requests: $bst \in \mathbb{B}$ and $bsize \in \mathbb{B}^b$ are used to sample the corresponding host signals upon a request. $bfst \in \mathbb{B}$ indicates if the first burst sub-transfer has to be sent. This flag is needed because the local address register has to be incremented before every burst sub-transfer except the very first. Thus, the configuration of a burst master is defined as the tuple

$$m_{burst} = m_x \times (bst, bsize, bfst)$$

where x is either seq or $pipe$.

In contrast to the $InpTrans$ for the pipelined master, this transformation is not only a combinatorial circuit but a state-representing extension to the master containing the newly introduced configuration parts. Thus, we describe the $InpTrans$ box by a next state function $dInpTrans$ and denote its output function by $InpTrans$ which is defined afterwards.

Definition 11 (Input Transformation) *Given a master m_{burst} supporting burst transfers. Let $inp = (startreq, hreq, rdata, rdy, grant)$ denote its inputs with $hreq = (hwr, haddr, hwdata, hbst, hbsize)$ and let m_x denote the master which is extended. Moreover, let $done = (bsize = 0)$ and*

$$bupd = \begin{cases} rdy \wedge \neg bfst & : x = pipe \\ rdy \wedge (state = data) & : x = seq \end{cases}$$

Then we define $dInpTrans$ by:

$$dInpTrans\ (bst, bsize, bfst)\ inp = let$$

$$bfst' = \begin{cases} 1 & : startreq \wedge \neg m_x.busy \\ & \quad \wedge\ hbst \wedge \neg(grant \wedge rdy) \\ 0 & : rdy \wedge grant \\ bfst & : otherwise \end{cases}$$

$$bst' = \begin{cases} hbst & : startreq \wedge \neg m_x.busy \\ 0 & : bupd \wedge done \\ bst & : otherwise \end{cases}$$

$$bsize' = \begin{cases} hbsize & : startreq \wedge \neg m_x.busy \\ bsize - 1 & : bupd \wedge bst \wedge \neg done \\ bsize & : otherwise \end{cases}$$

$$in$$

$$(bst', bsize', bfst')$$

The outputs generated by the $InpTrans$ box are specified by the $InpTrans$ function defined in the following.

Definition 12 (Input Transformation Signals) *Let $(bst, bsize, bfst)$ denote the current state of the InpTrans box and let $inp = (startreq, hreq, rdata, rdy, grant)$ denote its inputs with $hreq = (hwr, haddr, hwdata, hbst, hbsize)$. Additionally, let m_x denote the master which is extended, $done = (bsize = 0)$, and*

$$bupd = \begin{cases} rdy \wedge \neg bfst & : x = pipe \\ rdy \wedge (state = data) & : x = seq \end{cases}$$

Moreover let $nextbst$ be a shorthand for $bupd \wedge bst \wedge$

$\neg done$. *Then, the output function* $InpTrans$ *is given by:*

$$
\begin{array}{l}
InpTrans\ (m_x, bst, bsize, bfst)\ inp = let \\
\quad startreq_{burst} = startreq \vee nextbst \\
\quad wr_{burst} = \begin{cases} m_x.wr_{out} : & nextbst \\ hwr : & otherwise \end{cases} \\
\quad addr_{burst} = \begin{cases} m_x.addr_{out} + 1 : & nextbst \\ haddr : & otherwise \end{cases} \\
in \\
\quad (startreq_{burst}, wr_{burst}, addr_{burst})
\end{array}
$$

Now, we can define the master supporting burst transfers based on a previous master and the presented input transformation.

Definition 13 (Master for Burst Transfers) *Given a master* $m_{burst} = (m_x, bst, bsize, bfst)$ *and the transition function* dM_x *for* x *either seq or pipe. Moreover, let* $inp = (startreq, hreq, rdata, rdy, grant)$ *denote its inputs with* $hreq = (hwr, haddr, hwdata, hbst, hbsize)$. *Then, the burst master is defined by the following transition function.*

$$
\begin{array}{l}
dM_{burst}\ m_{burst}\ inp = let \\
\quad (startreq_{burst}, wr_{burst}, addr_{burst}) = \\
\qquad InpTrans\ (m_x, bst, bsize, bfst)\ inp \\
\quad m'_x = dM_x\ m_x\ (startreq_{burst}, wr_{burst}, addr_{burst}, \\
\qquad hdata, grant, rdy) \\
in \\
\quad (m'_x, dInpTrans\ (bst, bsize, bfst)\ inp)
\end{array}
$$

Finally, we specify the output transformation. Except for the $busy$ and $bdataupd$ outputs, the outputs remain the same as for the previous masters. The signal to update the burst data is given by $bdataupd^t = bupd^t \wedge bst^t \wedge \neg done^t$ where $done$ denotes $bsize = 0$ as before. The $busy$ signal has to be adapted for the case a burst access is in progress. Usually the $busy$ signal turns off at the end of a request if the bus is still granted, i. e. at time $d(i)$ for a single non-burst transfer. The $busy$ signal is modified such that it remains active during a burst transfer. Thus we obtain $busy^t = m_x.busy^t \vee (bst^t \wedge \neg done)$. For all other outputs, $TransOut$ is just the identity.

In the remaining of this section, we argue about the local correctness of the presented construction. The local correctness of the master supporting burst transfers is again split into the correctness regarding the host interface and regarding the bus. The correctness with respect to the host interface is very similar to the previous ones, extended by the correct sampling of the $wdata$ for the sub-transfers during a burst request. Since the sequential or pipelined master can only update all host data registers at the same time, the transition function of the burst master simulates a completely new transfer with the same wr, the increased $addr$, and the new $wdata$.

As before, the local correctness with respect to the bus is the more interesting case. Here we spilt cases on burst transfers: (i) in case of a non-burst transfer, the master has to obey the local correctness statement from the sequential (Lemma 4) or pipelined (Lemma 8) master. (ii) in case of a burst transfer, it has to simulate a sequence of locally correct single transfers as defined in Definition 14.

Definition 14 (Local Master Correctness) *Given a master* m_{burst}, *a transfer* $tr_{burst}(i)$, *and a burst host request* $hreq^{trt(i)}$ *on* m_{burst}. *Moreover, let* $0 < n \wedge n < bsize(i) - 1$. *In order to increase readability, we omit the transfer index* i *in the following. Then, we define the predicate indicating local burst correctness* $lbcorr(m_{burst}, tr_{burst}, hreq^{trt})$ *by:*

$lbcorr(m_{burst}, tr_{burst}, hreq^{trt})$ *holds iff either* $m_{burst} \neq \langle gnt \rangle_u$ *or* $bst = 0$ *or* $m_{burst} = \langle gnt \rangle_u$, $bst = 1$, *and*

$$
\begin{array}{l}
lcorr\big(m_{burst}, (1, gnt, g, a(0), a(1)), \\
\quad (hwr^{trt}, haddr^{trt}, hwdata^{trt})\big) \quad \wedge \\
lcorr\big(m_{burst}, (1, gnt, a(n-1), a(n), a(n+1)), \\
\quad (hwr^{trt}, haddr^{trt} + n, hwdata^{a(n-1)})\big) \quad \wedge \\
lcorr\big(m_{burst}, (1, gnt, a(bsize-2), a(bsize-1), d), \\
\quad (hwr^{trt}, haddr^{trt} + bsize - 1, hwdata^{a(bsize-2)})\big)
\end{array}
$$

Finally, the argumentation about the global correctness is straight forward analogous to the global correctness of the sequential or pipelined design. As for the local correctness one splits cases if the requested transfer is a single or a burst transfer. For single transfer the global correctness of the corresponding design holds. For a burst transfer the local correctness statement from Definition 14 is lifted to the global level the same way as done for the sequential or pipelined design.

VI. Conclusion and Future Work

As on-chip communication systems have become more complex to meet performance requirements, their formal verification has become infeasible in reasonable time and effort with current post-hoc verification methodologies. Our new approach tries to resolve this by introducing a new methodology to design and verify communication protocols in an algebraic fashion. This approach is based on correctness-preserving or correctness-inheriting transformations to a simpler design providing enhanced functionality.

The results presented here are only initial work. Besides tackling more transformations, future work will also have to address the following issues. First, we need to find a more systematic method for applying transformations. For example, the pipelined design presented here is obtained by a parallel composition of two sequential masters and imposing extra constraints of the inputs (Definition 7) and combining the outputs in an appropriate way (Definition 9). In this initial work these constraints were devised by hand, but we want to provide a more systematic approach to this, either through a formal analysis such as model checking or by adopting a notational structure that makes them evident.

Second, we will need to devise a range of refinement relations to link different abstraction levels of a model; either temporal, logical, or a combination of them. We need to find methods to reason about the relationships between communications at an implementation level, e. g. register transfers, and a more abstract, system level. This is needed, for example, to relate an atomic, high-level transfer to non-atomic communications possibly spread over space and time on a cycle-accurate register transfer level.

Third, we must address the problem of incorporating methods to tackle problems evolving from high-performance aspects on gate-level or even physical layers. For example, a desirable goal is to develop a method to argue about low-level timing constraints or distributed, asynchronous clocks within our methodology.

Finally, we emphasise that our main interest for future development of these ideas is not to analyse AMBA or standard system-on-chip bus protocols. Our aim is develop a framework for the design and formal verification of scalable, high-performance communication platforms that allow customisation.

References

[1] *AMBA Specification Revision 2.0*, ARM, 1999. [Online]. Available: http://www.arm.com

[2] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, ser. LNCS. Springer, 2002, vol. 2283.

[3] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. R. Marco Pistore, R. Sebastiani, and A. Tacchella, "NuSMV 2: An open source tool for symbolic model checking," in *CAV '02*. Springer-Verlag, 2002, pp. 359–364.

[4] S. Tverdyshev, "Combination of Isabelle/HOL with automatic tools," in *FroCoS 2005, Vienna Austria*, ser. LNCS, vol. 3717. Springer, 2005, pp. 302–309.

[5] A. Roychoudhury, T. Mitra, and S. Karri, "Using formal techniques to debug the amba system-on-chip bus protocol," *Design, Automation and Test in Europe Conference and Exhibition, 2003*, pp. 828–833, 2003.

[6] H. Amjad, "Model checking the AMBA protocol in HOL," University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-602, Sep. 2004. [Online]. Available: http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-602.pdf

[7] J. Schmaltz and D. Borrione, "Towards a formal theory of on chip communications in the acl2 logic," in *ACL2 '06: Proceedings of the sixth international workshop on the ACL2 theorem prover and its applications*. New York, NY, USA: ACM, 2006, pp. 47–56.

[8] F. Müffke, "A better way to design communication protocols," Ph.D. dissertation, University of Bristol, May 2004. [Online]. Available: http://www.cs.bris.ac.uk/Publications/Papers/2000199.pdf

[9] C. Seger, "The design of a floating point unit using the integrated design and verification (idv) system," in *Sixth International Workshop on Designing Correct Circuits: Participants' Proceedings*, M. Sheeran and T. Melham, Eds., March 2006.

[10] M. Aagaard, B. Cook, N. A. Day, and R. B. Jones, "A framework for microprocessor correctness statements," in *CHARME '01: Proceedings of the 11th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*. London, UK: Springer-Verlag, 2001, pp. 433–448.

[11] P. Chatterjee, H. Sivaraj, and G. Gopalakrishnan, "Shared memory consistency protocol verification against weak memory models: Refinement via model-checking," in *CAV '02: Proceedings of the 14th International Conference on Computer Aided Verification*. London, UK: Springer-Verlag, 2002, pp. 123–136.

[12] A. Datta, A. Derek, J. C. Mitchell, and D. Pavlovic, "Abstraction and refinement in protocol derivation," in *CSFW '04: Proceedings of the 17th IEEE workshop on Computer Security Foundations*. Washington, DC, USA: IEEE Computer Society, 2004, p. 30.

Describing Hardware with Parallel Programs

David Greaves
Computer Laboratory
University of Cambridge
Cambridge CB3 0FD
United Kingdom
David.Greaves@cl.cam.ac.uk

Satnam Singh
Microsoft Research Cambridge
Cambridge CB3 0FB
United Kingdom
satnams@microsoft.com

Abstract

This paper explores the idea of using regular parallelism and concurrency constructs in mainstream languages for the description, implementation and verification of digital circuits. We describe the Kiwi parallel programming library and its associated synthesis system which is used to transform parallel programs into circuits for realization on FPGAs. Although there has been much work on compiling sequential C-like programs to hardware by automatically discovering parallelism, we work by exploiting the parallel architecture communicated by the designer through the choice of parallel and concurrent programming language constructs. In particular, we describe a system that takes .NET *assembly language with suitable custom attributes as input and produces Verilog output which is mapped to FPGAs. We can then choose to apply analysis and verification techniques to either the high-level representation in C# or other* .NET *languages or to the generated RTL netlists. For example, we show how a parallel combinator style description can be used to describe a parallel sorting network in F#. A distinctive aspect of our approach is the exploitation of existing language constructs for concurrent programming and synchronization which contrasts with other schemes which introduce specialized concurrency control constructs to extend a sequential language. One of our aims is to be able to deploy formal analysis techniques developed for parallel programs to the verification of digital circuits represented as parallel programs.*

1 Introduction

The design of correct circuits has given rise to many formal techniques and tools which try to prove properties about digital circuits. The design of correct programs is also exploiting formal analysis techniques and in particular formal analysis techniques are a valuable tool for the challenging task of writing correct concurrent and parallel programs. This paper explores the idea that we can gain practical benefits from describing digital circuits has regular parallel and concurrent programs which can then be automatically transformed into digital circuits. Using this approach lets us exploit the rich set of tools software engineers have available for writing and debugging software. Furthermore, any verification techniques that are applicable to parallel programs should also benefit the verification of parallel programs that represent digital circuits. This gives rise to the possibility of a unified set of tools and techniques that are applicable to software and hardware.

A significant amount of valuable work has already been directed at the problem of transforming sequential imperative software descriptions into good-quality digital hardware and these techniques are especially good at control-orientated tasks which can be implemented with finite-state machines. Our approach builds upon this work by proposing the use of parallel software descriptions which capture more information from the designer about the parallel architecture of a given problem that can then be exploited by our tools to generate good-quality hardware for a wider class of descriptions.

A novel contribution of this work is a demonstration of how systems-level concurrency abstractions, like events, monitors and threads, can be mapped onto appropriate hardware implementations. Furthermore, our system can process bounded recursive methods and object orientated constructs (including object pointers).

The output of our system has been tested with the Xilinx XST synthesis flow and we report the experience of producing circuits that perform low level bus control and parallel implementations of test pattern images for a DVI controller. These circuits were implemented on a Virtex-5 FPGA on the ML-505 development board.

Throughout this paper when when we refer to an 'assembly' language file we specifically mean the textual representation of the byte code file produced by our compilation flow rather than an a .NET assembly which is an altogether different entity.

Although we present work in the context of the .NET system the techniques are applicable to other platforms like the Java Virtual Machine (JVM). The experimental work described in this paper was undertaken on Windows machines and also on Linux machines running the Mono system.

This paper starts off with background review of related work on compiling C-like descriptions into hardware. We then describe the Kiwi library which provide systems level concurrency abstractions for writing parallel programs that model circuits. A synthesis flow which transforms compiled versions of these programs into Verilog netlists is presented and then we illustrate the flow with two examples: an I2C controller and a DVI-based graphics application. Finally we show how a parallel sorting networking can be described in terms of a parallel composition combinator in F#.

2 Background

A lot of previous research has been directed at transforming sequential C-like programs into digital circuits and this work is strongly related to work on automatic parallelization. Indeed, it is instructive to notice that C-to-gates synthesis and automatic parallelization are (at some important level of abstraction) the same activity although research in these two areas has often occurred without advances in one community being taken up by the other community.

The idea of using a programming language for digital design has been around for at least two decades [3]. Previous work has looked at how code motions could be exploited as parallelization transformation technique [6].

Examples of C-to-gates systems include Handel-C [5], the DWARV [8] C-to-VHDL system from Delft University of Technology, single-assignment C (SA-C) [7], ROCCC [1], SPARK [4], CleanC from IMEC and Streams-C [2].

Some of these languages have incorporated constructs to describe aspects of concurrent behavior e.g. the **par** blocks of Handel-C. The Handel-C code fragment below illustrates how the **par** construct is used to identify a block of code which is understood to be in parallel with other code (the outer **par** on line 1) and a parallel for loop (the **par** at line 4).

```
1  par
2    { a[]0] = A; b[0] = B;
3      c[0] = a[0]b[0] == 0 ? 0 : b[0] ;
4      par (i = 1; i < W; i++)
5      { a[i] = a[i−1] >> 1 ;
6        b[i] = b[i−1] << 1 ;
7        c[i] = c[i−1] + (a[i][0] == 0 ? 0 : b[i]);
8      }
9      ∗C = c[W−1];
10   }
```

Our approach involves providing hardware semantics for existing low level concurrency constructs for a language that already supports concurrent programming and then to define features such as the Handel-C **par** blocks out of these basic building blocks in a modular manner.

3 Parallel Circuit Descriptions

We provide a conventional concurrency library, called Kiwi, that is exposed to the user and which has two implementations:

- A software implementation which is defined purely in terms of the supporting .NET concurrency mechanisms (events, monitors, threads).
- A corresponding hardware semantics which is used to drive the .NET IL to Verilog flow to generate circuits.

A Kiwi program should always be a sensible concurrent program but it may also be a sensible concurrent circuit.

A main paradigm in parallel programming is thread forking, with the user writing something like:

```
ConsumerClass consumer =
        new ConsumerClass(...);

Thread thread1 =
        new Thread(new ThreadStart(consumer.process));
thread1.Start();
```

Within the Kiwi hardware library, the .NET library functions that achieve this are implemented either by compilation in the same way as user code or using special action. Special action is triggered when the newobj ThreadStart is elaborated: the entry point for the remote thread is recorded and effectively added as a new -root argument. On the other hand, the call to Threading::Start that enables the thread to run is implemented entirely C# (and hence compiled to hardware) simply as an update to a fresh gating variable that the actual thread waits on before starting its normal behavior.

Another important paradigm in parallel composition is the *channel*. This uses blocking read and write primitives to convey a potentially composite item, of generic type T, atomically.

```
1  public class channel<T>
2  { T datum;
3    bool empty = true;
4    public void write(T v)
5    { lock(this)
6      { while (!empty)
7          Monitor.Wait(this) ;
8        datum = v ;
9        empty = false ;
10       Monitor.Pulse(this);
11     }
12   }
13
14   public T read()
15   { T r ;
16     lock (this)
17     { while (empty)
18         Monitor.Wait(this);
19       empty = true;
20       r = datum;
21       Monitor.Pulse(this);
22     }
23     return r;
24   }
25 }
```

The **lock** statements on lines 5 and 16 are translated by the C# compiler to calls to Monitor.Enter and Monitor.Exit with the body of the code inside a try block whose finally part contains the Exit call.

There are numerous levels at which we might introduce primitives when implementing parts of the Kiwi library for hardware synthesis. An entire function can be recognized and translated to the primitives of the underlying virtual machine. Alternatively, the C# code from the software implementation can be partially translated. In our current implementation of channels, calls to Monitor.Enter and Monitor.Exit were replaced with the following C# code (containing only native functions understood by the core compiler)

```
void Enter(object mutex)
{ while (hpr_testandset(mutex, 1))
    hpr_pause();
}
void Exit(object mutex)
{ hpr_testandset(mutex, 0);
}
```

Monitor.Wait was replaced with

```
void Wait(object mutex)
{ hpr_testandset(mutex, 0);
  hpr_pause();
  while (hpr_testandset(mutex, 1))
    hpr_pause();
}
```

and Monitor.Strobe was treated as a NOP (no operation), because the underlying hardware implementation is intrinsically parallel and can busy wait without cost.

4 Synthesis Flow

In our flow, shown in Figure 1, the .NET Assembly Language is parsed to an AST and then a CIL (common intermediate language) elaborate stage converts the AST to the internal form known as an HPR machine. This was used because a library of code from the University of Cambridge can operate on this format. The HPR machine contains imperative code sections and assertions. The library

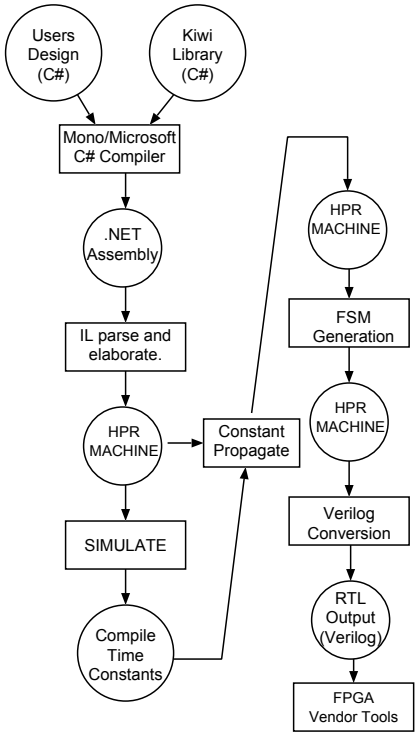


Figure 1: Overall System Flow.

enables HPR machines to be emitted in various hardware and software forms and converted between them. The imperative code code sections can be in series or parallel with each other, using Occam-like SER and PAR blocks.

For illustration, we show some IL code below. Key aspects of the IL code include the use of a stack rather than registers (e.g. mul pops two elements off the stack, multiplies them and pushes the result onto the stack); local variables stored in mutable state (e.g. ldloc.1 pushes the value at local memory location 1 onto the stack); control flow through conditional and unconditional branches; and direct support for overloaded method calls.

```
IL_0019: ldc.i4.1
IL_001a: stloc.0
IL_001b: br IL_005b
IL_0020: ldc.i4.1
IL_0021: stloc.1
IL_0022: br IL_0042
IL_0027: ldloc.0
IL_0028: ldloc.1
IL_0029: mul
IL_002a: box [mscorlib]System.Int32
IL_002f: ldstr " "
IL_0034: call string string::Concat(object, object)
```

The IL elaboration stage subsumes a number of variable present in the input source code, including all object pointers.

The resulting machine is simulated with all inputs set to don't know. Any variables which are assigned a constant value and not further assigned in the body of the program (i.e. that part which is not deterministic given uncertain inputs) are determined as compile-time constants and subsumed by the constant propagation function shown in Figure 1. Constructor code must not depend on run-time inputs.

The resulting HPR machine is an array of imperative code for each thread, indexed by a program counter for that thread. There is no stack or dynamic storage allocation. The statements are: assign, conditional branch, exit and calls to certain built-in functions, including hpr_testandset(), hpr_printf() and hpr_barrier(). The expressions occurring in branch conditions, r.h.s. of assignment and function call arguments still use all of the arithmetic and logic operators found in the .NET input form. In addition, limited string handling, including a string concat() function are handled, so that console output from the

.NET input is preserved as console output in the generated forms (e.g. $display() in Verilog RTL).

4.1 .NET Assembly Language Elaboration

From the .NET AST, an hierarchic dictionary is created containing the classes, methods, fields, and custom attributes. Other declarations, such as processor type, are ignored.

A variables is either a static or dynamic object field, a top-level method formal, a local variable, or a stack location. For each variable we decide whether to *subsume* it in the elaboration. If not subsumed, it appears in the VM code that is fed to the next stage (where it may then get subsumed for other reasons). Variables that are subsumed during elaboration always include object and array handles.

We perform a symbolic execution of each thread at the .NET basic block level and emit VM code for each block. .NET label names that are branch destinations define the basic block boundaries and these appear verbatim in the emitted VM code.

Although .NET byte-code defines a stack machine, no stack operations are emitted from the .NET processing stage. Stack operations within a basic block are symbolically expanded and values on the stack at basic block boundaries are stored and loaded into stack variables on exit and entry to each basic block. The stack variables are frequently subsumed, but can appear in the VM code and hence, from time-to-time, in the output RTL.

A -root command line flag enables the user to select a number of methods or classes for compilation. The argument is a list of hierarchic names, separated by semicolons. Other items present in the .NET input code are ignored, unless called from the root items.

Where a class is selected as the root, its contents are converted to an RTL module with I/O terminals consisting of various resets and clocks that are marked up in the C# source using attributes defined in the Kiwi library. Where a method is given as a root component, its parameters are added to the formal parameter list of the RTL module created. Where the method code has a preamble before entering an infinite loop, the actions of the preamble are treated in the same way as constructors of a class, viz. interpreted at compile-time to give initial or reset values to variables. Where a top-level method exits and returns a non-void value, an extra parameter is added to the RTL module formal parameter list. Additional fields may be declared as I/O terminals using attribute mark up within the C# source code:

```
[OutputBitPort(''scl'')]
static bool scl;
[InputBitPort(''sda_in'')]
static bool sda_in;
```

RTL languages, such as Verilog an VHDL, do not support the common software paradigm of method entry points and upcalls. Threads are not allowed to cross between separately-compiled modules. To provide C# users with procedure call between separately-compiled sections, while keeping our output RTL synthesisable to FPGA, we must provide further connections to each RTL module that implement RPC-like stubs for remote callers.

Certain restrictions exist on the C# that the user can write. Currently, in terms of expressions, only integer arithmetic and limited string handling are supported, but floating point could be added without re-designing anything, as could other sorts of run-time data. More importantly, we are generating statically allocated output code, therefore:

1. arrays must be dimensioned at compile time
2. the number of objects on the heap is determined at compile time,
3. recursive function calling must bottom out at compile time and so the depth cannot be run-time data dependent.

The start-up path is split off from the main process loop by running an interpreter on the code up to the first blocking primitive or to the top of the first loop that cannot be unwound (for reasons of it being infinite or the number of trips depending on run time data values). The main process loop of each thread is converted to an LTS by defining a sequencer. The sequencer is an automata with a state defined for each blocking primitive and for each .net program label that is the destination for more than one flow of control. The I/O operations and operations on local variables performed by the program are all recorded against the appropriate arc of the sequencer.

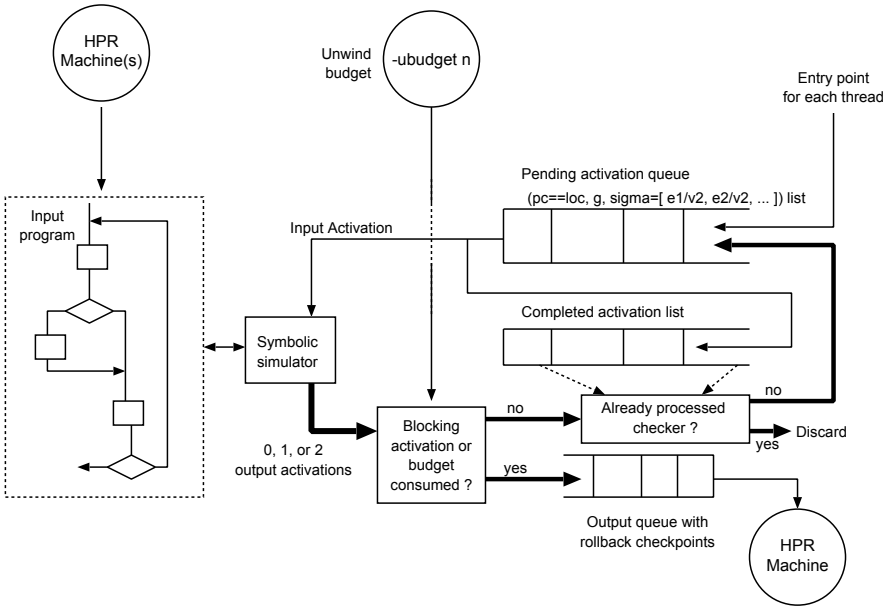


Figure 2: Conversion of control flow graph to FSM.

4.2 FSM Generation

The input and output to the FSM generation stage are both HPR machines. Each input machine consists of an array of instructions addressed by a program counter. Each instruction is either an assignment, exit statement, built-in primitive or conditional branch. The expressions occurring in various fields of the instructions may be arbitrarily complicated, containing any of the operators and referentially-transparent library calls present in the input language, but their evaluation must be non-blocking.

The output machine consists of an HPR parallel construct for each clock domain. The parallel construct contains a list of finite-state-machine edges, where edges have two possible forms:

```
(g, v, e)
(g, f, [ args])
```

where the first form assigns `e` to `v` when `g` holds and the second calls built-in function `f` when `g` holds.

An additional input, from the command line, is an unwind budget: a number of basic blocks to consider in any loop unwind operation. Where loops are nested or fork in flow of control, the budget is divided amongst the various ways. Alternatively, in the future, the resulting machine can be analyzed in terms of meeting a user's clock cycle target and the unwinding decisions can be adjusted until the clock budget is met.

The central data structure is the pending activation queue (Figure 2), where an activation has form $(p == v, g, \sigma)$ and consists of a program counter (p) and its current value (v), a guard (g) and an environment list (σ) that maps variables that have so far been changed to their new (symbolic) values. The guard is a condition that holds when transfer of control reaches the activation.

Activations that have been processed are recorded in the completed activation queue and their effects are represented as edges written to the output queue. All three queues have checkpoint annotations so that edges generated during a failed attempt at a loop unwind can be rolled-back.

The pending activation queue is initialized with the entry points for each thread. Operation removes one activation and symbolically steps it through a basic block of the program code, after which zero, one or two activations are returned. These are either converted to edges for the output queue or added to the pending activation queue.

An exit statement terminates the activation and a basic block terminating in a conditional branch returns two activations. A basic block is terminated with a single activation at a blocking native call, such as hpr_pause(). When returned from the symbolic simulator, the activation may be flagged as blocking, in which case it goes to the output queue. Otherwise, if the unwind budget is not used up the resulting activation(s) go to the pending queue. If the budget is used up, the system is rewound to the latest point where that activation had made some progress.

The basic rules for assignment and conditional branch, implemented by the symbolic simulator, with guard g and with environment σ, are:

$$\begin{aligned} [\![\texttt{v := e;}]\!]_{(n,g,\sigma)} &\rightarrow [(n+1, g, [[\![e]\!]_\sigma / v]\sigma)] \\ [\![\texttt{if (e) goto d;}]\!]_{(n,g,\sigma)} &\rightarrow [(d, g \wedge [\![e]\!]_\sigma, \sigma), \\ &\quad\ (n+1, g \wedge {}^\sim[\![e]\!]_\sigma, \sigma)] \end{aligned}$$

The conditional branch returns a list of two activations.

Activations are discarded instead of being added to the pending queue if they have already been successfully processed. Checking this requires comparison of symbolic environments. These are kept in a 'close to normal form' form so that typographical equivalence can be used. A more-powerful proof engine can be used to check equivalence between activations, but there will always be some loops that might be unwound at compile time that are missed (decidability).

Operation continues until the pending activation queue is empty.

The generated machine contains an embedded sequencer for each input thread, with a variable corresponding to the program counter of the thread and states corresponding to those program counter values of the input machine that are retained after unwinding. However, the sequencer is no longer explicit; it is just one of the variables assigned by the FSM edges. When only one state is retained for the thread, the program counter variable is removed and the edges made unconditional.

The output edges must be compatible. Compatible means that that no two activations contain a pair of assignments to the same variable under the same conditions that disagree in value. Duplicate assignments of the same value at the same time are discarded. This checking cannot always be complete where values depend on run-time values, with array subscript comparison being a common source of ambiguity. Where incompatibility is detected, an error is flagged. When not detected, the resulting system that can be non-deterministic.

The built-in hpt_testandset() function, operating on a mutex, m, solves non-determinism arising from multiple updates at the same time using an ordering that arises from the order the activations are processed. (Other, fairer forms of arbiter could also be implemented.) Mutexes are statically-allocated boolean values. The acquire operation returns the previous value from the symbolic environment, σ, of the activation, or the mutex itself if it is not present, while updating the environment to set the mutex. A clear operation is implemented as a straightforward reset of the mutex:

$$\begin{aligned} [\![\texttt{hpr\_testandset(m, 1)}]\!]_\sigma &\rightarrow (\sigma(m), [1/m]\sigma) \\ [\![\texttt{hpr\_testandset(m, 0)}]\!]_\sigma &\rightarrow (0, [0/m]\sigma) \end{aligned}$$

Multiple set and clear operations can occur within one clock cycle of the generated hardware with only the final value being clocked into the hardware register.

If all variables are kept in flip-flops, it is almost trivial to convert the HPR machine in FSM form to an RTL design. However, we map arrays in the C# source code to arrays to RAMs in hardware and scalar variables to flip-flops. In the future we will extend the Kiwi attributes set so that the user can control which variables are placed in which output RAMS. A static structural hazard occurs when the actions of a single transition require more than one operation out of a RAM, such as reads to a single-ported RAM at more than one location. Other expensive components, that must be shared, such as an ALUs, can also generate structural hazards. A dynamic structural hazard occurs when two different *threads* both try to use the same location at once. Static structural hazards are resolved at compile time, using rewrites that stall one FSM at the expense of another and introducing any necessary holding registers. Dynamic structural hazards can be resolved at run-time using arbitration circuits that delay the advancement of one FSM while a needed resource is in use by another.

5 I2C Controller Example

In this section we demonstrate how a circuit that performs communication over an I2C bus can be expressed using the Kiwi library. The motivation for tackling such an example arises from the fact that the typical coding style for such circuits involves hand coding state machines using nested case statements in VHDL (or equivalent features in Verilog). In particular, the sequencing of operations is tedious and error prone. However we can exploit the built in semi-colons of a conventional language to capture sequencing.

The example we use is a circuit that writes a series of values into the register of a DVI chip on the Xilinx ML-505 board. These registers are written using an I2C interface between the FPGA and the DVI chip. The code below demonstrates what is wrong with a typical VHDL implementation for performing I2C control, which also representative of many other kinds of control code:

```
case state is
  when initial
    => case phase is
         when 0 => scl <= '1' ; sda_out <= '1' ;
         when 1 => null ;
         when 2 => sda_out <= '0' ; -- Start condition
         when 3 => scl <= '0' ;
                   index := 6 ;
                   state := deviceID_state ;
         end case ;
  when deviceID_state
     => case phase is
          when 0 => sda_out <= deviceID (index) ;
```

Such nested **case** statements are typical of situations like this where the designer ends up hand coding a nested finite state machine to capture a sequence of operations i.e. we are missing the semi-colons of an imperative language. Using Kiwi, we can more directly represent the sequencing operations e.g. in the following deserialization code.

```
int deviceID = 0x76;
for (int i = 7; i > 0; i--)
{ scl = false;
  sda_out = (deviceID & 64) != 0;
  WaitForClk1(); // Set it i-th bit of the device ID
  scl = true; WaitForClk1(); // Pulse SCL
  scl = false; deviceID = deviceID << 1;
  WaitForClk1();
}
```

The call to WaitForClk1() in turn makes a call to a AutoResetEvent object by waiting for an event that corresponds to the rising edge of a clock signal.

This is is processed by our system which generates Verilog RTL code. Although not designed to be human readable, the RTL can then be processed by FPGA vendor tools to generate a programming bitstream:

```
always @(posedge clk)
  begin
  if (!((pcnet115p^42)))
      I2CTest_I2C_ProcessACK_fourth_ack1
      <= !((pcnet115p^42)) ? I2CTest_I2C_sda: 1'bx;
    if ((!(!(!(!((pcnet115p^41))&&!(!(I2CTest_I2C_sda))))))||
      !(!(!(!((pcnet115p^41))&&!(I2CTest_I2C_sda))))))
      i2c_demo_inBit <= (!((pcnet115p^41))&&!(
     !(I2CTest_I2C_sda))) ? 1: (!((pcnet115p^41))&&
     !(I2CTest_I2C_sda)) ? 0: 1'bx;
    if ((!((pcnet115p^44))||!((pcnet115p^39))))
      i2c_demo_versionNumber <= !((pcnet115p^44)) ?
      ((i2c_demo_versionNumber<<1)|i2c_demo_inBit):
      !((pcnet115p^39)) ? 0: 1'bx;
    if (!((pcnet115p^37))) I
      2CTest_I2C_ProcessACK_third_ack1 <=
      !((pcnet115p^37)) ? I2CTest_I2C_sda: 1'bx;
```

From the 143 lines of C# code we got 73 lines of generated Verilog which was fed to the Xilinx ISE 9.2.03i tools and a programming netlist was produced for the Virtex-5 XC5VLX50T part on the ML-505 development board. The source program performed a series of write operations to the registers of the Chrontel DVI chip over the I2C bus. The design was then used to successfully configure the Chrontel DVI chip. The generated design used 126 slice registers and 377 slice LUTS with a critical path of 5.8ns. A hand written version will be smaller because these results do not yet reflect the use of integer sub-ranges so some registers are represented with 32-bits rather than just a hand full of bits.

6 DVI Controller Example

This section describes the implementation of a DVI controller in behavioral VHDL and in C# using Kiwi. This design is used with the I2C controller presented in the pre-

vious section in a larger design which initializes a Chrontel DVI chip over I2C and then generates a test image.

The source C# program has at its heart a loop that computes the multiplexed RGB values to be send to the DVI inputs of a Chrontel DVI chip.

```
while (true)
  // For each line
  for (int y=0; y<525;y++)
    // For each column
    for (int x = 0; x < 800; x++)
    { Clocks.clk1.WaitOne();
      // HSYNC
      DVI_Ports.dvi_h
        = x > 640 + 16 && x < 640 + 16 + 96;
      // VSYNC
      DVI_Ports.dvi_v
        = y > 480 + 10 && y < 480 + 10 + 2;
      // DVI_DE
      DVI_Ports.dvi_de
        = x < 640 && y < 480;
      if (!DVI_Ports.dvi_de)
      // blank pixels
        for (int i = 0; i < 12; i++)
          DVI_Ports.dvi_d[i] = 0;
      else
      { // Compute pixel color
```

The call to Clocks.clk1.WaitOne() corresponds to a blocking call waiting for an event which corresponds to the rising edge of a clock signal and is one of the concurrency constructs provided by the .NET framework that we remap to hardware.

The test image draws various color bars and ramps and animations and our systems generates about 3,000 lines of Verilog which is synthesized into a design that uses 69 slice registers and 262 LUTs on a Virtex-5 XC5VLX50T part. The critical path for the generated design is 6.6ns which is sufficient for a 25MHz target pixel clock rate to generate a full color 640 by 480 image. Once again we expect performance improvements when we can prune logic through the use of integer sub ranges.

7 Describing Circuits with Parallel Combinators in F#

Having developed the basic infrastructure for representing and translating systems level concurrency abstractions into hardware we can build upon this layer to provide higher level idioms for parallel circuit descriptions. We illustrate such an approach by giving a parallel combinator style description of a parallel sorting network written in F#.

First we define a serial compositional combinator which gives us the notational convenience of describing circuits with a left to right data-flow:

```
let (>->) f1 f2 = function i -> f2 (f1 i)
```

The basic combinator that introduces parallelism is the parallel composition combinator par2 which can be defined in F# sequentially as shown below:

```
let par2 f1 f2 (a, b) -> (f1 a, f2 b)
```

A parallel version is easily defined by creating separate threads for the computation of f1 a and f2 b and waiting for both threads to complete. We will omit the details here because for an efficient parallel program one may wish to introduce more sophisticated mechanisms to control the overhead of scheduling many small work items e.g. by using threadpools.

Using the parallel combinator we can then build up a series of definitions which describe a butterfly network for merging using Batcher's bitonic merger.

First we define a couple of combinators for riffling and unriffling wires:

```
let riffle = halveList >-> zipList >-> unPair
let unriffle = pair >-> unzipList >-> unhalveList
```

The two combinator creates two parallel instances of a circuit r. It works by halving the input and feeding the bottom half into one instance of the circuit r and the top half of the input into the other parallel instance of the circuit r:

```
let two r = halve >-> par2 r r >-> unhalve
```

We have omitted the definitions of halve and unhalve.

The following higher order combinator creates two parallel instances of the circuit r with the inputs unriffled and the output riffled.

```
let ilv r = unriffle >−> two r >−> riffle
```

The evens definition applies several parallel instances of the circuit r to adjacent pairs of the input.

```
let rec chop n l =
  match l with
  [] −> []
  | l −> take n l :: chop n (drop n l)

let evens f = chop 2 >−> map f >−> concat
```

We now have enough machinery to recursively define Batcher's bitonic merger:

```
let rec bfly r n =
  match n with
  1 −> r
  | n −> ilv (bfly r (n−1)) >−> evens r
```

The merger can now be used to define Batcher's bitonic sorter:

```
let rec bsort n =
  match n with
  1 −> sort2
  | n −> two (bsort (n−1)) >−> sndList rev >−>
        bfly sort2 n
```

The sndList function applies the list reverse function to the second element of a list of pairs. In these descriptions the base two element sorter (sort2) are scheduled for parallel execution and in the corresponding circuits they are distinct instances of two sorters (all of which execute in parallel). To represent a realistic circuit these descriptions would be applied to streams of inputs and the base two sorter would be pipelined.

In addition to executing these descriptions as parallel software we can also translate them into .NET intermediate code and then use our system to transform this representation into a circuit.

8 Future Work

The techniques presented in this paper lay the groundwork for expressing higher-level concurrency and parallelism idioms in terms of lower level concurrency constructs in the form of events, monitors and threads.

Aggressive loop unwinding increases the complexity of the actions on each clock step in exchange for reducing the number of clock cycles used. Currently an unwind budget is given as a command line option but it may be worth exploring higher level ways of guiding such space/-time trade-offs.

In software designs, threads pass between separately compiled sections and update the variables in the section they are in. This is not supported in synthesisable RTL, so instead updates to a variable from a separately compiled section must be via a special update interface with associated handshaking for write and test and set. It would be interesting to explore others mechanisms for separate compilation and composability.

One initial source of inefficient circuits was the use of int types in C# which resulted in circuits with 32-bit ports after synthesis. Our fix for this problem involves attaching yet another custom attribute that specifies the range for integer values which can then be used by our system to generate bit-vectors of the appropriate size in Verilog. Another approach would have been to follow the example of System-C and provide a new type that encapsulates the idea of an integer range but we felt that this would be change that permeates the whole program in a negative way.

Our hypothesis for our future work is that because we have a good translation for the low level concurrency constructs into hardware then we should be able to translate the higher level idioms by simply implementing them in the usual way. An interesting comparison would be examine the output of our system when used to compile join patterns and then compare them to existing work on compiling join patterns in hardware Join Java.

Another direction to take our work is to generate code for other kinds of parallel computing resources like GPUs. It is not clear if we can continue to use the same concurrency abstractions that we have developed for Kiwi or if we need to add further domain specific constructs and custom attributes.

The F# descriptions presented in the previous section have not yet been translated into circuits as this requires further refinement of our synthesis flow e.g. the treatment of tail calls in .NET .

9 Conclusions

We have demonstrated that it is possible to write effective hardware descriptions as regular parallel programs and then compile them to circuits for realization on FPGAs. Furthermore, we have shown that we can transform programs written using conventional concurrency constructs and synchronization primitives into hardware. Specifically, we have provided translations for events, monitors, the **lock** synchronization mechanism and threads under specific usage idioms. By providing support for these core constructs we can then automatically translate higher level constructs expressed in terms of these constructs e.g. join patterns, multi-way rendezvous and data-parallel programs.

The designs presented in this paper were developed using an off the shelf software integrated development environment (Visual Studio 2005) and it was particularly productive to be able to use existing debuggers and code analysis tools. By leaveraging an existing design flow and existing language with extension mechanisms like custom attributes we were able to avoid some of the issues that face other approaches which are sometimes limited by their development tools.

Our approach complements the existing research on the automatic synthesis of sequential programs (e.g. ROCCC and SPARK) as well as work on synthesizing sequential programs extended with domain specific concurrency constructs (e.g. Handel-C). By identifying a valuable point in the design space i.e. parallel programs written using conventional concurrency constructs in an existing language and framework we hope to provide a more accessible route reconfigurable computing technology for mainstream programmers. The advent of many-core processors will require programmers to write parallel programs anyway so it is interesting to consider whether these parallel programs can also model other kinds of parallel processing structures like FPGAs and GPUs.

Our initial experimental work suggests that this is a viable approach which can be nicely coupled with vendor based synthesis tools to provide a powerful way to express digital circuits as parallel programs. Furthermore, we have shown how it is possible to layer higher level concurrency idioms on top of the base Kiwi layer by defining parallel circuit combinators written in F#.

References

[1] B. A. Buyukkurt, Z. Guo, and W. Najjar. Impact of loop unrolling on throughput, area and clock frequency in ROCCC: C to VHDL compiler for FPGAs. *Int. Workshop On Applied Reconfigurable Computing*, March 2006.

[2] M. Gokhale, J. M. Stone, J. Arnold, and M. Kalinowski. Stream-oriented FPGA computing in the Streams-C high hevel language. *8th IEEE Symposium on Field-Programmable Custom Computing Machines*, 2000.

[3] Rajesh K. Gupta and Stan Y. Liao. Using a programming language for digital system design. *IEEE Design and Test of Computers*, 14, April 1997.

[4] Sumit Gupta, Nikil D. Dutt, Rajesh K. Gupta, and Alex Nicolau. SPARK: A high-level synthesis framework for applying parallelizing compiler transformations. *International Conference on VLSI Design*, January 2003.

[5] Celoxica Inc. Handel-C language overview. *http://www.celoxica.com*, 2004.

[6] Monia S. Lam and Robert P. Wilson. Limits of control flow on parallelism. *The 19th Annual International Symposium on Computer Architecture*, May 1992.

[7] W. A. Najjar, A. P. W. Bohm, B. A. Draper, J. Hammes, R. Rinker, J. R. Beveridge, M. Chawathe, and C. Ross. High-level language abstraction for reconfigurable computing. *IEEE Computer*, 36(8), 2003.

[8] Y. D. Yankova, G.K. Kuzmanov, K.L.M. Bertels, G. N. Gaydadjiev, Y. Lu, and S. Vassiliadis. DWARV: Delftworkbench automated reconfigurable VHDL generator. *17th International Conference on Field Programmable Logic and Applications*, August 2007.

Efficient Circuit Design with uDL

Steve Hoover
Intel
Massachusetts Microprocessor Design Center
Hudson, MA 01749
steve.hoover@intel.com

Abstract

uDL (Microarchitecture Design Language) is an RTL-replacement concept language that separates the cycle-accurate behavioral description of a design from the physical details necessary to implement it. Physical details such as pipeline staging, physical partitioning, and signal encoding augment the behavioral model and do not affect behavior. Furthermore, a refinement mechanism provides a generic mechanism for abstraction. Equivalence of refinements is verified formally or via shadow simulation.

The behavioral model is the target of dynamic verification. It is both simpler than RTL and less subject to physically motivated design changes. uDL designs can be more-easily leveraged and retargeted to new technologies as the high-level design is not obscured by the physical detail, and that detail can easily be changed without introducing bugs. A uDL model has information about the flow and validity of data. This meta-information provides opportunities that can fuel the next generation of design automation tools.

1. Introduction

The design of a large, high-performance integrated circuit, such as a microprocessor chip, can employ hundreds of people for several years. The investment, especially in verification, continues to grow from one generation to the next, as designs become more complex, and as fabrication technologies advance, allowing more logic to fit onto a single microchip. To combat this trend design teams increasingly reuse and leverage previous designs and integrate third-party IP. However, retargeting these designs for new process technologies and new constraints, especially frequency, can also involve significant effort, including re-verification.

Many high-level modeling languages and methodologies have been introduced by academia, from EDA companies, and within design teams over the years in an attempt to bring the design focus to a level above that of RTL. Previous attempts have run into several technical obstacles:

- Those such as Esterel [1] and, to a lesser extent, BlueSpec [2], which rely on synthesizing designs from higher-level models, sacrifice control over physical details.

- Performance modeling and design space exploration languages, like ADL [9] and Asim [11], represent an additional model and an additional language beyond RTL in the design process, which imposes a significant development and maintenance cost. They also do not provide a verification path to RTL and thus do not eliminate verification effort.
- Models requiring cycle-accuracy to validate versus RTL must incorporate complexity as physical details begin to impact behavior.

Non-technical obstacles exist as well. Academic pursuits have generally failed to integrate into complex real-world environments, and endeavors motivated from within design teams have been constrained by project deliverables. The Microarchitecture Design Language (uDL) project is an attempt to encapsulate ideas motivated by real-world experience and share them with the industry. uDL is an experimental RTL-replacement language concept that introduces several mechanisms for abstraction in a manner that reduces design (and redesign) effort.

The rest of the paper covers uDL as follows. Section 2 describes the methodology enabled by uDL. Section 3 describes the mechanisms for mapping the abstract model to physical details. A "refinement" mechanism is described in Section 4. Section 5 describes the uDL data flow model and its benefits. Treatment of reset and other mode transitions is discussed in Section 6. The applicability of uDL to performance modeling is discussed in Section 7. And Section 8 provides a summary.

2. uDL Methodology

uDL addresses the technical challenges identified above. Rather than introducing an additional model and language, it replaces RTL, providing the same level of detail. It does so in two distinct (though interwoven) parts: the **behavioral model**, which, though cycle-accurate, is void of physical detail, and the **mappings** and **refinements** from that model to the physical details needed to feed downstream tools and flows. This is pictured in Figure 1. Mappings are correct by construction (with a path to verify this assumption through simulation) and refinements are verified individually, formally or in shadow simulation.

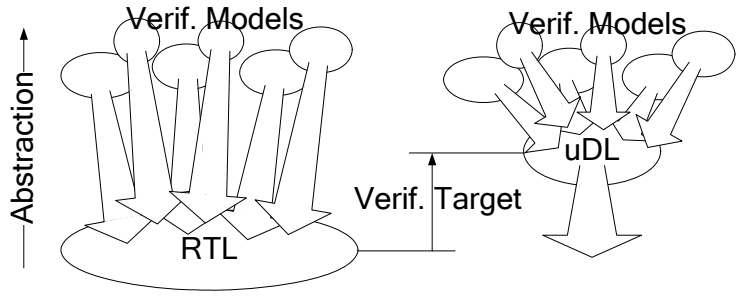


Figure 1: uDL Methodology

Total effort is reduced in several dimensions:

- The behavioral model is the target of verification, and this model is more abstract than RTL, so the verification effort is reduced.
- The entire uDL model (behavioral model plus mappings) tends to be about half the size of RTL, based on our exploration, and provide the same information, so the RTL modeling effort is reduced.

- A plethora of verification models map into the RTL: transactors, reference models, coverage monitors, visualization tools, protocol models, instruction set models, and more. The mappings can, at times, be more complex than the verification model themselves. With uDL, these mappings have less "distance" to cover vs. RTL, so they are simpler. More importantly, they are subject to far less churn as they are isolated from most physically-motivated changes.
- Since most physically-motivated changes do not impact the behavioral model, they cannot introduce bugs, do not require regression testing, and can be incorporated much faster.
- The design is easier to understand as the physical details can easily be identified and filtered.
- Designs are much more easily leveraged due to their ease of understanding and the ability to retarget the design through uDL mapping capabilities.

3. Mappings

Design elements which are absent from the behavioral model include: cycle (and phase) timing (pipe stages), physical partitioning, and signal encodings. Each of these is provided via mappings, which provide the physical detail while preserving cycle-accurate functionality. As illustrated in Figure 2, every element in the design, such as an "opcode" variable, is provided these three properties through mappings.

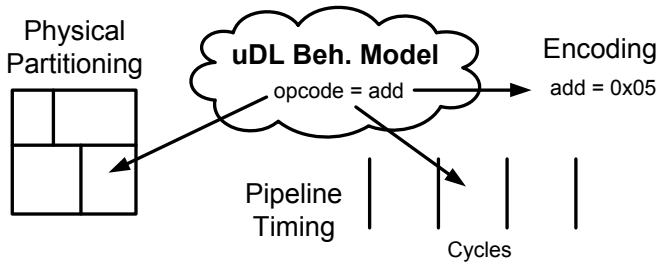


Figure 2: Forms of uDL Mappings

While Lava [3] and Wired [4] introduce mechanisms for providing data-path logic with physical layout properties, uDL focuses on mappings which provide the same information that can be found in RTL and does not necessitate changes to implementation methodology. uDL does provide a generic attribute mechanism that can propagate arbitrary attributes to downstream implementation flows.

Time Abstraction

Clocked logic, especially in high-performance designs, is generally thought of as belonging to a pipeline. The pipeline performs a function, and that function is distributed in time amongst the stages of the pipeline, so that every stage contains a depth of logic which can be evaluated within the period of the clock. This division is a physical detail. In uDL, the behavioral model describes the function of the pipeline, and each behavioral statement is bucketed to a pipeline stage as a physical mapping.

This is straight forward for pipelines which simply perform a function based on inputs and output results some time later. Pipelined logic, however, interacts with state, contains feedback and feed-forward paths, and feeds into and out from other pipelines. It is possible to describe timing-abstract functionality of these situations. In fact, it is

possible to describe arbitrary logic in a timing-abstract way, while still preserving overall functionality that is cycle-accurate. The key is to define timing relationships between pipelines (and feedback/feed-forward within a pipeline) with relative alignment. Figure 3.A shows an example containing stall logic with two feedback paths with relative cycle deltas of 1 and 2, and an input into another pipeline with a particular alignment of the pipelines. After a timing fix (Figure 3.B) to alleviate broken input timing, the overall function is preserved because the logic functions and the relative deltas remain unchanged.

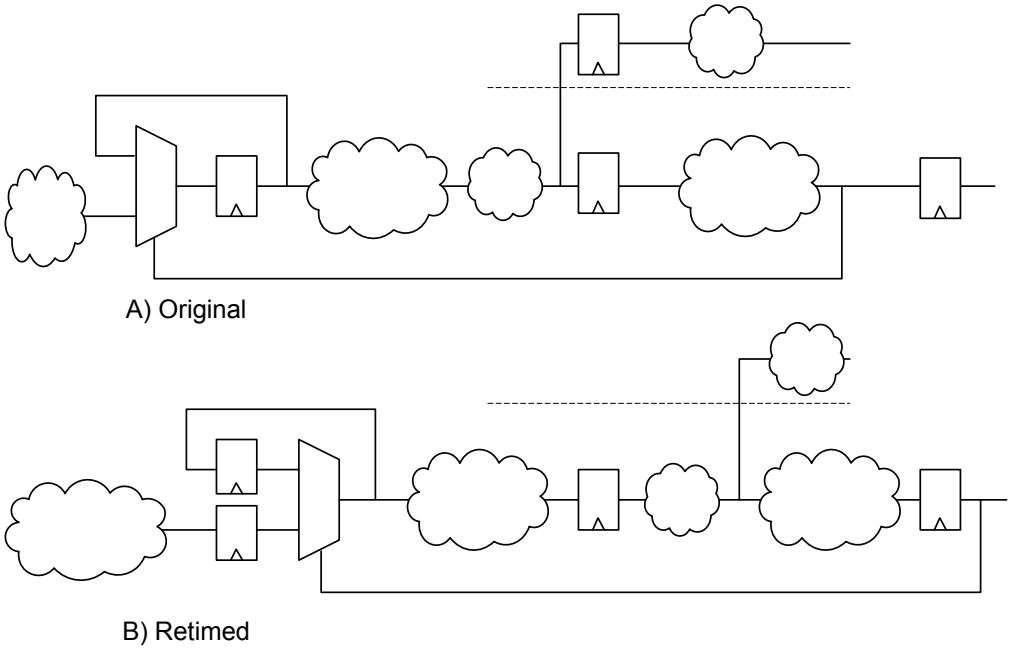


Figure 3: Example of Logic Retiming

Stage bucketing is entirely absent from the uDL behavioral model, and exists only as a physical mapping. Sequential elements are implicit, though they can be referenced to provide physical information such as attaching arbitrary attributes for downstream tools or bucketing them to physical partitions.

Physical Partitioning

A design is partitioned into a hierarchy of subcomponents to enable a divide-and-conquer approach to physical implementation. These subcomponents are often restricted to rectangles for ease of integration. So the boundaries are physically, not behaviorally, motivated. uDL creates a complete separation between the behavioral hierarchy and the physical hierarchy (though often the behavioral partition provides a general guideline for physical partitioning). As with pipeline stages, every behavioral expression is also bucketed to physical hierarchy. Logic can be replicated by bucketing it to multiple places in the physical hierarchy.

Encoding Abstraction

The data type of a uDL variable is defined as the set of values the variable may have (its range). These values may be integers or tokens. An encoding mapping defines the encodings of these values to physical signals. It is up to uDL compilation tools to generate the logic for each uDL expression appropriate for the encodings of its inputs and outputs. Logic synthesis tools tend to do a good job optimizing the type of combinational logic that would be generated to satisfy these encodings in most situations. As an example, an addition of two one-hot values becomes a shift operation, and it is up to the tool flow to generate an optimal shift. (If the tool flow does not generate acceptable results, the refinement mechanism, described in Section 4, can be used to provide further detail.)

This form of abstraction simplifies the behavioral model in several ways. For example, functions like encoders and decoders need not be present in the behavioral model. As another example, experiments can be done with state encodings for state machines to meet timing requirements without impacting the behavioral model.

Each expression understands its range based on the ranges of its inputs. So a variable's type can be implied. If an explicit range is specified, it can imply an assertion if the specification is a subset of what is implied. The assertion can be checked formally, or in simulation. A fair number of assertions seen in today's RTL are unnecessary or are simpler to capture in uDL, and many assertions that would be implicit in uDL are never coded in today's RTL due to the tedious and time-consuming nature of doing so. Examples of such assertions might be: checking that the 4-bit read index into a 10 entry array is less than 10; or checking that the array's read word line is one-hot (the read index and the read word line could actually be the same variable in uDL, differentiated only through physical mappings).

4. Refinements

The above mapping techniques will sometimes be capable of providing the level of physical detail necessary to implement a desired function. In other cases they will be insufficient, or awkward. In such cases, a generic refinement mechanism can be used. Logic requiring a more detailed representation can be bucketed (similar to pipeline stage or physical partition bucketing) for refinement. An alternate representation of the logic can be provided for use in implementation, while the behavioral model serves as the functional verification target. Equivalence of the two models is verified either via formal equivalence verification or via simulation where the refinement is built as a shadow model.

Refinements can be nested. Because the mechanism allows logic to be isolated and incrementally refined with nested refinements, each refinement step is generally small in scope, and it should generally be possible to verify refinements formally.

The utility of refinements can be broken into two categories. First, are scenarios where a block of logic is implemented very differently from its simplest behavioral expression. For example, a large first-in-first-out queue might be implemented using two register files (broken apart due to size) with a bypass path. Second, are scenarios where the pipeline stage or physical partitioning mappings must be applied to part of a single

expression. For example, an adder might be distributed across multiple cycles or physical partitions.

More generally, the refinement mechanism is an alternative-representation mechanism, where compilation options can select among the alternatives. It could be used to explore behaviorally-distinct design alternatives. It could also be used to provide abstraction above cycle-accuracy. Though not explored in our research, support for non-deterministic behavior is desirable. This can be introduced through free variables. An arbiter, for example, may provide fairness, but not correctness, and the arbitration choice can be left free to convey this. BlueSpec [2] allows the compiler to make such decisions. Various heuristics could be applied to the choice in simulation. A corresponding cycle-accurate refinement can provide true behavior and formal tools must be capable of proving conformance. The refinement mechanism could also provide performance-approximate models for estimating performance, as discussed in Section 7.

5. Data Flow

While RTL is purely an expression of the logic that implements a design, uDL has a higher-level understanding of the data and its flow through the pipelines of the design. [5] and [6] have shown in different ways that knowledge about pipelines and dataflow can be beneficial to formal analysis. BlueSpec [2] has shown that it can avoid bugs and enable synthesis of data flow control logic.

In uDL's data flow model, all logic is retimable and belongs to a pipeline (possibly a default and/or degenerate one). A pipeline may contain multiple partitions of logic, called "parcel logic", that process parcels (machine instructions or packets for example), generally one per cycle. Each parcel (or sub-parcel) has an independent indication of whether it is valid. In fact this is the real distinction among parcels/sub-parcels. A Boolean variable called "valid" can be assigned to define the valid condition for the parcel/sub-parcel. If unassigned, it is always considered valid. The valid variable does not correspond to a physical signal unless it is consumed.

Parcels can traverse multiple pipelines. For example, an instruction can be enqueued as the output of a fetch pipeline and read from the queue as the input to an issue pipeline. Parcel logic in this case, can be moved across the queue (if it has no interaction with other parcels), affecting the state values that are captured in the queue, similar to the way moving logic across pipeline stages affects the sequential elements that stage the parcel.

The data flow model, and knowledge of validity in particular has several potential benefits, including: implicit checking, safer and easier use of clock gating for power savings, improved visualization, and improved simulation speed, each described below.

Implicit checking: Logic corresponding to a valid parcel may not consume a variable of an invalid parcel. This implies a fair number of assertions and ensures that the values held by signals corresponding to invalid parcels are truly dont-care states. It also simplifies the coding of assertions as they need only apply for valid parcels. In particular, today's assertions must often be conditioned off during reset and for invalid parcels. Conditioning based on reset is unnecessary in uDL as there is no simulation of valid parcels during reset. (See Section 6.)

Clock gating: Clock gating is an important technique for saving power. Clock gating avoids driving a clock pulse to sequential elements in cycles where data does not need to propagate (or must be held). This is the case, when the values do not belong to valid parcels. Based on timing constraints of generating the gating function, the function that is implemented could be a subset of the cases that could be gated. The choice of gating function is restricted by physical constraints. Since the uDL model understands legal clock gating functions, it would be possible for tools to either synthesize clock gating logic within the constraints of timing (and area, and power), or to verify the choice of gating functions in hand-drawn schematics or uDL attributes.

Visualization: Significant time is spent in the debug process trying to understand simulations. Waveform viewers are one common type of visualization tool. Finding an activity of interest in a waveform view can be a bit like finding a needle in a haystack. A viewer that understands the uDL data flow model could filter out a significant amount of data based on knowledge of validity. It could represent data as meaningful integer and mnemonic values, and could group these as parcels. And this can all be automatic. Attempts to represent a design at this level today are highly customized and require a significant development and maintenance investment.

Simulation speed: All forms of abstraction offered by uDL represent opportunities for faster simulation. Refinements eliminate detail, as does the absence of physical partitioning. Encodings can be chosen by a compiler to optimize simulation speed, as can stage bucketing. Furthermore stage bucketing can be an enabler for multithreaded simulation. One of the challenges with partitioning logic for multithreaded simulation is the tight communication between partitions. Careful stage bucketing can potentially eliminate intra-cycle dependencies, back-to-back cycle dependencies, and so on. Additionally, a parcel can be modeled as a single data structure, and a pointer to it can be tracked through its pipeline(s) rather than staging the data of the parcel. Invalid parcels need not be allocated or simulated at all. Parcel logic can be grouped for both temporal and spatial locality. On a cautionary note, these opportunities also introduce irregularity, which results in branch mispredictions and cache misses, so these thoughts represent a careful balancing act for the simulator to target the simulation platform.

6. Mode Transitions

Much of the logic in a high-performance design is there for “special” modes of operation, in particular, to support reset, test, and debug capabilities. Retaining cycle accuracy in a model that includes these features can limit the ability to abstract away details of the primary functionality. Eliminating the need to model these modes in uDL will significantly improve the ability to model abstractly. Furthermore, this logic tends to be motivated heavily by physical constraints, so it is appropriate for the schematics to be their definition or for the logic to be synthesized from an understanding of the mode’s requirements.

Functional verification of these modes would have to be done on the gate-level model, where simulation is significantly slower. Fortunately, special modes do not generally require the breadth of coverage that is required of the primary functionality. And it is generally easier to divide the logic into isolated pieces. Verification must be

done not only of the function of the mode, but also of the transitions into and out of the mode.

At a minimum, it would be valuable to avoid modeling SCAN and reset logic in uDL. Various methodologies have been employed to date which eliminate SCAN from RTL [10]. For reset, the uDL model would understand the various levels of reset that are supported and the conditions that would trigger them. Each state would have a reset value and a reset level at which the reset value would be applied. The specific logic that assigns the reset values would not be specified. It is reasonable to imagine flows to synthesize such logic. It is also reasonable to imagine flows to verify the behavior in three-state, gate-level simulation. Each level of reset can be validated in simulation by considering the machine to be in an unknown state, simulating the reset flow, and checking that the correct reset state values were applied.

7. Performance Modeling

Performance modeling and design space exploration is generally done today in general-purpose programming languages like C++, often augmented with a library specifically for performance modeling. These languages are more appropriate for performance modeling than RTL as RTL does not support run-time parameterization, easy model configurability, and data collection. So the performance model is a separate model to develop, maintain, and verify against the RTL. This is a significant investment. Generally, there is no formal linkage between the RTL and a performance model. There is always question as to how closely the performance model matches RTL performance. For these reasons, RTL is often used in addition to performance modeling to evaluate the performance of a design. However, by the time performance feedback from RTL models is available it is costly to rework the design to address issues.

A few capabilities have been considered for uDL in support of using uDL for performance modeling. The refinement mechanism is one such capability for easy modeling and configuring of alternatives. Run-time parameterization is another. The ability to insert pipeline stages (at compile-time or runtime) is an important form of parameterization for studying the impact of latency on performance.

While there are benefits to targeted performance modeling solutions, the integration of performance modeling and “RTL” development would eliminate the second model and its associated costs. It would make the transition from performance modeling to design more graceful as some performance model components could be used or leveraged for design components. And it would enable easier functional verification model creation, as performance modeling components could be used to provide stimulus for designs under test. And lastly, and possibly most importantly, the performance model could now be synthesized into logic. This means it could be targeted for hardware emulation, to get dramatically improved performance, which is an area of much current research [7] [8].

8. Summary

uDL introduces several mechanisms for enabling more abstract cycle-accurate microarchitectural modeling. A separation is established between the behavioral model, which is the target of functional verification, and the details required for physical implementation of that model, with a verification path between the two. Pipeline timing, physical partitioning, and encoding are uDL's three main forms of correct-by-construction mappings to physical detail. Refinement adds a generic mechanism to abstract a design in ways that are not enabled by the other mechanisms, and these are verified either formally or in simulation.

Furthermore knowledge of the flow of data provides implicit assertion checking (as does the encoding mapping), and it is an enabler for future improvements in visualization, logic synthesis (especially of clock-gating and reset logic), and simulation speeds. There is a need for new innovation which will enable "special mode" logic to be absent from the abstract model. And lastly, uDL presents opportunities for unifying performance models and hardware models, simplifying the design process and enabling hardware emulation of performance models.

By bridging the gap between abstract modeling (at a cycle-accurate level) and implementation details, uDL presents a viable direction for the evolution of microarchitecture design over the next several decades. By enabling abstraction and stability in verification target models, it can help to tame the growth in verification effort that threatens the industry. It improves the ability to leverage designs and their verification collateral through multiple technology generations, which is of extreme importance for today's system-on-chip designs.

Acknowledgements

uDL brings together ideas found in many past efforts within Intel. Members of the working group exploring uDL were: Eugene Earlie, Sergey Bogdanov, Steve Hoover, Timothy Leonard, Gabriel Bischoff, Mark Tuttle, Peter Morrison, Armaghan Naik, and Emily Shriver.

References

[1] http://www.esterel-technologies.com

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

[3] P, Bjesse, K. Claessen, M. Sheeran, and S. Singh, "Lava: Hardware Design in Haskell," International Conference on Functional Programming, September 1998.

[4] E. Axelsson, K. Claessen, and M. Sheeran, "Wired: Wire-aware circuit design," Conference on Correct Hardware Design and Verification Methods, 2005.

[5] J. Higgins and M. Aagaard, "Simplifying the Design and Automating the Verification of Pipelines with Structural Hazards," ACM Transactions on Design Automation of Electronic Systems, October 2005.

[6] J. Cortadella, M. Kishinevsky, and W. Grundmann, "Synthesis of Synchronous Elastic Architectures," Design Automation Conference 43rd ACM/IEEE, 2006.

[7] M. Pellauer and J. Emer, "HAsim: Implementing a Partitioned Performance Model on an FPGA," Proceedings of the Fifth Annual Boston Area Architecture Workshop, January, 2007.

[8] D. Chiou, H. Sunjeliwala, D. Sunwoo, J. Xu, and N. Patil. "FPGA-based Fast, Cycle-Accurate, Full-System Simulators," Tech Report Number UTFAST-2006-01, Austin, TX, 2006.

[9] P. Mishra, A. Shrivastava, and N. D. Dutt, "Architecture Description Language (ADL)-driven SoftwareToolkit Generation for Architectural Exploration of Programmable SOCs," ACM Transactions on Design Automation of Electronic Systems, July 2006.

[10] S. Barbagallo, M. Lobetti Bodoni, D. Medina, F. Corno, P. Prinetto, and M. Sonza Reorda, "Scan Insertion Criteria for Low Design Impact," Proceedings of the 14th IEEE VLSI Test Symposium, 1996.

[11] J. Emer, P. Ahuja, E. Borch, A. Klauser, C. Luk, S. Manne, S. Mukherjee, H. Patil, S. Wallace, N. Binkert, R. Espasa, and A. Juan, "Asim: A Performance Model Framework," Computer, 35(2): pp. 68-72, 2002.

Some "Real World" Problems in the Analog and Mixed Signal Domains

Kevin D Jones

Jaeha Kim, Victor Konrad

DCC 08

Rambus

Abstract

- Design and verification problems in the digital circuit space are well studied and there are many approaches to establishing correctness, both academic and industrial. However, the analog and mixed signal space is much less developed.
- More and more systems are being designed with “mixed mode” functionality. That is to say, there are digital and analog components working synergistically and, often, in complex ways with many interactions across the domains. There are few good tools available for designers working in this space, with the standard approach being based on custom schematics and directed transistor level simulation.
- Some of the problems encountered are natural extensions of the digital space and are amenable to extensions of the same formal solutions; some problems are fundamentally different and require novel approaches for verification. We believe that formal verification in this field is a nascent discipline with opportunities for fundamental contributions.
- In this talk, we will discuss the design approaches commonly used for mixed mode systems. We will present a synopsis of the way such systems are currently verified in industry and talk about some of the drawbacks of these approaches. We will give a collection of interesting “real world” examples – examples drawn from actual failures of real designs but reduced to their essential elements to make them of a suitable size for academic study – and show some of the key concerns analog/mixed mode designers have in practice. We will present digital, analog and mixed mode examples. Finally, to seed what we hope will be an on-going discussion, we will illustrate some of the novel approaches we are developing to these issues, particularly in the analog domain.

DCC 08

Slide 2

Rambus

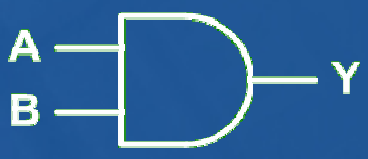
Digital and Analog

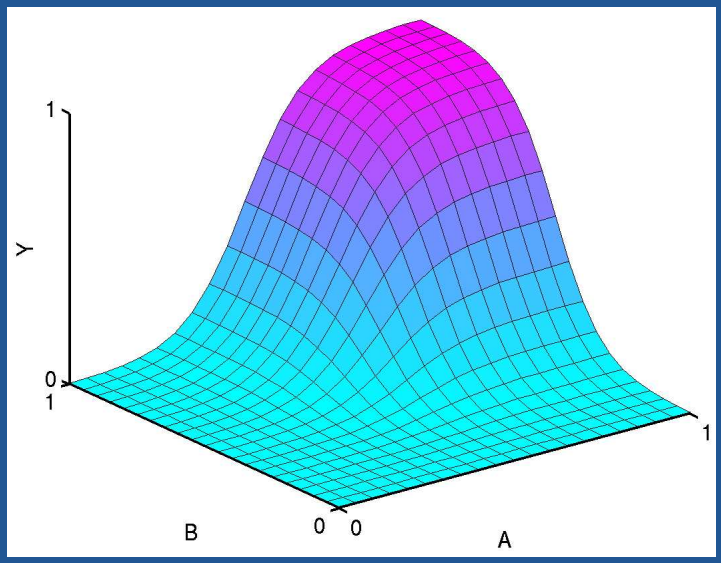
- Different worlds
- Think differently and often miscommunicate
- Occasionally forced to coexist in mixed signal designs

- I'm going to be focusing on a verification perspective
  - (and am really going to be focusing on analog rather than mixed signal since it is the required first step for most digital folk)

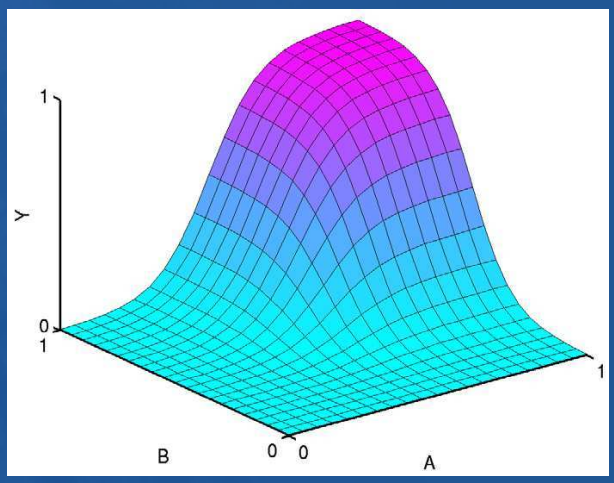
Digital Circuits

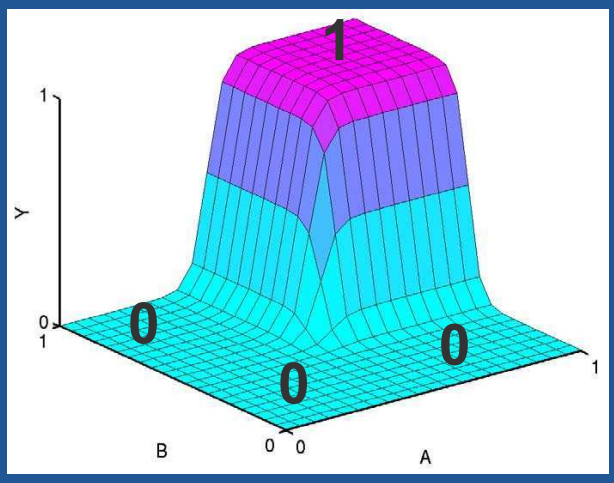
Verify this two-input AND gate





Digital Abstraction





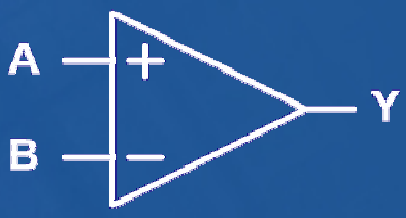
- States are discrete and countable
- Must verify the properties at all states

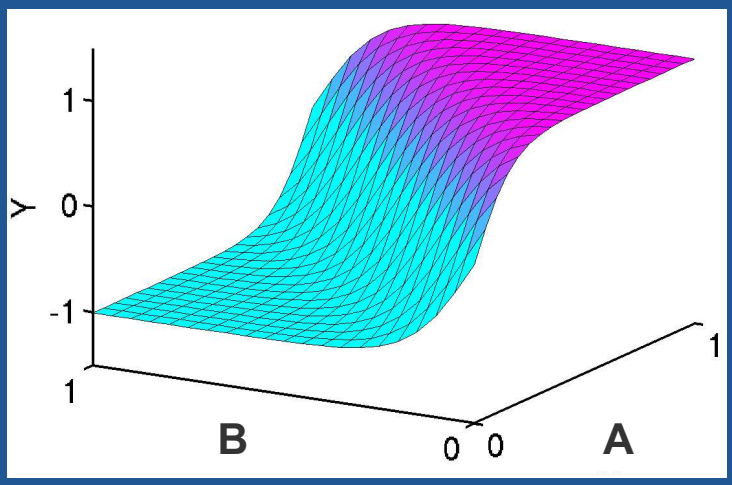
Digital Verification

- Exhaustive simulation
  - Directed pseudo-random simulation
  - Coverage based
- Formal methods
  - BDD based exhaustive state analysis
  - Symbolic simulation
- Tools: Specman, SystemVerilog, Equivalence checkers, Model checkers, Theorem Provers, …

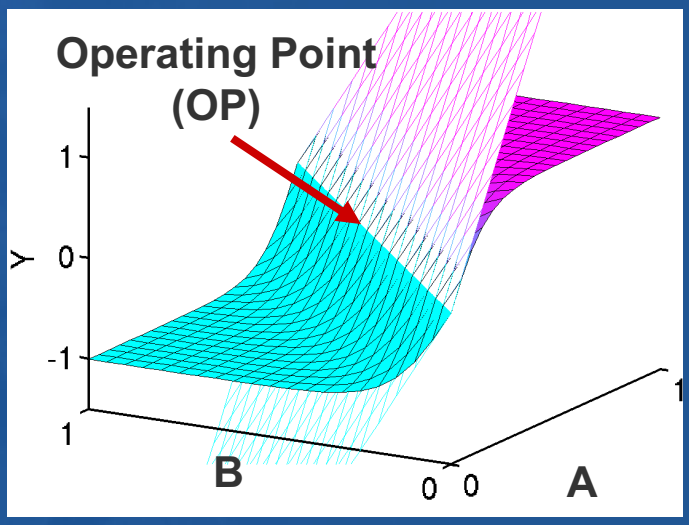
Analog Circuits

Verify this op-amp





Linear Abstraction



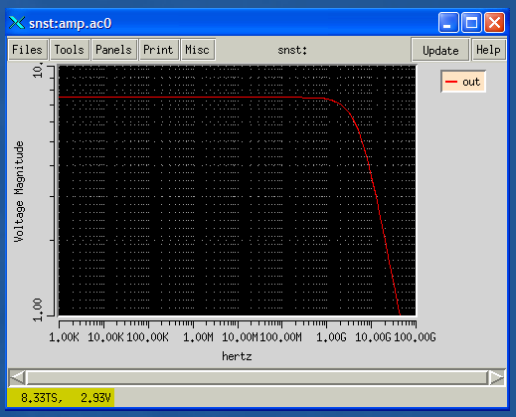
- Choose an Operating Point
- Apply small-signal perturbations to the OP (ΔA, ΔB)
- Derive the relation $\Delta Y = \alpha \cdot \Delta A + \beta \cdot \Delta B$
- *"Linearizing the system at the OP"*

- If $\alpha \cong -\beta \Rightarrow$ it's an op-amp with gain $|\alpha|$

Verification of Analog circuits

- Transistor level models
- Circuit simulator (Spice)
  - DC simulation
  - AC simulation
  - Parameter sweeping
  - Statistical variation (Monte Carlo)
- Tools: SPICE (HSPICE, PSPICE, Spectre, eldo, …)
  - Transient (time based) simulation is very expensive
  - Simulation inputs based on designers' understanding/experience/intuition

AC Analysis is a Formal Method for Analog

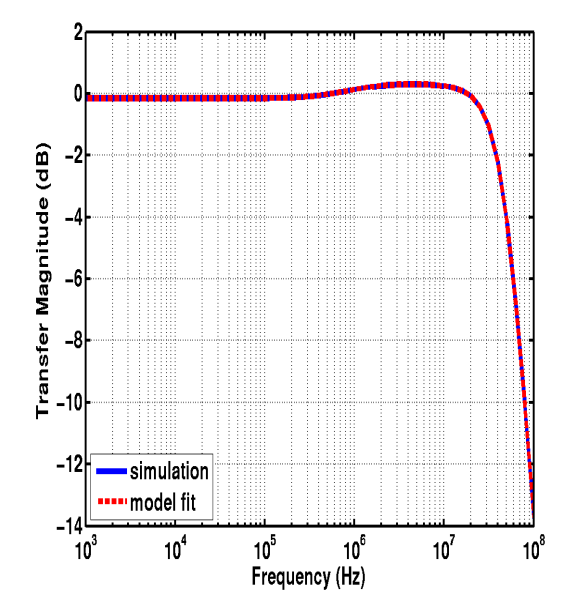


- AC analysis in SPICE provides the transfer function (TF)
- TF: a complete description of the linear system in frequency domain

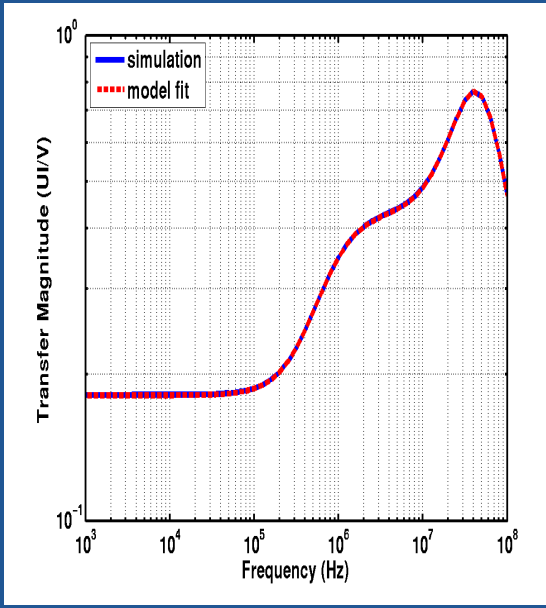
- No further need to simulate circuits in transient simulation as long as the input signals remain small
- The circuit is *formally* verified

PAC Results on PLL

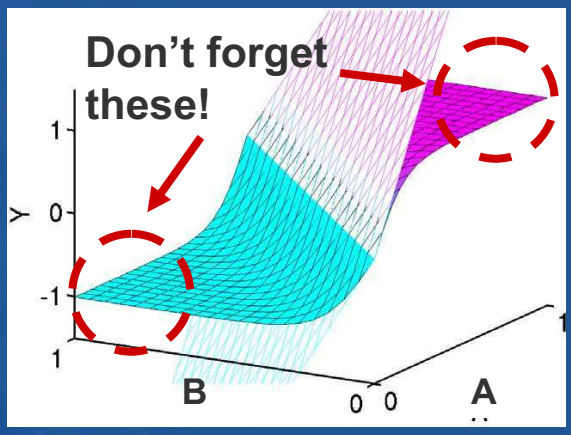
Phase Transfer



Supply Sensitivity



Limitation of Linear Analysis



- Linear methods verify only LOCAL properties
- Most circuits are nonlinear

- Must make sure that the circuit does operate at the assumed OP

Bugs in Analog/MS Systems

- Parametric bugs
  - E.g. Jitter too high
  - The usual focus of designers effort
  - Requires better models and simulation techniques for improvement
- Functional bugs
  - E.g. PLL doesn't lock!
  - Very difficult to find due to long simulation times and large spaces

Open problems

- Avoiding transient simulation
- Establishing that operating point assumptions are valid
- Establishing that all initial conditions result in correct behavior
- Dealing with non-linearity

An Example from the "Real world"

- The example is extracted from an actual design failure
  - Some issues were only found in measurement of fabricated test chips
  - The design was validated as well as any analog designs are in practice today
- The scale of the example is reduced to make it approachable from an academic perspective but all of the issues are still present

An Even Stage Ring Oscillator

- Ring oscillators are a common component used in a variety of analog applications
- The obvious design uses an odd number of inverter stages to produce oscillating "0" and "1" signals at a frequency dependent on the inverter delay
- Even (2 or 4) stage oscillators are used when quadrature clocks are useful

An even stage ring Oscillator

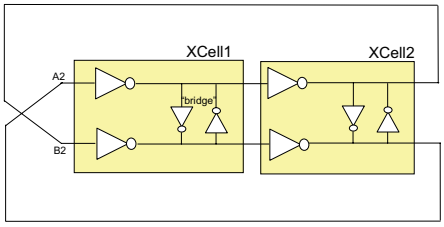


Figure 1: "XChainOfCells2"

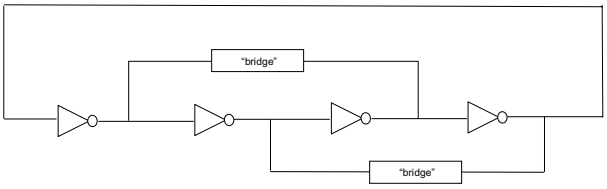


Figure 2: An alternative view

DCC 08

Slide 17

Rambus

The complete description

```
* sring - a ring oscillator with bridges
*-------------------------------------------------------------
* Libraries, parameters and models
*-------------------------------------------------------------
.protect
*.lib '/user/cad/process/tsmc65/model/spice/g+/v1.0/local/fets.lib' TT
.lib ./cln90g_lk.l  TT
.unprotect
.model pch pmos
.model nch nmos
*-------------------------------------------------------------
* Operating point
*-------------------------------------------------------------
.temp 30
.param supplyVoltage=1.5V
*-------------------------------------------------------------
* Initial conditions and stimuli
*-------------------------------------------------------------
.IC V(A2)='supplyVoltage*1.'
*.IC V(B2)='supplyVoltage/2.'
Vdd vdd gnd supplyVoltage DC
*-------------------------------------------------------------
* Simulation controls
*-------------------------------------------------------------
.tran 25ps 5000ps   UIC
.option post
.plot  v(A2))
```

```
*-------------------------------------------------------------
* Simulation netlist
*-------------------------------------------------------------
.
.param lp=1.0u  ln=1.0u
.param  wnbridge= 2u wpbridge=4u
* Circuit ceases to oscillate above this ratio
*.param ratioChainToBridge = 1.972
*
*Circuit ceases to oscillate below  this ratio
*.param ratioChainToBridge = 0.31
*
.param wnchain = 'wnbridge*ratioChainToBridge'
.param wpchain = 'wpbridge*ratioChainToBridge'
.global vdd gnd
.subckt INV  in out psource nsource
            Mpullup    out in psource vdd pch l=lp w=wp
            Mpulldown  out in nsource gnd nch l=ln w=wn
.ends  INV
.subckt  CELL inp inm outp outm
            Xinv1  inp outm vdd gnd  INV wp='wpchain' wn='wnchain'
            Xinv2  inm outp vdd gnd  INV wp='wpchain' wn='wnchain'
            Xinv3  outp outm vdd gnd INV wp='wpbridge' wn='wnbridge'
            Xinv4  outm outp vdd gnd INV wp='wpbridge' wn='wnbridge'
.ends CELL
.subckt CHAINOFCELLS2   ina inb outa outb
            XCell1  ina inb oa1 ob1   CELL
            XCell2  oa1 ob1 outa outb  CELL
.ends CHAINOFCELLS2
XChainOfCells2  A2 B2 B2 A2  CHAINOFCELLS2
.end sring
```

DCC 08

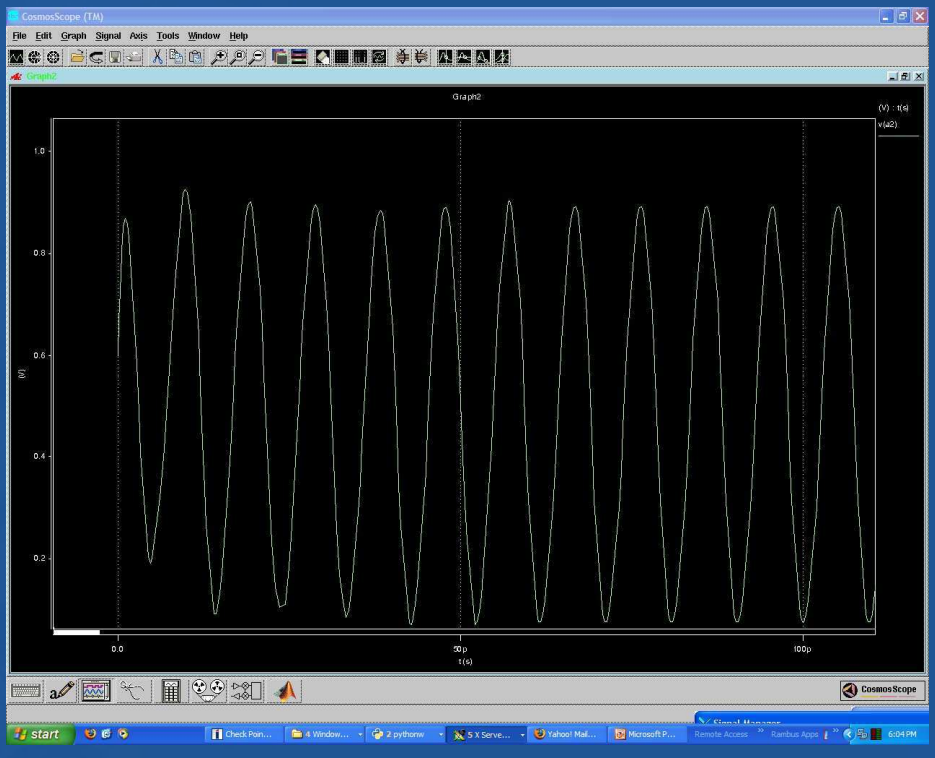
Slide 18

Rambus

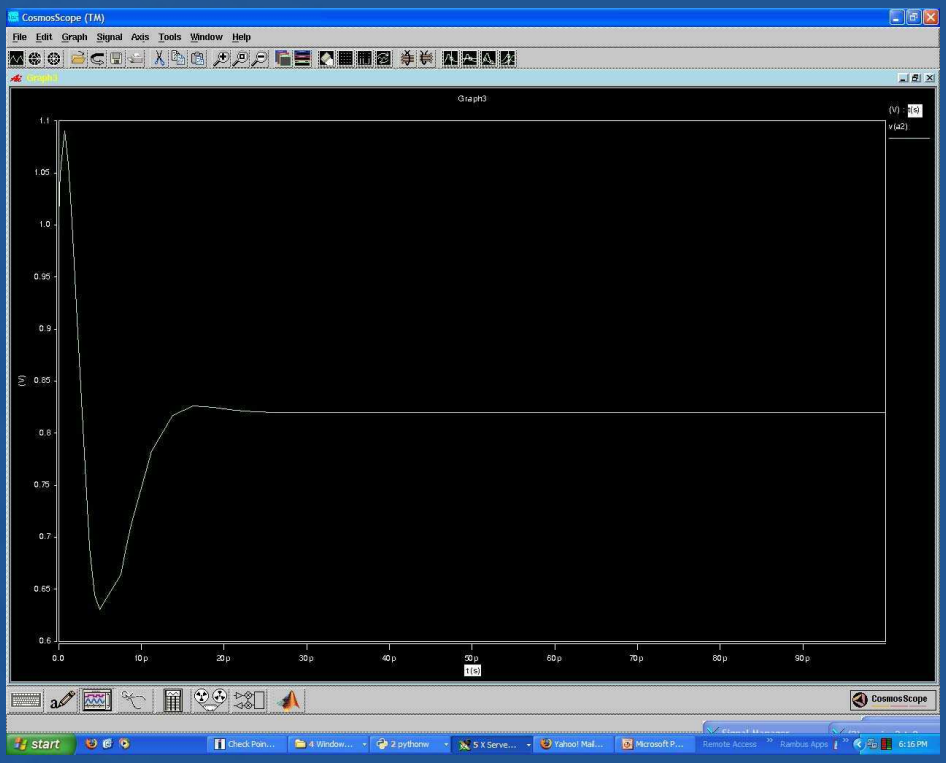
Challenge for Formal Methods

- The “obvious” digital abstraction doesn’t hold
- It has interesting failure modes
  - It is very sensitive to the exact sizing of the ring and bridge transistors
  - It has sensitivities to initial conditions for some sizes
- The challenge: Show that this circuit oscillates for all initial conditions for some sizing
- Extra credit: Show the sizing ratios regions for the ring and bridge transistors

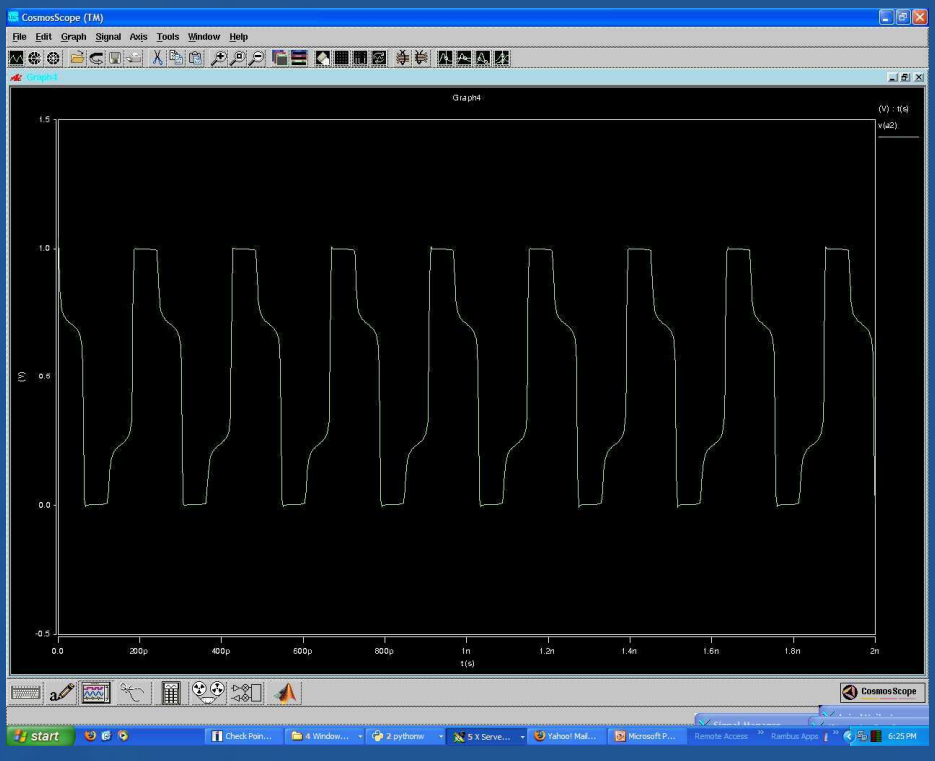
Good



Bad


CosmosScope (TM)
File Edit Graph Signal Axis Tools Window Help
Graph3
(V) : t(s)
v(a2)
1.1
1.05
1.0
0.95
0.9
0.85
0.8
0.75
0.7
0.65
0.6
(V)
0.0
10p
20p
30p
40p
50p
60p
70p
80p
90p
t(s)
CosmosScope
start
6:16 PM

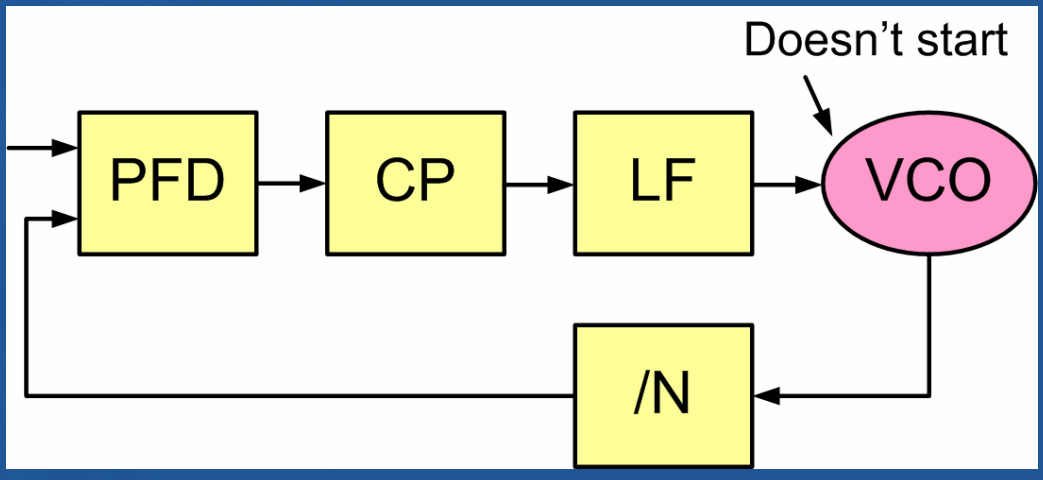
Ugly


CosmosScope (TM)
File Edit Graph Signal Axis Tools Window Help
Graph4
(V) : t(s)
v(a2)
1.5
1.0
0.6
0.0
-0.5
(V)
0.0
200p
400p
600p
800p
1n
1.2n
1.4n
1.6n
1.8n
2n
t(s)
CosmosScope
start
6:25 PM

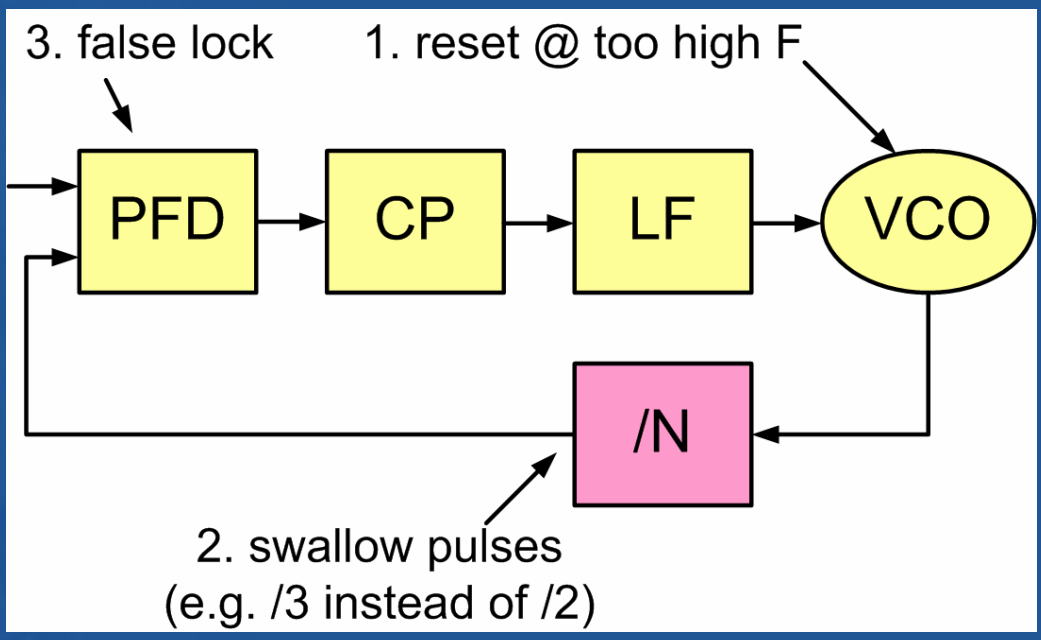
Moving up the food chain

- Phase Locked Loops are critical components of most high speed PHYs
  - They are made up of subcomponents
    - Phase detector
    - Charge pump
    - Linear filter
    - Voltage Controlled Oscillator
    - Divider
- There are many possible failure modes
- Simulating the locking behavior of a PLL at the transistor level in the time domain is very expensive

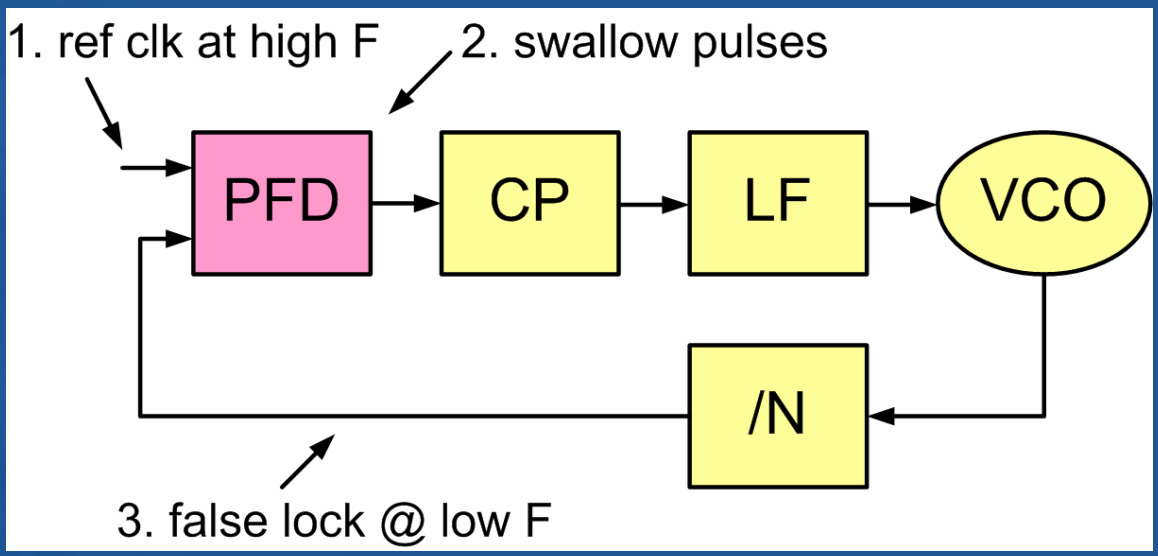
PLL locking bugs



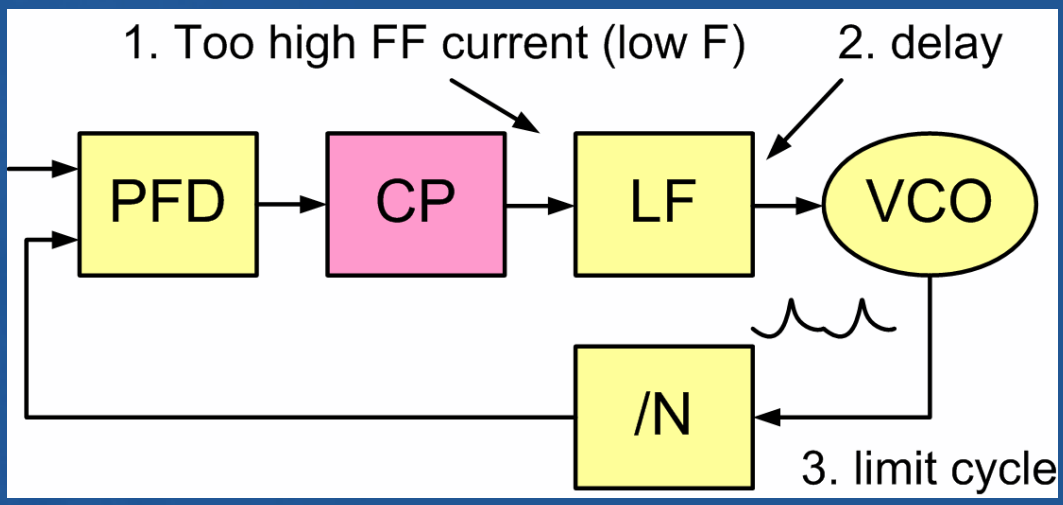
PLL Locking Bugs (II)



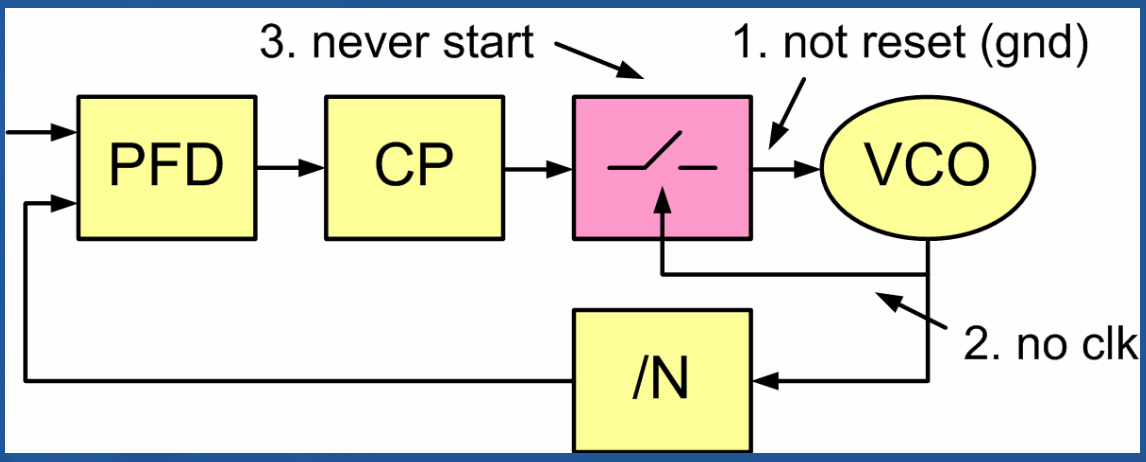
PLL Locking Bugs (III)



PLL Locking Bugs (IV)



PLL Locking Bugs (V)



A Verification Nightmare

- Individual components can be wrong
- Individual components can be fine but assumptions between components can be wrong
- Most of these issues are not visible if we assume "correct initialization" i.e. we start simulation from a locked state, as we do for most parametric simulation
- It takes a very (very, very, …) long time to simulate, using sufficiently accurate models, from any arbitrary initial state to reach lock
- Bugs like these make it through to production

Novel thinking for analog verification

- Analog verification is an underdeveloped field
- We've been looking at a number of interesting ideas that hint at some promising directions of research
- We'll present one example

Avoiding non-linear domains

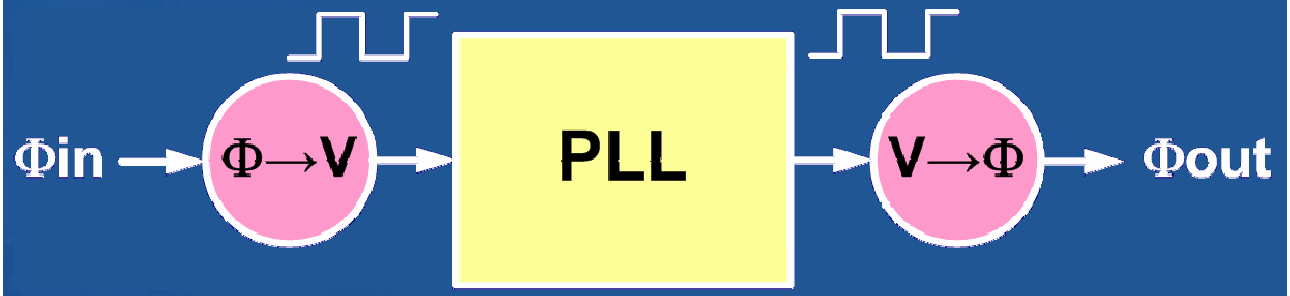
- The "formal" techniques for analog work very well in linear domains
- Many (most) circuits are not linear in the voltage or current domains
  - PLLs are obviously very non-linear in V
- All the simulator AC analysis tools work in V or I
- This seems to imply they are not useful for most circuits

Variable Domain Transformation

- Most circuits are not linear in voltage or current

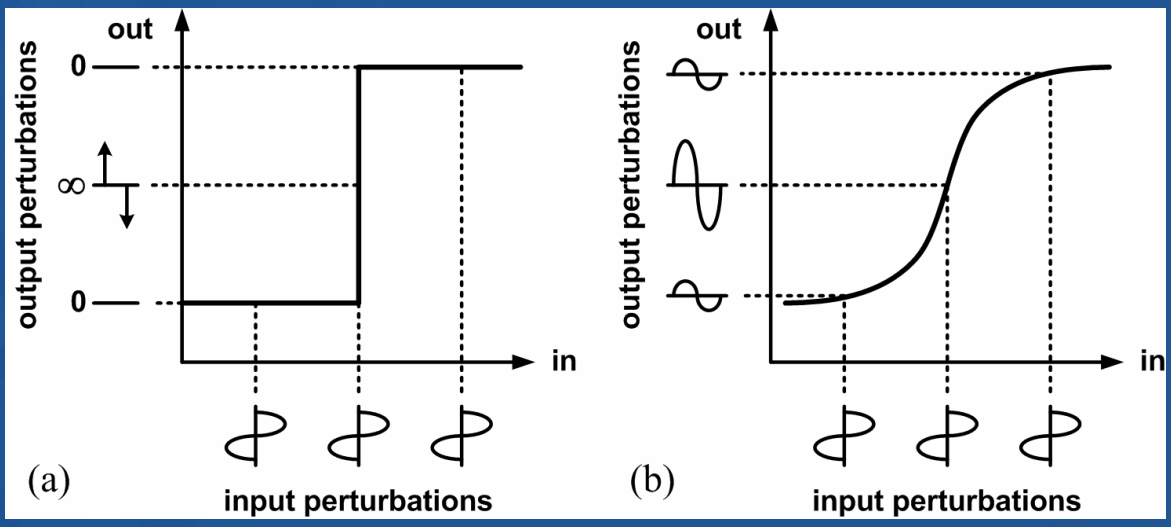
•but are linear in some domain:

•phase, frequency, delay, duty-cycle, …



Domain Translators

- V→Φ ≈ Phase Detector and
  Φ→V ≈ Phase Mixer
- Must propagate perturbations correctly



Domain transformation for verification

- Use domain translators to map the linear domain to V or I
  - Verilog-A is a good vehicle for developing translators
- Perform analysis in this domain using AC analysis tools
- Use inverse translator to map back to original linear domain for results
- All the benefits of linearity together with all the benefits of V/I AC simulation techniques

Conclusions and future work

- Analog and digital are different
  - Different mindsets, different tools, different problems
- There are problems in the analog space that are really looking for solutions
  - Different points of view yield valuable approaches
- We can provide "realistic" examples to interested parties
  - Some are small, representative and tractable
  - If any one really wants it, we have a software → digital → analog → physics problem (ms → ns → ps → fs) in a 20GB SERDES system

If your interest has been piqued

- kdj@rambus.com
  Project Director
- jaehak@rambus.com
  Circuit design and simulation
- vkonrad@rambus.com
  Analog verification

Some further reading

- General analog design
  - Gray & Meyer: Analysis and Design of Analog integrated Circuits, Wiley.
  - B. Razavi: Design of Analog CMOS Integrated Circuits, McGrawHill
  - Thomas H. Lee: The Design of CMOS Radio-Frequency Integrated Circuits, Cambridge University Press
- Verification issues for Analog/MS
  - Thomas Sheffler, Kathryn Mossawir, Kevin Jones: PHY Verification - What's Missing?, DVCon 2007
- More information on Domain Transformation
  - Jaeha Kim, Kevin D. Jones, Mark A. Horowitz: Variable domain transformation for linear PAC analysis of mixed-signal systems. ICCAD 2007: 887-894

Defining Elastic Circuits with Negative Delays

Sava Krstić[1], Jordi Cortadella[2], Mike Kishinevsky[1], and John O'Leary[1]

[1] Strategic CAD Labs, Intel Corporation, Hillsboro, Oregon, USA
[2] Universitat Politècnica de Catalunya, Barcelona, Spain

An elastic circuit can take an arbitrarily long time to compute its results and it can wait arbitrarily long for cooperating circuits to produce its inputs and to consume its outputs. For example, the streams $X = \mathbf{1}\square\mathbf{00}\square\square\mathbf{1}\square\cdots$ and $Y = \square\square\mathbf{0}\square\mathbf{1}\square\square\mathbf{1}\square\mathbf{0}\cdots$ could be an observed input/output behavior of an elastic inverter. The transfer of data on the input channel occurs at cycles 1,3,4,7,...; the absence of transfer is shown as the bubble symbol $\square$. When we ignore the bubbles, we see a behavior of an ordinary inverter: $X' = \mathbf{1001}\cdots$ and $Y' = \mathbf{0110}\cdots$.

Elastic circuits promise novel methods for microarchitectural design that can exploit variable latencies, while still employing standard synchronous design tools and methods [1]. The Intel project SELF (Synchronous Elastic Flow) [3, 2] demonstrates the industry interest. In SELF, every elastic circuit $\mathcal{E}$ is associated with an ordinary (non-elastic) system $\mathcal{S}$ in the "bubble removal" sense indicated in the inverter example. For each wire X of $\mathcal{S}$, there is a *channel* $\langle D_X, V_X, S_X\rangle$ in $\mathcal{E}$ consisting of the *data* wire D_X, and the handshake wires V_X and S_X (*valid*, *stop*). A transfer along the channel occurs when $V_X = 1$ and $S_X = 0$, thus requiring producer-consumer cooperation.

The first formal account of elasticity was given by the *patient processes* of [1]. Our *elastic machines* [5] are a more readily applicable theoretical foundation, but even this is insufficient for modeling relevant subtler aspects, notably anti-token counterflow [2]. Here, we would like to discuss the progress we have made in devising a more general theory and the challenges we encountered.

An adequate theory needs to provide a definiton of elastic circuits that makes two fundamental results provable: (1) the *liveness theorem* that guarantees the infinite flow of tokens whenever an elastic circuit is put in an environment that offers communication on each channel infinitely often; (2) the *elastic compositionality theorem* guaranteeing that, under reasonable assumptions, if elastic circuits $\mathcal{E}_1, \ldots, \mathcal{E}_n$ implement the systems $\mathcal{S}_1, \ldots, \mathcal{S}_n$ and if $\mathcal{S}$ is the network obtained by connecting some wires of the systems $\mathcal{S}_i$, then connecting the corresponding channels (wire triples) of the elastic circuits $\mathcal{E}_i$ produces a new elastic circuit which implements $\mathcal{S}$.

As in [5], we model ordinary (non-elastic) systems as stream transformers, but now we use a more precise notion of causality: given an inputs-by-outputs integer matrix $\delta = \|\delta_{XY}\|$, we call a system δ-*causal* if the nth Y-output depends only on the first $n - \delta_{XY}$ X-inputs. In [5], we dealt only with systems satisfying $\delta_{XY} \geq 0$ ("circuits") and had a compositionality theorem based on the absence of suitably defined "combinational loops". The new setup admits systems with negative delays (like the *tail* function on streams), but we can still prove compositionality, this time based on Wadge's *cycle sum condition* [6, 4].

Now, given a δ-causal system $\mathcal{S}$, when do we say that a system $\mathcal{E}$ is an elastic implementation of $\mathcal{S}$? In addition to a simple data-correctness property, $\mathcal{E}$ also needs to satisfy liveness properties $\mathsf{G}\,(\mathsf{G}\,\mathsf{en\_in}_X \rightarrow \mathsf{F}\,\bar{S}_X)$ and $\mathsf{G}\,(\mathsf{G}\,\mathsf{en\_out}_Y \rightarrow \mathsf{F}\,V_Y)$ for all inputs X and outputs Y. Surprisingly, the enabling predicates $\mathsf{en\_in}_X$ and $\mathsf{en\_out}_Y$ are difficult to guess. By introducing the "backward delay numbers" δ_{YX}, we can express (as $\delta_{XY}+\delta_{YX}$) the capacity of the fifos between X and Y, and make a formal connection with a marked graph model of $\mathcal{E}$. We expect the predicates $\mathsf{en\_in}_X$ and $\mathsf{en\_out}_Y$ should use no parameters other than the delay numbers, and the elastic compositionality theorem to be provable assuming only the absence of cycles with a negative δ-sum. With a current definition, however, we can prove the theorem only with additional assumptions. The right generalization of our enabling predicates, and with it, the right definition of elastic circuits, remains elusive.

References

1. L. Carloni, K. McMillan, and A. Sangiovanni-Vincentelli. Theory of latency-insensitive design. *IEEE Trans. Comput.-Aided Design Integrated Circuits*, 20(9):1059–1076, Sept. 2001.
2. J. Cortadella and M. Kishinevsky. Synchronous elastic circuits with early evaluation and token counterflow. In *DAC '07: Proceedings of the 44th annual Conference on Design Automation*, pages 416–419, New York, NY, USA, 2007. ACM.
3. J. Cortadella, M. Kishinevsky, and B. Grundmann. Synthesis of synchronous elastic architectures. In *DAC '06: Proceedings of the 43rd annual Conference on Design Automation*, pages 657–662, New York, NY, USA, 2006. ACM.
4. S. J. Gay and R. Nagarajan. Intensional and extensional semantics of dataflow programs. *Formal Asp. Comput.*, 15(4):299–318, 2003.
5. S. Krstić, J. Cortadella, M. Kishinevsky, and J. O'Leary. Synchronous elastic networks. In *FMCAD '06: Proceedings of the Formal Methods in Computer Aided Design*, pages 19–30, Washington, DC, USA, 2006. IEEE Computer Society.
6. W. W. Wadge. An extensional treatment of dataflow deadlock. In *Proceedings of the International Sympoisum on Semantics of Concurrent Computation*, pages 285–299, London, UK, 1979. Springer-Verlag.

Bridging the gap between abstract RTL and bit-level designs

Andrew K. Martin

7 Dec, 2006

Abstract

System-ML is a framework, embedded in the Ocaml programming language, for describing synchronous hardware designs. Using system-ML, one describes a hardware design at the RTL level using Hawk-style polymorphic, synchronous streams to represent values in the system that change over time. Designs are expressed using a library of functions for manipulating this polymorphic stream abstract data type, and lifting arbitrary Ocaml code to operate on streams. The framework supports simulation by executing of the underlying Ocaml code. Translation to a conventional hardware description language, for subsequent synthesis using standard tools, is supported through the addition of a reflection mechanism to the Ocaml language.

While the current hardware generation mechanism produces a synthesizable representation of the high-level design, it is not likely to yield high performance hardware. We envision rectifying this situation by allowing the designer to specify a refinement map from the high-level description to bit-level description. This refinement map will allow the designer to specify representation choices for data values, re-timing, and physical partitioning of logic into blocks that can be built using standard tools

This vision has given rise to two research questions. Is the abstraction framework provided by system-ML truly rich enough to provide a significant, meaningful abstraction of real designs in practice? Assuming it is, what should be the nature of the refinement maps to permit synthesis into realistic descriptions of high performance hardware ?

In this talk, I shall address these two questions. In particular, I shall discuss the semantic gap between the system-ML representation and the high-performance bit-level representation of various hardware components, and the extent to which this gap can be be bridged by a user-specified refinement map. Examples will be given, both of situations in which such a map can be constructed, and at least one situation in which such a map cannot.

Model checking transactional memory

John O'Leary
Intel

Bratin Saha
Intel

Mark R. Tuttle
Intel

Transactional memory is a programming abstraction for multi-threaded programming that provides thread synchronization without requiring the explicit use of locks. Locks are notoriously difficult to use correctly: They can lead to deadlock, priority inversion, and subtle errors, and they can impede compositionality. To the programmer, transactional memory has a very simple interface: Every block of code labeled "atomic" is executed as an "atomic transaction" without any interferences from other threads. To the implementer, transactional memory is an entre into the fascinating and subtle world of shared memory protocols.

Shared memory protocols are the way of the future. The programming model for the many-core chips coming from Intel and AMD is likely to be a shared memory model, but shared memory protocols are notoriously hard to get right, and their verification is not well supported by widely-available model checking implementations. For software protocol verification, explicit state model checkers like SPIN and Murphi appear to be the state of the art. Most explicit state model checkers, however, were written with a message-passing model of concurrency in mind, and not shared memory, which leads to two problems. First, shared memory protocols seem to bring explicit state model checker quickly to their knees, because, we suspect, the transition graph for shared memory protocols has a higher branching factor than message passing protocols. Second, models for most explicit state model checkers are written in a guarded-command style, which is a natural style for message passing but not for shared memory.

Our goal is to learn to model check shared memory algorithms in general, and transactional memory in particular. We take the Intel McRT STM [1] implementation of transactional memory as our example, which is itself an interesting, innovative, and subtle shared memory protocol. We choose SPIN [4] as our model checker because it is engineered for high performance, it is famous for its partial order reduction algorithm which has the effect of reducing the branching factor of the transition graph, and its input language, Promela, is a natural language for describing the sequential natural of each thread in a concurrent program. SPIN assumes processes communicate through message channels, but it does have global variables, and what is shared memory but a collection of globabl variables? The use of global variables, however, decimates the performance of the partial order reduction algorithm, which was the whole reason for choosing SPIN in the first place. It took significant engineering to model transactional memory efficiently in SPIN, but now, for example, we can model check the relatively large configuration of 2 threads each running a transaction consisting of 3 loads and stores — proving that every execution of every one of the 14,400 such programs is serializable — and we can do so in just over an hour and a half.

We are not the first paper to consider model checking transactional memory [2, 3]. What distinguishes our work, however, is our intention to allow both transactional and non-transactional loads and stores, meaning that load and stores are not required to appear within an "atomic" block. This means that we need to view the transactional model as an extension to the existing processor memory ordering model, which itself is already challenging to specify. Adding transactions adds new complexities: how do we understand the behavior when two threads access the same memory location, one in a transaction and one outside? This extension is work in progress, and we conclude our talk with a discussion of the issues arising from this extension.

References

[1] A. Adl-Tabatabai and et al. Compiler and runtime support for efficient software transactional memory. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 26–37, June 2006.

[2] R. Alur, K. McMillan, and D. Peled. Model-checking of correctness conditions for concurrent objects. In *Eleventh Annual IEEE Symposium on Logic on Computer Science (LICS'96)*, pages 219—228, July 1996.

[3] A. Cohen, J. O'Leary, A. Pnueli, M. Tuttle, and L. Zuck. Verifying correctness of transactional memories. In M. Sheeran and J. Baumgartner, editors, *Seventh International Symposium on Formal Methods in Computer Aided Design (FMCAD'07)*, Nov. 2007.

[4] G. Holzman. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004.

Access to Circuit Generators in Embedded HDLs

Gordon J. Pace
gordon.pace@um.edu.mt

Christian Tabone
christian.tabone@um.edu.mt

University of Malta

Abstract

General purpose functional languages have been widely used as host languages for the embedding of domain specific languages, especially hardware description languages. The embedding approach provides various abstraction techniques, enabling the description of generators for whole families of circuits, in particular parameterised regular circuits. The two-stage language setting that is achieved by means of embedding, provides a means to reason about the generated circuits as data objects within the host language. Nonetheless, these circuit objects lack information about their generators, or about the manner in which these where generated, which can be used for placement and analysis. In this paper, we use *reFLect* as a functional language with reflection features, to enable us not only to access the circuits, but also the circuit generators. Through the use of code quotation and pattern matching, we propose a framework through which we can access the structure of the circuit in terms of nested blocks that map the generation flow that was followed by the generator.

1 Introduction

Designing and developing a new language for a specific domain, presents various challenges. Not only does one need to identify the basic underlying domain-specific constructs, but if the language will be used for writing substantial programs, one has to enhance the language with other programming features, such as module definition and structures to handle loops, conditionals and composition. Furthermore, one has to define a syntax, and write a suite of tools for the language — parsers, compilers, interpreters, etc — before it can be used. One alternative technique that has been explored in the literature is that of embedding the domain-specific language inside a general purpose language, borrowing its syntax, tools and most of the programming operators. The embedded language is usually developed simply as a library in the host language, thus effectively inheriting all its features and infrastructure.

Functional programming languages have proved to be excellent vehicles for embedding languages in a two-stage language approach. One domain in which this approach has been extensively applied is hardware design. These embedded hardware description languages enable access to the hardware descriptions, although not to the host language code that creates the domain-specific objects. Having access to the generators themselves may be useful since certain structuring information inherited from the control structure of the code generating the domain-specific program may be useful in the analysis of the resulting program. Recently, the use of meta-programming techniques for the embedding of

HDLs has started to be explored [MO06, Tah06, O'D04]. A meta-programming language enables the development of programs that are able to compose and manipulate other programs or even themselves at runtime through the use reflection. Meta-programming techniques provides an opening not only to access the circuits being generated, but even the generators that created these circuits. Therefore, the reasoning about the structure of the circuit generators is made possible, enabling to inspect and analyse the composition of circuits in terms of nested blocks, thus achieving a higher level of abstraction. Furthermore, the information that is provided through the nesting of these circuit blocks, should provide placement hints for the circuits, that can be combined with other user defined placement information.

In this paper, we explore the use of *reFLect*, a meta-programming language, to embed an HDL in such a manner that we can not only access and manipulate the circuit descriptions, but also the circuit generators themselves. We present our approach to mark the boundary of circuit blocks, and illustrate its potential by means of a couple of prefix circuit examples. We plan to use these features to access and control the structure of the circuit generated. In particular, in the future, we plan to use this to optimise circuits produced by hardware compilers, maintaining a compositional view of the compiler, but at the same time having access to information as to which parts of the circuits resulted from which features of the compiled language.

2 Functional Meta-Programming in *reFLect*

reFLect [MO06] is a strongly-typed functional language with meta-programming capabilities. *reFLect* was developed by Intel as the successor of *FL*, as part of the Forte tool [SJO+05]; a hardware verification system used by Intel. *reFLect* and Forte were purposely developed to aid the development of applications in hardware design and verification, and are mostly used for model checking, decision making algorithms and theorem provers for hardware analysis.

reFLect provides quotation and antiquotation constructs, allowing the composition and decomposition of unevaluated expressions, defined in terms of the *reFLect* language itself. These meta-programming constructs provide a form of reflection within a typed functional paradigm setting, enabling direct access to the structure of programs as data objects. This is made possible by giving access to the internal representation of the abstact syntax tree of the quoted expressions. Traditional pattern matching can even be used on this representation, allowing the structure of unevaluated expressions to be inspected and interpreted according to the developer's requirements. Furthermore, by combining the pattern matching mechanism with the quotation features, the developer is able to modify or transform the quoted expression at runtime before evaluation. A more in-depth overview of *reFLect* can be found in [GMO06].

2.1 A brief introduction to meta-programming in *reFLect*

Expressions in *reFLect* can be quoted by enclosing them between `{|` and `|}`. Such expressions are typed as a *term*, denoting a representation for the abstract syntax tree for the enclosed expression. For instance, consider the simple expres-

sion `T AND F`. Normal functional features would evaluate this expression resulting to be semantically equal to `F`. However, the application of quotation marks around this expression, `{| T AND F |}`, delays the evaluation. Note that, the expression `{| T AND F |}` is therefore semantically, and not just syntactically different from `{| F |}`.

The antiquotation construct ` raises its operand one level outside the quotation marks. An antiquotation always appears within quotations, and has two main applications — composition and decomposition of the type term. To compose terms, an antiquotation acts as a splicing construct to join one abstract syntax tree to another. For example, the function below constructs a new term, representing the logical conjunction of two sub-expressions, `a` and `b`. Note that these sub-expressions are also quoted expressions.

```
let compose (a, b) = {| `a AND `b |};
```

A typical functional application of the above definition is given below, followed by the resulting output.

```
:  compose ( {| T OR F |}, {| NOT F |} );
{| (T OR F) AND (NOT F) |}
```

Antiquotations may be used to decompose a term type, by applying pattern matching on the structure of the quoted expression. For example, the function below decomposes the given term into the two operands applied to the `AND` operator, binding the left sub-expression as a term to the variable `x` and the right sub-expression as a term to the variable `y`.

```
let decompose {| `x AND `y |} = (x, y);
```

Consider the previously composed term, and how this can be decomposed back to the original sub-expressions by means of pattern matching, where `x` is bound to `{| T OR F |}` and `y` to `{| NOT F |}`.

```
:  decompose {| (T OR F) AND (NOT F) |};
( {| T OR F |}, {| NOT F |} )
```

The antiquote is needed to extract the sub-expression as a term type. If the function had to be defined without antiquotes using the pattern `{| x AND y |}`, the variables `x` and `y` would be non-binding, thus this would match the expression `{| x AND y |}` literally.

The *reFLect* language offers a number of built-in evaluation functions, to allow total control over the evaluation of the terms being constructed. The most elementary is the **eval** function, which is used to evaluate the contents of a given term, returning the result as a quotation. The **value** function is similar, since it also evaluates the given term, but the result is type casted into the specified type. A **lift** function is available, and it can be applied to any *reFLect* expression. This works by first evaluating the given expression and then by applying quotation marks around the resulting expression, conclusively lifting the evaluated expression to a higher level of quotations.

2.2 Embedding Languages in *reFLect*

The *reFLect* language, together with the meta-functional features that it offers, provides interesting grounds for the implementation of HDLs. Typically, when embedding a language, a deep-embedding is required, since one would want not only to generate programs, but allowing the possibility to give them different interpretations as may be required, and have access to the underlying syntax of the domain-specific language.

In a meta-programming language, one may quote all of the language constructs, resulting in having access to the actual programs as data objects. In *reFLect*, the possibility to pattern match over programs also gives the possibility to look at the structure of an expression. Consequently, in a language like *reFLect* one can build a deep embedding mechanism, simply by using quotations and antiquotations to represent the embedded language using the term datatype. Term manipulation is easily achieved through the use of quotations and antiquotations. The ability to directly control the abstract syntax tree of quoted expression, can be applied to expressions representing elements within a circuit model.

Furthermore, using this style of embedding, one can mark blocks of code, effectively giving structure to the generator of the domain-specific program, which can be accessed and therefore reasoned about in terms of these blocks. This enables the reasoning about the embedded language itself at a higher level of abstraction.

3 Embedding a HDL in *reFLect*

Usually, in a language without reflection, to achieve a deep embedding of a language, one has to handle descriptions as complex data objects. Through the use of reflection, a shallow embedding approach suffices, since quotation can be used to maintain the structure. Terms thus become the primary type of embedded programs which, in our case, contain circuit descriptions with the potential to evaluate to any structure of signals. We use phantom types are used to keep track of the type of the quoted expression, thus enabling a strongly typed embedded language, able to handle type checking over quoted expressions, and distinguish between the different types of signal structures that a term can be evaluated to.

```
lettype *a signal = Signal term;
```

The primitive gates ensure that the signals are of the correct structure and type, whilst decomposing the structure within the type term into the appropriate input signals. These signals or sub-expressions are hence used to compose the required expression.

```
let inv (Signal {| `a |}) = Signal {| NOT `a |};
let and2 (Signal {| (`a, `b) |}) = Signal {| `a AND `b |};
```

Other primitive gates are defined using functions similar to the above, which can be presented to the end user to be used for other circuit descriptions. The constant expressions *high* and *low* are defined for `Signal {|T|}` and `Signal {|F|}` respectively. Additional constants and display functions are also defined to hide

the meta-programming constructs from the end user.

3.1 Representing Signals

A crucial design decision that is needed when developing a HDL is the way circuits inputs and outputs are to be structured [CP07]. In Lava [BCSS98], for example, signals used by the circuit descriptions are grouped together as a structure of signals as opposed to a signal of structures as represented in Hawk [LLC99].

Currently, we are using the signal of structures representation, primarily since it simplifies language design (although not necessarily language usage). An advantage of this representation is that all circuits defined in a language using this representation will always have the same type — taking a single input and producing a single output. This makes the design much cleaner, and the interpretations work seamlessly even when describing complex circuits built from smaller circuit descriptions. On the other hand, the user has to handle the wrapping and unwrapping of the signal type whenever the inner vector values are required. For this we provide functions to convert the signal structure back and forth to the structure values.

```
// From signal values to signal structure
zipp :: (Signal {|bool|}, Signal {|bool|}) -> Signal {|(bool, bool)|}

// From signal structure to signal values
unzipp :: Signal {|(bool, bool)|} -> (Signal {|bool|}, Signal {|bool|})
```

Following this approach, circuit descriptions are require additional code to handle the wrapping and unwrapping of signal. For instance a two-bit multiplexer circuit would be defined as follows:

```
let mux s_ab =
      val (s, ab) = unzipp s_ab in
      val (a, b)  = unzipp ab in
      or2 (zipp (and2 (zipp (inv s, a)), and2 (zipp (s, b))));
```

Applying input values to our multiplexer definition, the function would return the structure of an unevaluated program, which represents the circuit that has been defined. Note that named variable inputs can also be applied.

```
:  mux (zipp (low, zipp (low,high)));
Signal {| (low AND low) OR ((NOT low) AND high) |}
```

3.2 Marking Blocks in Circuits

In *reFLect*, as in most other HDLs, one views and defines circuits as functions. As a circuit description is unfolded, all the internal structure (implicit in the way the generators are invoked) is lost, and all that remains is a netlist of interconnected gates. To enable marking such sub-components inside a circuit, we enable marking blocks, which may be used at any stage in the description.

Such blocks are used in netlist generation, and are planned to be used also in other non-functional features of circuits we plan to implement, including modular verification, placement and local circuit optimisation. For example, one may mark a half-adder definition as a block, and then use two instances of this block to define a full-adder, which may itself be marked as a block (thus containing two sub-blocks inside).

When the abstract circuit description corresponds to a good layout, or describes together related components, preserving such information can be useful. Adding block information to the whole structure of the circuit, adds a higher level of abstraction over the circuit description, enabling not only the possibility to reason about the structure in terms of primitive gates, but also in terms of blocks. For instance, information gathering functions could be defined to count full-adders or half-adders, or any other block. The placement of circuits will also benefit, since this can be organised into blocks, hence decreasing the level of complexity.

The function `makeBlock` composes the structure of a lambda expression of the given circuit definition that is to be marked as a block. The function first generates quoted variables to match the inputs of the circuit. Next, we apply these variables to the function, which would return the program as a data object using the generated variables. Hence, we compose a lambda expression by means of the generated variables and the program structure. Finally, the output of the `makeBlock` function, is yet another function, which has the same type of the input circuit definition, where the input is placed to the rest of the structure as the input of the composed lambda expression, effectively composing a functional application within quotations.

```
let makeBlock circuit =
       let vars = genInputVariables circuit in
       let fnct = circuit vars in
       \(Signal inp) . {| (\`(getTerm vars) .  `(getTerm fnct)) `inp |}
```

The `makeBlock` function composes a term in which the functional application is delayed by means of a quoted lambda expression. Since this function returns another function of the same type, this can therefore be used seamlessly within other circuit definitions. For instance, consider a second multiplexer function, that is defined as a block of the previously defined multiplexer function.

```
let multiplexer = makeBlock mux;
```

By apply a set of inputs to the function multiplexer, the resulting structure is a lambda expression representing the multiplexer circuit as a block or a component. The inputs are separated from the rest of the structure, by means of the delayed functional application. Note that by marking a circuit definition as a block, the inputs are not folded within the internal structure of the circuit, leaving a clear boundary which can be extracted by means of pattern matching.

```
:  multiplexer (zipp (low, zipp (low, inv high)));
Signal {| (\ (v1,(v2,v3)) .
           ((v1 AND v3) OR ((NOT v1) AND v2)) (low,(low, NOT high)) |}
```

Adopting this approach does not affect the simulation of the program in any

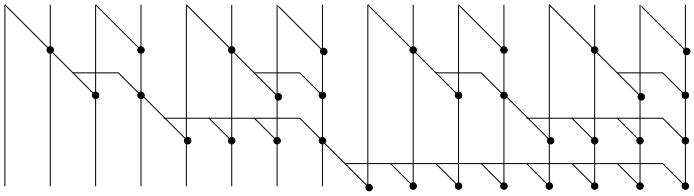


Figure 1: A Sklansky parallel prefix circuit

way, since the only modifications that are made to the object program, are of a syntactic and not semantic nature. The benefit for this program transformation is that patterns matching lambda expressions within terms can be interpreted as blocks or components.

3.3 An Illustrative Example: Parallel Prefix Circuits

In this section we present a some examples for the definitions of parallel prefix circuits, namely the Sklansky network and the Slices network [She07]. Our motivation is to show how to adopt the marking of nested blocks, thus enabling to output a final description with added block details that map the generation flow that was followed by the circuit generator.

The Sklansky network performs the parallel prefiex operation by dividing the input bus into two recursively. The binary operator is applied to the last bit of the first half over all of the second half of the bus. In implementing the Sklansky prefix network circuit (see figure 1), we focus on how the marking of blocks is handled within such descriptions. The recursive definition for the Sklansky network is given as the auxiliary function `skl'` marking blocks as the recursive description unfolds. Any additional parameters, such as the bus width and the operator, are eliminated leaving only the circuit signal as the input. The `makeBlock` function is then used to mark all function calls to the circuit description.

```
letrec skl n op inp =
         let skl' 1 op inps = inps
         /\  skl' n op inps =
               val (lst,rst)  = unzipp (splitSignalBus n inps) in
               let ls2        = skl (busLength lst) op lst in
               let rs2        = skl (busLength rst) op rst in
               let carry      = lastSignal ls2 in
               let apply r    = op (zipp (carry, r)) in
               zipp (ls2  @  map apply rs2) in
         makeBlock (skl' n op) inp;
```

By means of input variables we can create term structures to any specified bus width, hence the create structure is translated to a more readable format. Listing 1 gives the output for the Sklansky circuit for an 8-bit input bus. The statement `BLOCK(vars)... ENDBLOCK` signifies a circuit block with the variables `vars` as inputs, and the final result as outputs. While if this is preceded

Listing 1: The generated output for the Sklansky description of 8 inputs

```
BLOCK (bus_0)
  let (bus_1,bus_2) = ([bus_0 (1)... bus_0 (4)], [bus_0 (5)... bus_0 (8)]) in
  let bus_3 =
        INPUT (bus_1) IN
        BLOCK (bus_4)
          let (bus_5,bus_6) = ( [bus_4 (1), bus_4 (2)],
                                [bus_4 (3), bus_4 (4)] ) in
          let bus_7 =
                INPUT (bus_5) IN
                BLOCK (bus_8)
                  let (bus_9,bus_10) = ([bus_8 (1)], [bus_8 (2)]) in
                  [bus_9 (1), AND2 (bus_9 (1), bus_10 (1))]
                ENDBLOCK in
          let bus_13 =
                INPUT (bus_6) IN
                BLOCK (bus_14)  ...  ENDBLOCK in
          [ bus_7 (1), bus_7 (2),
            AND2 (bus_7 (2), bus_13 (1)), AND2 (bus_7 (2), bus_13 (2)) ]
        ENDBLOCK in
  let bus_19 =
        INPUT (bus_2) IN
        BLOCK (bus_20) ... ENDBLOCK in
  [ bus_3 (1) ... bus_3 (4),
    AND2 (bus_3 (4), bus_19 (1)) ...  AND2 (bus_3 (4), bus_19 (4)) ]
ENDBLOCK
```

by `INPUT(signal)` signifies that `signal` is supplied as the input to the block. Notice how the block markings follow the pattern in which the circuit has been generated.

To illustrate further the use of our embedded language, consider the following description of the Slices parallel prefix circuit description [She07]. Following the recursive decomposition of the circuit (see figure 2), the functions `applyOnEvens` and `applyOnOdds` applies the operator `op`, at even and odd intervals of the bus respectively:

```
let applyOnEvens n op inp =
        let t = unzipp n inp in
        zipp (evens op t);


let applyOnOdds n op inp =
       val (a:as) = unzipp n inp in
       zipp (a:(evens op as));
```

The function `applyOnOddL` makes use of the `unriffL` function to divide the bus into two separate buses, grouping the odd signal occurrences into a single bus, and the even signal occurrences into another bus. The parametrised function is applied to the even signals, and the function `riffL` is used to reverse the functionality of `unriffL`.

```
let applyOnOddL f n inp =
        let as = unzipp n inp in
        val (un_odds, un_evens) = unriffL as in
        let f_un_evens = unzipp (n/2) ( f (zipp un_evens) ) in
        zipp (riffL (un_odds, f_un_evens));
```

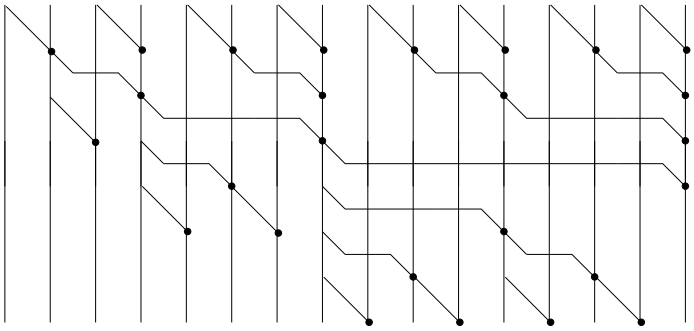


Figure 2: A Slices parallel prefix circuit

The definition for the Slices network is given below, where the functions presented previously are used to compose the prefix network for the general case. At this stage, the circuit is encapsulated in a block, similar to the approach used in the Sklansky definition.

```
letrec pslices n op as =
          let pslices' 1 op as = as
          /\  pslices' 2 op as = val (a:b:cs) = unzipp 2 as in
                                 zipp [a, op (zipp (a,b))]
          /\  pslices' n op as =
                  let slices  =  applyOnEvens n op ->-
                                 applyOnOddL (pslices (n/2) op) n ->-
                                 applyOnOdds n op in
                  slices as in
          makeBlock (pslices' n op) as;
```

4 Related work

HDL implementations like Lava [BCSS98], Hydra [O'D06] and Hawk [LLC99], differ from the work presented in this paper, since these have been developed using the deep embedding technique within the functional language Haskell, while our approach is that of using reflection within *reFLect* as a replacement for deep embedding. Deep embedding allows the developer to provide multiple semantic interpretations of the defined circuits, which is clearly seen in Lava, Hydra and Hawk. These HDLs provide several alternative interpretations of a circuit. For example, an inverter gate can have alternative interpretations defined for simulation, netlist creation and timing analysis. Unlike this approach, our implementation uses quotations to capture the circuit structure as an unevaluated expression. Note that, given a different setting, this expression would have been used to simulate the circuit. However, by delaying the evaluation and by having access to the abstract syntax tree of the expression, we are able to traverse this structure and output additional semantical interpretations. The advantage is that the different semantic interpretations operate on the same instance of the quoted expression. However, this needs to be done in two separate stages, first

to compose the structure, and then to interpret the structure.

The meta-programming features found in *reFLect*, provides not only the possibility to manipulate terms representing primitive gates, but also to manipulate terms representing whole circuit definitions. Embedding a HDL using such features can result in an advantage over other HDL embeddings, since the access and manipulation of whole circuit definitions (the circuit generators), should aid in the reasoning of non-functional aspects of circuits, such as the placement of the primitive elements.

Pebble [LM98], a small language similar to structural VHDL, defines circuit components in terms of blocks. The end-user can describe how the blocks are positioned, meaning that a block can be defined to be placed above or beside another allowing blocks to be placed either vertically or horizontally to each other. In our implementation we adopted this idea of blocks, by means of the meta-programming features provided by *reFLect*. However, the challenges are different from those of Pebble, since Pebble is not an embedded language within a function language. In Pebble, language constructs were developed to define blocks and the placement of these blocks, while our implementation uses quotation constructs to compose lambda expressions, thus delaying the functional application to represent a block in a functional setting. We are currently adding Pebble-style placement constructs to our language.

Wired [ACS05] is another embedded HDL, built upon the concept of connection patterns, in a certain way extending Lava to enable reasoning about connection of circuit blocks. The concepts behind Wired are mostly inspired by Ruby [JS94], more precisely on the adoption of combinators for the placement of circuits. We foresee to follow certain features of Wired, for instance to use combinators at the abstract level of blocks.

Our work is based on similar work done in embedding a Lava-like HDL in *reFLect* [MO06]. As in their case, we base our access to the structure of the circuit descriptions on reflection features of the host language. One difference in our approach is in the signal representation. One of the reasons for this variation is that we try to conceal the use of quotation marks in the circuit descriptions, hence making the reflection features used only in the underlying framework — not forcing the end user to use these constructs. In our approach we emphasise the use of marked blocks which we plan to extend for placement and circuit analysis. We still have a number of features unimplemented — such as the lack of implicit wrapping and unwrapping of structures of signals — which we plan to develop in the near future.

5 Conclusions and Future Work

In this paper, we have presented a rudimentary HDL embedded in the functional meta-programming language *reFLect*. Our main motivation behind the use of reflection is to enable the creation of tagged blocks by looking at the structure and control-flow of the circuit generator. By having access to the circuit generators, it is possible to map the structure of the generators to the structure of the resulting circuits in terms of blocks. We plan to add placement constructs similar to those found in Pebble [LM98], to provide a means to describe how circuit blocks are to be placed in relation to each other. We plan to extend

this by adding circuit combinators, similar to the ones used in Ruby [JS94], and thus use the control given to us into looking at the circuit generators to aid the generation of placement hints.

The additional block information to the generated circuit, provides a higher abstract level than the actual circuit, on which compositional model checking techniques and verification can be applied. Furthermore, by analysing the structure of the generator itself, it should be possible to verify properties of a whole family of circuits.

Another area we intend to explore is that of optimisation of circuits produced by hardware compilers. The use of embedded HDLs for describing hardware compilers has been explored [CP02]. Despite the concise, compositional descriptions enabled through the use of embedded languages, the main drawback is that the circuits lack optimisation. Furthermore, introducing this into the compiler description breaks the compositional description, resulting with a potential source of errors in the compilation process. If one still has access to the recursive structure of the control flow followed by the compiler to produce the final circuit, one can perform post-compilation optimisation, without having to modify the actual compiler code. We plan to investigate this further through the use of the features provided by *reFLect*.

References

[ACS05] Emil Axelsson, Koen Linström Claessen, and Mary Sheeran. Wired: Wire-aware circuit design. In *Proc. of Conference on Correct Hardware Design and Verification Methods (CHARME)*, volume 3725 of *Lecture Notes in Computer Science*. Springer Verlag, October 2005.

[BCSS98] Per Bjesse, Koen Linström Claessen, Mary Sheeran, and Satnam Singh. Lava: Hardware design in Haskell. In *Proc. of International Conference on Functional Programming (ICFP)*. ACM SIGPLAN, 1998.

[CP02] Koen Claessen and Gordon J. Pace. An embedded language framework for hardware compilation. In *Designing Correct Circuits '02, Grenoble, France*, April 2002.

[CP07] Koen Linström Claessen and Gordon J. Pace. Embedded hardware description languages: Exploring the design space. In *Hardware Design and Functional Languages (HFL'07), Braga, Portugal*, March 2007.

[GMO06] Jim Grundy, Tom Melham, and John O'Leary. A reflective functional language for hardware design and theorem proving. *Journal of Functional Programming*, 16(2):157–196, 2006.

[JS94] Geraint Jones and Mary Sheeran. Designing arithmetic circuits by refinement in ruby. *Sci. Comput. Program.*, 22(1-2):107–135, 1994.

[LLC99] John Launchbury, Jeffrey R. Lewis, and Byron Cook. On embedding a microarchitectural design language within haskell. *SIGPLAN Not.*, 34(9):60–69, 1999.

[LM98] Wayne Luk and Steve McKeever. Pebble: A language for parametrised and reconfigurable hardware design. In *FPL '98: Proceedings of the 8th International Workshop on Field-Programmable Logic and Applications, From FPGAs to Computing Paradigm*, pages 9–18, London, UK, 1998. Springer-Verlag.

[MO06] Tom Melham and John O'Leary. A functional HDL in reFLect. In Mary Sheeran and Tom Melham, editors, *Sixth International Workshop on Designing Correct Circuits: Vienna, 25–26 March 2006: Participants' Proceedings*. ETAPS 2006, March 2006. A Satellite Event of the ETAPS 2006 group of conferences.

[O'D04] John O'Donnell. *Embedding a Hardware Description Language in Template Haskell*, chapter Embedding a Hardware Description Language in Template Haskell, pages 143–164. Springer Verlag, 2004.

[O'D06] John O'Donnell. Overview of hydra: a concurrent language for synchronous digital circuit design. *International Journal of Information*, pages 249–264, 2006.

[She07] Mary Sheeran. Parallel prefix network generation: an application of functional programming. In *Hardware Design and Functional Languages (HFL'07), Braga, Portugal*, 2007.

[SJO+05] Carl-Johan H. Seger, Robert B. Jones, John O'Leary, Tom Melham, Mark D. Aagaard, Clark Barrett, and Don Syme. An industrially effective environment for formal hardware verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(9):1381–1405, September 2005.

[Tah06] Walid Taha. Two-level languages and circuit design and synthesis. In *Designing Correct Circuits*, 2006.

Wire-wise correctness for Handel-C synthesis in HOL

Juan Ignacio Perna, Jim Woodcock
Computer Science Department
The University of York
York - United Kingdom

Abstract

The compilation of Handel-C programs into net-list descriptions of hardware components has been extensively used in commercial tools but never formally verified. In this paper we first introduce a variation of the existing semantic model for Handel-C compilation that is amenable for mechanical proofs and detailed enough to analyse properties about the generated hardware. We then use this model to prove the correctness of the wiring schema used to interconnect the different components at the hardware level and propagate control signals among them. Finally, we present the most interesting aspects of the mechanisation of the model and the correctness proofs in the HOL theorem prover.

1 Introduction

Handel-C [6] is a Hardware Description Language (HDL) based on the syntax of the C language extended with constructs to deal with CSP-based [8] parallel behaviour and process communications.

Handel-C's compilation into hardware components (synthesis) was initially formulated to be used in a hardware/software co-design project [13, 12] and later adopted by commercial tools [10]. The compilation is based on the fact that the control of the generated circuits is handled by a pair of handshake signals (*start* and *finish*). The semantics is based on the assumption that the *start* signal will be given to a circuit only if all the previous circuits have already finished. A formal model of the compilation and denotational semantics for the generated hardware has been proposed as a first step towards verifying the compilation scheme [14]. The semantics relies on the concept of state-transformers and uses branching sequences of atomic hardware actions as its semantic domain.

The aim of this paper is to give an accessible account of the work we have carried out on wire correctness for Handel-C [15]. By doing so, we deliberately omit all proofs and consequently many technical details. The rest of the paper is organized as follows: we first propose a reformulation of the semantic domain in [14] in terms of a slightly higher level semantic domain that allows us to capture the actual parallelism at the sequence level. We then redefine the semantic function in terms of the new domain and prove the existence of fix-point solutions for the recursive equations. Finally, we define the concept of wire-satisfiability in the new semantic model and prove the correctness of the wiring schema used in the compilation.

2 The semantic model

The existing denotational semantics for the synthesis process [14] is based in the notion of "branching sequences", where non-branching sequences denote the execution of actions in which no information from the persistent part of the state (i.e., memory locations) is required. Branching nodes, on the other hand, model a choice point (such as the evaluation of the condition in a `while` construct), where the state must be

accessed in order to evaluate the condition. The actual *trace* of the program is obtained by keeping track of the updates over the environment and by pruning the branches that do not get executed.

Even though this semantic model was successfully implemented and tested against existing semantic models for Handel-C [1, 3], we are trying to achieve a higher degree of confidence on it by means of proving its correctness.

When trying to prove a correctness result from the semantics, we observed that the semantics fails to capture the hardware parallelism at the branching sequence level (parallel actions performed combinatorially in the hardware are, in fact, linearised in the branching sequences). The main drawback of this feature of the semantics is that most of the properties we are interested in proving hold after pruning the branching sequence and, in particular, at the end of clock cycles. In this context, proving relatively simple properties (such as that parallel composition is commutative if only the state updates are considered[1]) showed itself to be very complicated, given the need to establish the equivalence at synchronous points (i.e., clock cycle edges) that did not occur at the same depth in different branches being executed in parallel.

To overcome this problem, we observed that Handel-C's synchronous time model allowed us to group the actions performed on each clock cycle into two: *combinatoric actions* (performed before the end of the clock cycle) and *sequential actions* (performed at the very end of the clock cycle). The idea of grouping similar actions together is not fully compatible with the tree-like structure used in branching sequences. In particular, branching conditions are still combinatoric actions and we would ideally want to include them in the corresponding set rather than having them branching the sequences' structure. In this sense, we could regard a sequence of the form $S \boxplus cond \rightarrow S_1 \mid S_2$ (where $\boxplus$ stands for the concatenation operation over Seq) as the factorisation of two sequences: $S \boxplus (cond)^{\perp} : S_1$ and $S \boxplus (\neg cond)^{\perp} : S_2$. Based on this observation, we turn the semantic domain from branching sequences into a set of finite-length linear sequences (this view of the selection construct is consistent with the one used in other formalisms such as [9]).

A similar problem is encountered when constructs are composed in parallel: a structured node is generated in the sequence, preempting the actions in the parallel branches to be put together with the other actions that are executed during the same clock cycle. We overcame this shortcoming by introducing a merge operator that produces a single sequence out of the pair of sequences being composed in parallel.

Our new semantic domain is the powerset of sequences of heterogenous constructors of the type:

$$
\begin{array}{lll}
e \in Seq ::= & \texttt{Empty} & \\
& \mid \texttt{CombBlk}\ \mathscr{P}(Action) \rightarrow Seq & \text{combinatoric behaviour} \\
& \mid \texttt{ClkBlk}\ \mathscr{P}(Action) \rightarrow Seq & \text{sequential behaviour}
\end{array}
$$

The *Action* type, on the other hand, captures the notion of actions that the hardware can perform together with assertion-like extensions (in the sense of Floyd [7] or Hoare and He [9]) in order to allow the verifications we need over the hardware. More precisely, we define the *Action* type as:

[1] The physical wiring of $c_1 || c_2$ is different than the one for $c_2 || c_1$, but the effect both have over the state is the same.

$$
\begin{array}{rll}
e \in \mathit{Action} ::= & w & w \text{ is set to the } \textit{high} \text{ value} \\
| & w_1 \multimap w_2 & \text{the value from } w_2 \text{ is transferred to } w_1 \\
| & w_1 \wedge w_2 & \text{the logical “and” of values in } w_1 \text{ and } w_2 \\
| & \mathit{var} \leftarrow \mathit{val} & \text{the store location } \mathit{var} \text{ gets the value } \mathit{val} \\
| & (\mathit{cnd})^{\perp} & \mathit{cnd} \text{ must be satisfied in the current clock cycle}
\end{array}
$$

2.1 Domain operations

Our semantic domain is going to need two operations in order to be able to manipulate sequences within the semantic function: *concatenation* ($\boxplus$) and *parallel merge* ($\uplus$).

We use a trivial extension of the standard concatenation operator (in order to handle our heterogeneous sequences). This definition allows consecutive nodes based on the same constructor to be put in sequence and this kind of behaviour is not suitable in our semantics (we would expect the actions in the two consecutive nodes to be collected in a single set of actions[2]). We solve this problem by ensuring that the sequences being concatenated using the $\boxplus$ operator avoid the problematic cases (see section 3 for further details).

For the parallel-merge operator, the major complication arises from trying to merge *out-of-phase* sequences (i.e. a pair of sequences in which the structure is not node-wise equal). Even though it would be possible to implement a merge operator that accounts for this problem (using priorities for example), an operator defined in such terms will be very difficult to reason about. On the other hand, the definition of the function becomes completely symmetric (and there is no longer need for priorities) if we can ensure that the sequences are in-phase (i.e., not *out-of-phase*):

$$
\begin{array}{l}
S_1 \uplus S_2 : \mathit{Seq} \times \mathit{Seq} \to \mathit{Seq} \\
S_1 \uplus [] = S_1 \\
[] \uplus S_2 = S_2 \\
(\texttt{CombBlk}\ a_1\ s_1) \uplus (\texttt{CombBlk}\ a_2\ s_2) = (\texttt{CombBlk}\ a_1 \cup a_2\ (s_1 \uplus s_2)) \\
(\texttt{ClkBlk}\ a_1\ s_1) \uplus (\texttt{ClkBlk}\ a_2\ s_2) = (\texttt{ClkBlk}\ a_1 \cup a_2\ (s_1 \uplus s_2)) \\
\textbf{pre}\ \mathit{inPhase}(S_1, S_2)
\end{array}
$$

2.2 Fix-points

As our semantic function is going to use recursive equations, it is necessary to assure the existence of an appropriate semantic domain with fix-point solutions. In order to achieve this goal, we first extend Seq with a bottom element $\perp$ to get $\mathit{Seq}_{\perp} ::= \mathit{Seq} \cup \{\perp\}$ and define our ordering over $\mathit{Seq}_{\perp}$ as the relation:

$$
\begin{array}{l}
\perp \preceq s \wedge [] \preceq [] \wedge \\
(\texttt{CombBlk}\ a\ s_1) \preceq (\texttt{CombBlk}\ a\ s_2) \Leftrightarrow s_1 \preceq s_2 \wedge \\
(\texttt{ClkBlk}\ a\ s_1) \preceq (\texttt{ClkBlk}\ a\ s_2) \Leftrightarrow s_1 \preceq s_2
\end{array}
$$

[2] See [15] for a more detailed explanation about this issue.

and prove it to be a partial order and that the constructors are monotonic with respect to this order.

We have now to verify that the concatenation function preserves the ordering as well. Firstly, we extend the function in order to treat $\perp$ as left and right zero. With this extended definition, we can easily prove $\boxplus$ to be monotonic on its first and second arguments (by structural induction on s_1, s_2 and s).

We also extend the parallel-merge operator to treat $\perp$ as zero when it appears in any of its arguments. We also use $\perp$ as the result for the function when the arguments are out-of-phase as we need to totalise the functions in order to be able to encode them in the HOL theorem prover.

The proof of right monotonicity for $\uplus$ is done by case analysis on the result of $\uplus$'s application (after using the right kind of induction). The proof of $\uplus$'s left monotonicity, on the other hand, cannot be performed by structural induction because the sequences cannot be handled as a pair (the induction principle must be applied to individual sequences in a sequential way). The solution is to make the proof by complete induction over the sum of the lengths of the sequences being merged. This way of proving the theorem is quite laborious, but allows us to instantiate the inductive hypothesis to the sequences of the right shape when needed. Following this approach combined with the case analysis described above, we prove $\uplus$ left monotonic. We then use this result, together with the fact that $\uplus$ is commutative, to prove that $\uplus$ is also right monotonic.

Having shown a suitable partial order over the semantic domain and proved that all the operators preserve that ordering, we can guarantee that fix-point solutions to the recursive equations introduced in the next section exist in our model.

3 Compilation semantics

Using the sequence-based domain defined in the previous section, a denotational semantics can be given to the translation for any given Handel-C syntactic construct. In particular, the semantics is going to be described as a set containing all possible execution traces for the program being synthesised.

As our hardware model captures the semantics of circuits with loop-back connections by replicating the appropriate behaviour (i.e., by means of successive syntactic approximations), we also need to account for the possibility of having an infinite set of traces. This feature clearly preempts any form of explicit description of the semantic set, especially if we take into account that we are also aiming at mechanically verifying our approach.

The obvious solution is to find a predicate $smPred$ capturing the semantics of the generated hardware from Handel-C program c such that we can define the semantic function $\mathtt{Sm}$ as $(\mathtt{Sm}\ c) = \{s : Seq_\perp \mid smPred(c)\}$. In this context, *inductively defined relations* [11, 5] are predicates defined by a set of rules for generating their elements.We adopt this approach to define our semantic predicate, taking the (informal) rules in the compilation schema as the basis for defining the set of rules for our semantic predicate $smPred$.

We also need for a way to incorporate, at the semantic level, the unique pair of wires (*start* and *finish*) generated for each statement in the program by the compilation process. We do so by means of the pair of functions $\pi_s(c)$ and $\pi_f(c)$ returning, respectively, the *start* and *finish* identifiers (belonging to the *wireId* type) for a given circuit c.

3.1 The semantic function

The hardware components generated by the synthesis process start their execution by performing a set of combinatoric actions (essentially to propagate their start signal to their constituent constructs) and also finish by carrying out a set of combinatoric actions (to propagate the finish signal appropriately). Even

more, these "before" and "after" combinatoric actions performed by all the circuits are likely to be executed during the same clock cycle in which the previous circuit was terminating (or the next one is starting). This suggests that these are points in which the joining of action sets should take place (i.e., to condense actions of the same type that happen in the same clock cycle in a single node of our semantics).

We capture this notion by isolating these special points in the semantics, allowing us to have greater control over the structure of the sequences and the way in which they get concatenated/merged. In particular, we redefine our semantic predicate to relate a circuit c with two sets of combinatoric actions: *prologue* and *epilogue* (accounting respectively for actions at the beginning and end of the execution of the circuit) and a *behavioural sequence* (capturing all the actions being executed in between these two combinatoric fragments).

On the other hand, the fact that the `while` construct reduces to pure combinatoric actions in the case where its condition is false makes its behaviour different from the the rest of the constructs (that take, by definition, at least one clock cycle to finish their execution). In this sense, we allow *smPred* to produce two kind of results: *mixed*, to capture the semantics of construct involving both combinatoric and sequential actions (this is our formulation based in the prologue, behavioural sequences and epilogue mentioned above); and *combinatoric*, to encode the particular case in which the resulting semantics involve only combinatoric actions that finish within the current clock cycle. In particular we split the actions in this last type of semantics into two, allowing us to also have a prologue and epilogue when we produce the combinatoric type of result. We formalise this notion by defining the result type:

$$\begin{aligned} e \in smPredResult ::= \ & \texttt{Mixed}\ \mathscr{P}(Action) \rightarrow Seq \rightarrow \mathscr{P}(Action) \\ & \mid \texttt{Combin}\ \mathscr{P}(Action) \rightarrow \mathscr{P}(Action) \end{aligned}$$

In these terms, the semantics of the `delay` construct state that its combinatoric prelude only includes a verification for the *start* wire, while its combinatoric prologue just sets its finish wire to the high value. The behavioural part of the circuit just states that it delays its execution for a single clock cycle (rule 1).

$$smPred(\delta, (\texttt{Mixed}\ \{(\pi_s(\delta))^{\perp}\}\ (\texttt{ClkBlk}\ \{skip\})\ \{\pi_f(\delta)\}))$$

Rule 1: Delay construct

The semantics for the assignment construct is very similar but they the behavioural component is modified to capture the update to the store:

$$\forall\, var, val \bullet smPred(var = val, (\texttt{Mixed}\ \{(\pi_s(var = val))^{\perp}\}\ (\texttt{ClkBlk}\ \{var \leftarrow val\})\ \{\pi_f(var = val)\}))$$

Rule 2: Assignment construct

In the case of constructs c_1 and c_2 being sequentially composed, the prelude transfers the *start* signal from the sequential composition circuit to c_1's *start* and also includes c_1's combinatoric prelude (notice here that we are joining at this point the sequential composition's and c_1's preludes). The behavioural part comprises c_1's behaviour followed by a combinatoric set of actions turning the finish signal of c_1 into c_2's start and performing c_2's prelude, to conclude with c_2's behaviour. Finally, the sequential composition's prologue is composed of c_2's prologue and combinatoric hardware to propagate c_2's finish signal as the sequential composition's one. Rule 3 presents a more formal description of these actions.

We also need to add rules to handle the cases in which one of the constructs (or both) being composed sequentially only performs combinatoric actions. As an example, the case in which the first construct terminates "immediately" is described by rule 4.

$$
\begin{array}{l}
(\forall\, c_1, init_{c_1}, seq_{c_1}, link_{c_1}, c_2, init_{c_2}, seq_{c_2}, link_{c_2} \bullet \\
\quad smPred(c_1,\ (\texttt{Mixed}\ initc_1\ seq_{c_1}\ link_{c_1})) \wedge smPred(c_2, (\texttt{Mixed}\ init_{c_2}\ seq_{c_2}\ link_{c_2})) \Rightarrow \\
\qquad smPred(c_1 \fatsemi c_2, \\
\qquad\quad (\texttt{Mixed}\ init_{c_1} \cup \{(\pi_s(c_1 \fatsemi c_2))^{\perp}\,;\pi_s(c_1) \longleftarrow\!\circ\ \pi_s(c_1 \fatsemi c_2)\} \\
\qquad\qquad seq_{c_1} \boxplus (\texttt{CombBlk}\ (link_{c_1} \cup \{\pi_s(c_2) \longleftarrow\!\circ\ \pi_f(c_1)\} \cup init_{c_2})\ seq_{c_2}) \\
\qquad\qquad (link_{c_2} \cup \{\pi_f(c_1 \fatsemi c_2) \longleftarrow\!\circ\ \pi_f(c_2)\})))
\end{array}
$$

Rule 3: Sequential composition, both components produce `Mixed` results

$$
\begin{array}{l}
(\forall\, c_1, init_{c_1}, link_{c_1}, c_2, init_{c_2}, seq_{c_2}, link_{c_2} \bullet \\
\quad smPred(c_1, (\texttt{Combin}\ initc_1\ link_{c_1})) \wedge smPred(c_2, (\texttt{Mixed}\ init_{c_2}\ seq_{c_2}\ link_{c_2})) \Rightarrow \\
\qquad smPred(c_1 \fatsemi c_2, \\
\qquad\quad (\texttt{Mixed}\ init_{c_1} \cup \{(\pi_s(c_1 \fatsemi c_2))^{\perp}\,;\pi_s(c_1) \longleftarrow\!\circ\ \pi_s(c_1 \fatsemi c_2)\} \cup link_{c_1} \cup \{\pi_s(c_2) \longleftarrow\!\circ\ \pi_f(c_1)\} \cup init_{c_2} \\
\qquad\qquad seq_{c_2} \\
\qquad\qquad (link_{c_2} \cup \{\pi_f(c_1 \fatsemi c_2) \longleftarrow\!\circ\ \pi_f(c_2)\})))
\end{array}
$$

Rule 4: Sequential composition, left component produces a `Combin` result

The symmetric case (the second construct terminates within the same clock cycle in which it was started) is described in a similar way and we do not show it here. The case in which the two constructs being composed in sequence terminate in the same clock cycle also terminates in a single clock cycle (rule 5).

$$
\begin{array}{l}
(\forall\, c_1, init_{c_1}, link_{c_1}, c_2, init_{c_2}, link_{c_2} \bullet \\
\quad smPred(c_1, (\texttt{Combin}\ initc_1\ link_{c_1})) \wedge smPred(c_2, (\texttt{Combin}\ init_{c_2}\ link_{c_2})) \Rightarrow \\
\qquad smPred(c_1 \fatsemi c_2, \\
\qquad\quad (\texttt{Combin}\ init_{c_1} \cup \{(\pi_s(c_1 \fatsemi c_2))^{\perp}\,;\pi_s(c_1) \longleftarrow\!\circ\ \pi_s(c_1 \fatsemi c_2)\} \cup link_{c_1} \cup \{\pi_s(c_2) \longleftarrow\!\circ\ \pi_f(c_1)\} \cup init_{c_2} \\
\qquad\qquad link_{c_2} \cup \{\pi_f(c_1 \fatsemi c_2) \longleftarrow\!\circ\ \pi_f(c_2)\})))
\end{array}
$$

Rule 5: Sequential composition, both components produce `Combin` results

In the case of the parallel composition of c_1 and c_2, the combinatoric prelude propagates the parallel composition's *start* signal to c_1 and c_2 *start* wires and brings together their combinatoric preludes. The behavioural component of the semantics is just constructed by parallel merging c_1 and c_2's behavioural sequences. Finally, the actions in the prologue include the prologues of both c_1 and c_2, together with combinatoric logic to generate the finish signal for the parallel composition only if $\pi_f(c_1)$ and $\pi_f(c_2)$ are in *high* (rule 6).

$$
\begin{array}{l}
(\forall\, c_1, init_{c_1}, seq_{c_1}, link_{c_1}, c_2, init_{c_2}, seq_{c_2}, link_{c_2} \bullet \\
\quad smPred(c_1, (\texttt{Mixed}\ init_{c_1}\ seq_{c_1}\ link_{c_1})) \wedge smPred(c_2, (\texttt{Mixed}\ init_{c_2}\ seq_{c_2}\ link_{c_2})) \Rightarrow \\
\qquad smPred(c_1 || c_2, \\
\qquad\quad (\texttt{Mixed}\ init_{c_1} \cup init_{c_2} \cup \{(\pi_s(c_1 || c_2))^{\perp}\,;\pi_s(c_1) \longleftarrow\!\circ\ \pi_s(c_1 || c_2)\,;\pi_s(c_2) \longleftarrow\!\circ\ \pi_s(c_1 || c_2)\} \\
\qquad\qquad seq_{c_1} \uplus seq_{c_2} \\
\qquad\qquad (link_{c_1} \cup link_{c_2} \cup \{\pi_f(c_1 || c_2) \longleftarrow\!\circ\ \pi_f(c_1) \wedge \pi_f(c_2)\}))))
\end{array}
$$

Rule 6: Parallel composition, both components produce `Mixed` results

As with the sequential composition construct, it is also necessary to address the cases involving the instantaneous termination of the constructs being composed in parallel. We omit these additional rules (together with the rules handling the selection and input/output constructs) due to space constraints, but

the whole set of rules can be found in [15].

The semantics for the `while` construct needs to provide rules for handling the two possible outcomes of the evaluation of the looping condition. In particular, the first rule accounts for the case when the condition is false and the `while` terminates in the same clock cycle in which it was started (Rule 7).

$$(\forall\, cond \bullet smPred(cond * body, (\texttt{Combin}\ \{(\pi_s(cond * body))^{\perp}\,;((\neg cond))^{\perp}\}\{\pi_f(cond * body)\})))$$

Rule 7: While construct, false condition

The case in which the looping condition holds, on the other hand, applies the traditional notion of syntactic approximations [16, 17] in order to turn the loop-back wiring schema used at the hardware level into replicated behaviour in our semantics. In order to achieve this goal, we need to provide two rules for this case: a rule to capture the first approximation to the solution (by means of a single execution of the `while`'s body) and another one to capture the way in which we can construct a longer approximation from an existing one.

The first approximation calculates the semantics of the while's body, assumes that the looping-condition does not hold and uses its prologue to signal the `while`'s termination (rule 8).

$$\begin{array}{l}(\forall\, b, init_b, seq_b, link_b, cond \bullet \\ \quad smPred(b, (\texttt{Mixed}\ init_b\ seq_b\ link_b)) \Rightarrow \\ \quad\quad smPred(cond * b, \\ \quad\quad\quad (\texttt{Mixed}\ \{(\pi_s(cond * b))^{\perp}\,;(cond)^{\perp}\,;\pi_s(b) \longleftarrow\!\circ\ \pi_s(cond * b)\} \cup init_b \\ \quad\quad\quad\quad\quad seq_b \\ \quad\quad\quad\quad\quad \{\pi_s(cond * b) \longleftarrow\!\circ\ \pi_f(b)\} \cup link_b \cup \\ \quad\quad\quad\quad\quad\quad \{((\neg cond))^{\perp}\,;(\pi_s(cond * b))^{\perp}\,;\pi_f(cond * b) \longleftarrow\!\circ\ \pi_s(cond * b)\})\end{array}$$

Rule 8: While construct, first approximation

The final rule for the `while` construct is meant to extend an existing approximation (generated either by the basic rule above or by previous applications of itself). The approximation is constructed by appending one expansion of the body and the proper linking combinatoric action in front of the approximation's behavioural sequence (rule 9).

$$\begin{array}{l}(\forall\, b, init_b, seq_b, link_b, init_{approx}, seq_{approx}, link_{approx}, cond \bullet \\ \quad smPred(b, (\texttt{Mixed}\ init_b\ seq_b\ link_b)) \wedge smPred(cond * b, (\texttt{Mixed}\ init_{approx}\ seq_{approx}\ link_{approx})) \Rightarrow \\ \quad\quad smPred(cond * b, \\ \quad\quad\quad (\texttt{Mixed}\ init_{approx} \\ \quad\quad\quad\quad\quad seq_{approx} \boxplus (\texttt{CombBlk}\ (\{(cond)^{\perp}\,;(\pi_s(cond * b))^{\perp}\,;\pi_s(b) \longleftarrow\!\circ\ \pi_s(cond * b)\,; \\ \quad\quad\quad\quad\quad\quad\quad\quad \pi_s(cond * b) \longleftarrow\!\circ\ \pi_f(b)\} \cup link_{approx} \cup init_b)\ seq_b) \\ \quad\quad\quad\quad\quad link_{approx})\end{array}$$

Rule 9: While construct, inductive approximations

Having reformulated the semantic predicate *smPred*, we still need to provide a way to use it in the definition of our semantic function `Sm`. Considering the fact that the prelude and prologue components of the semantic predicate are just sets of combinatoric actions, we define the semantic function for a given syntactic construct c as:

$$(\texttt{Sm}\ c) = \{seq \mid (smPred(c, (\texttt{Mixed}\ init_c\ seq_c\ link_c)) \Rightarrow (seq = (\texttt{CombBlk}\ init_c\ seq_c) \boxplus (\texttt{CombBlk}\ link_c))) \wedge (smPred(c, (\texttt{Combin}\ init_c\ link_c)) \Rightarrow (seq = (\texttt{CombBlk}\ init_c \cup link_c)))\}$$

3.2 Consistency considerations

Our definition of the semantic function relies on the sequences produced to have certain properties in order to satisfy the preconditions imposed by the operations in our semantic domain.

In particular, the usage we have made of the concatenation function requires the behavioural sequences produces by the *smPred* to have the pattern ($\texttt{CombBlk}\ a_1\ \ldots\ s\ \ldots (\texttt{CombBlk}\ a_n)$) for suitable sets of combinatoric actions a_1 and a_n and sequence s. Even more, the way in which the semantics apply the parallel merge operator forces us to ensure that any possible pair of behavioural sequences produced by the semantic predicate is *inPhase*.

The methodology we use to ensure the satisfaction of this requirements is as follows: we first define a suitable subset of the sequences in our semantic domain and prove that the properties mentioned above hold for any element in this subset. Then we prove that all the sequences produced by the semantic predicate belong to this subset, ensuring that the semantic function also satisfies the required conditions.

3.2.1 Clock-Bound sequences.

Given the properties we need to satisfy, it is easy to observe that sequences in the subset we are trying to define should have their *boundaries* (i.e., first and last elements) constructed from the `ClkBlk` constructor. The predicate *clkBound* captures this notion by means of the inductive definition:

$$(\forall\, cka \in \mathscr{P}(Action) \bullet clkBound(\texttt{ClkBlk}\ cka) \wedge (\forall\, cka, ca \in \mathscr{P}(Action), s \in Seq \bullet clkBound(s) \Rightarrow clkBound(\texttt{ClkBlk}\ cka\ (\texttt{CombBlk}\ ca\ s)))$$

We also need to show that any pair of sequences satisfying the *clkBound* property is also *inPhase*. Fortunately, the conditions imposed by the *clkBound* predicate also guarantee the satisfaction of this requirement:

$$\forall\, s_1, s_2 \in Seq \bullet clkBound(s_1) \wedge clkBound(s_2) \Rightarrow inPhase(s_1, s_2)$$

3.2.2 The semantic predicate only generates clock-bounded sequences.

We need to show that all behavioural sequences generated by *smPred* belong to *clkBound* subtype. To do so, we first need two lemmas proving that the semantic domain operators preserve the *clkBound* property.

Regarding the application of $\boxplus$ to clock-bounded sequences, it is not possible to prove that the concatenation of two *clkBound* sequences is still a clock-bounded sequence (it is not even possible to apply $\boxplus$ as the arguments do not satisfy its precondition). On the other hand, it is possible to prove (by means of the induction principle induced by *clkBound*'s definition) that:

$$clkBound_{\boxplus} \vdash \forall\, s_1, s_2 \in Seq, a \in \mathscr{P}(Action) \bullet$$
$$clkBound(s_1) \wedge clkBound(s_2) \Rightarrow clkBound(s_1 \boxplus (\texttt{CombBlk}\ a\ s_2))$$

This result is strong enough to aid us in the proof of *smPred*'s preservation of the *clkBound* property. In fact, it is easy to observe that the way in which the concatenation function is applied in the above lemma is the only way in which the function is applied in *smPred*'s definition.

The proof of the lemma stating $\uplus$'s monotonicity respect to the *clkBound* property is very complicated. The complication arises because of the way in which the *clkBound*'s induction principle has to be applied (i.e., in sequential order) together with $\uplus$'s definition. In order to overcome this problem, we capture $\uplus$'s behaviour in the inductive predicate *parMerge* and prove it equivalent to $\uplus$.

The main advantage of the predicate-based form of $\uplus$ is that it provides an induction principle, allowing us to induct on the merge operator rather than on sequences or its properties. We then state that *parMerge* preserves the *clkBound* property:

$$\forall\, s_1, s_2, res \in Seq \bullet clkBound(s_1) \wedge clkBound(s_2) \wedge parMerge(s_1, s_2, res) \Rightarrow clkBound(res)$$

The proof of the above result is now very straightforward, as *parMerge* provides an induction principle over pairs of sequences of the right kind. We use the equivalence to show the result holds for $\uplus$ as well, providing us with the lemma:

$$clkBound_{\uplus} \vdash \forall\, s_1, s_2, res \in Seq \bullet clkBound(s_1) \wedge clkBound(s_2) \Rightarrow clkBound(s_1 \uplus s_2)$$

With the two main lemmas of this section, it is possible to prove (by induction on the semantic predicate rules) that the behavioural sequences generated by *smPred* belong to the subset of *Seq* induced by the *clkBound* predicate:

$$clkBound_{seq_c} \vdash \forall\, c \in Contructs; init_c, link_c \in \mathscr{P}(Action); seq_c \in Seq \bullet$$
$$smPred(c, (\texttt{Mixed}(init_c\ seq_c\ link_c)) \Rightarrow clkBound(seq_c)$$

This final result ensures that the operators in the semantic domain are always applied within their definition domain by the semantic predicate.

3.3 Pruning

So far we have described the semantics of the translation from Handel-C into net-lists as a (possibly infinite) set of finite-length sequences. In order to complete the semantic description of the generated circuits, we need to find (if it exists) a single sequence (from the set of possible traces) that specifies the actual execution path and outcome of the program being synthesised.

As in [14], we define two auxiliary functions: $\Delta Env : \texttt{env} \rightarrow Action \rightarrow \texttt{env}$ and $flattenEnv : \texttt{env} \rightarrow \texttt{env}$. The former updates the environment according to the action passed as argument by means of rewriting the appropriate function using λ-abstractions. In the particular case of *skip*, ΔEnv treats it as its unit value and returns the same environment.

On the other hand, $flattenEnv$ is meant to be used to generate a new environment after a clock cycle edge. In particular, it flattens all wire values (to the logical value false), resets the channel values to the undefined value and advances the time-stamp by one unit.

We define the execution of sets of hardware actions by means of the predicate $exec : \texttt{env} \times \mathscr{P}(Assertion) \rightarrow (\texttt{env}, \mathscr{P}(Assertion))$ defined operationally by the rules:

$$\frac{s = \{\}}{exec(e, s) = (e, \{\})} \qquad \frac{s \neq \{\} \wedge a \in s}{exec(e, s) = exec(\Delta Env(e, a), s - \{a\})}$$

We also need to be able to handle assertions, so we introduce the function $sat^{\perp} : \texttt{env} \times \mathscr{P}(Assertion) \rightarrow bool$ defined as $sat^{\perp}(e, set) = \forall\, a \in set \bullet holds(a, e)$, where $holds(a, e)$ is true iff the assertion a is true in the environment e.

As we are dealing with sets of actions and assertions on each node of our sequences, we need to define the collective effect of this heterogeneous set of actions over the environment. The first difficulty we face when defining how a set of actions is going to be executed is that the initial order between actions and conditions has been lost. This is, however, not a problem if we consider that assertions and control flow conditions refer only the present value of the memory and all variables preserve their values during the whole clock cycle. This fact makes the evaluation of assertions and control flow decisions independent of the combinatoric actions performed in parallel with them and they can be evaluated at any time.

From the observation above, we can collect assertions together into a set (As) and the remaining "unconditional actions" into another one (HAs). We induce a partition of the initial set into two equivalence classes, by means of the functions ∇_{A} and ∇_{HA}. Also from the observations above, we know that control-flow assertions can be evaluated at any time, and that wire-correctness assertions must be evaluated after HAs, allowing us to establish the following order of evaluation: HAs $\prec$ As.

Taking advantage of the execution order outlined above, we can introduce the function $setExec : \texttt{env} \times \mathscr{P}(Action) \rightarrow (\texttt{env} \cup \perp)$ defined by the rule[3] (we omit the projection operator in $exec$'s application):

$$\frac{e_{new} = exec(e, \nabla_{\mathrm{HA}}(s)) \wedge sat^{\perp}(e_{new}, \nabla_{\mathrm{A}}(s))}{setExec(e, s) = e_{new}}$$

In turn, the above functions can be used to define a single-node execution function for sequences $seqExec : \texttt{env} \times Seq \rightarrow ((\texttt{env}, Seq_{\perp}) \cup \{(\texttt{env}, \checkmark\})$. The simplest case is successful termination (i.e., when the sequence we are trying to execute is empty), captured by the rule:

$$\frac{seq = []}{seqExec(e, seq) = (e, \checkmark)}$$

The next case describes the result of the execution of a sequence that begins with a set of actions containing unsatisfiable conditions (the symmetric case is similar and we omit it here):

$$\frac{\exists\, ca \in \mathscr{P}(Action), s_1 \in Seq \bullet seq = (\texttt{CombBlk}\ ca\ s_1) \wedge setExec(e, ca) = \perp}{seqExec(e, seq) = (e, \perp)}$$

The case in which it is possible to perform all actions and satisfy all tests within a combinatoric node in the head of the sequence being executed is described by the following rule:

$$\frac{\exists\, ca \in \mathscr{P}(Action), s_1 \in Seq \bullet seq = (\texttt{CombBlk}\ ca\ s_1) \wedge setExec(e, ca) = e_{new}}{seqExec(e, seq) = (e_{new}, s_1)}$$

[3]To keep the presentation compact, we omit the counterpart of this rule that maps all the cases when the antecedent does not hold to the $\perp$ value.

The counterpart of the above rule (dealing with sequences starting with a clock-edged block) is as follows:

$$\frac{\exists\, ca \in \mathscr{P}(Action), s_1 \in Seq \bullet seq = (\texttt{ClkBlk}\ ca\ s_1) \wedge setExec(e, ca) = e_{new}}{seqExec(e, seq) = (flattenEnv(e_{new}), s_1)}$$

Note that the environment needs to be *flattened* after all actions at the clock edge have taken place. The flattening can take place only at this point because of the possibility of having a value being transmitted over a bus (we will loose the value being transferred if we flatten the environment before updating the store with it).

In order to get the actual execution path (for the case in which the program terminates) we define the operator $prune : \texttt{env} \rightarrow \mathscr{P}(Seq) \rightarrow (\texttt{env}, \mathscr{P}(Seq_\perp)) \cup \{\texttt{env}, \checkmark\}$ that advances one step at a time over all sequences in the set (using the function *seqExec* defined above), updating the environment accordingly and removing unsatisfiable sequences. To deal with the infiniteness of the set, we need to observe that the sequences in the set can be partitioned into equivalence classes, grouping together sequences that share the same head. In particular, the "infiniteness" is brought to the set by the approximation chains used in the semantics for circuits with loop-back wiring. The way in which the approximations are constructed forces all of them to share the same trace of actions and differ only at the very last node of each of them. In this way, we have only a finite number of equivalence classes (the amount of classes is directly proportional to the branching in the control flow of the program, which is known to be finite) at any given time and the effect of all the sequences in a given class over the environment in the current clock cycle is the same. Moreover, as Handel-C's control flow is governed by boolean conditions, only one of the possible branches is executable at any given time making our semantic traces *mutually exclusive*. From this observation, it follows that only one of the equivalence classes will remain in the set of traces after the execution of the combinatoric header (all the other traces with unsatisfiable conditions will reduce to bottom and will be removed from the set).

4 Wire-wise Correctness

We are now in a good position to verify the correctness of the wiring schema used to link the different components used at the hardware level. In particular, we are interested in proving the wire-correctness of the generated hardware by means of verifying whether (a) the activation signal is propagated from the *finish* signal of the previous circuit; (b) the start signal is given to each component at the right clock cycle; and (c) the internal wiring of each circuit propagates the control signals in the right way and produces the *finish* pulse at the right time.

In order to answer these questions, it is worthwhile noting that part of the verification is straightforward from the way in which the compilation is done. In particular, each construct is compiled into a *black box* and it is interfaced only through its *start* and *finish* wires. In this sense, it is impossible for a circuit to be started by any component but the circuit containing it, preempting the chance of a component c being activated by a random piece of hardware. After this observation, to prove (a) we only need to prove that the *finish* wire of the appropriate circuit is set to *high* by the time the subsequent circuit is started.

Regarding the verification of the *start* signal given at the right time (condition (b)), we have already included assertions regarding the *start* signal in the combinatoric prelude of all constructs in order to make sure that the circuit receives a *start* pulse during the first clock cycle of its execution. The remaining aspect of this question is whether our semantic model of the hardware activates the circuits at the right clock cycle (and hence, verifies the *start* signal is at the right clock cycle). Towards this question, the synchronous-time model used in Handel-C together with the component-based approach used in the compilation allows us to verify that the timing in our semantic model is equivalent to the one of the generated hardware (this is a

simple proof by induction over the amount of clock cycles each of the constructs take). In this context, assuming that the generated hardware implements the timing model correctly, we only need to verify that the wire-related assertions are satisfied in order to verify (b).

The rest of this section is devoted to verifying the wire-correctness of the hardware based on the observations above. In particular, we first define a way of calculating if a given wire is *high* within the current clock cycle. We then define the concept of wire-satisfiability capturing the notion of wire-based assertions being satisfied in a given set of combinatoric actions and use it to prove all the circuits are given the start signal in the clock cycle they are supposed to be started.

4.1 Wire-transfer closure

The fact that all the combinatoric actions happening at a given clock cycle are collected together in a set provides enough information to calculate which wires hold the *high* value in that clock cycle. This is because of the way in which clock edges are handled: all the wires are set to *low*, forcing the presence of explicit combinatoric actions to set the appropriate wires to *high* again in the next clock cycle.

On the other hand, the information about a wire holding the *high* value can be given by either an explicit single formula (such as $\pi_s(c)$) or as a chain of value transfers from other wires (such as $\{w_1 \longleftarrow\!\circ\, w_2\,;w_2\}$). In this context, we define the notion of a wire in *high* by means of the $highWire$ (inductive) predicate:

$$\begin{aligned}
&\forall\, w_1, set \bullet (w_1 \in set) \Rightarrow highWire(w_1, set)) \wedge \\
&\forall\, w_1, w_2, w_3, set \bullet \\
&\qquad highWire(w_2, set) \wedge highWire(w_3, set) \wedge (w_1 \longleftarrow\!\circ\, (w_2 \wedge w_3) \in set) \Rightarrow highWire(w_1, set) \wedge \\
&\forall\, w_1, w_2, set \bullet highWire(w_2, set) \wedge (w_1 \longleftarrow\!\circ\, w_2 \in set) \Rightarrow highWire(w_1, set)
\end{aligned}$$

The predicate $highWire$ captures the notion of a wire w holding the high value provided the actions in set are executed. From this definition we were able to prove some lemmas that will be necessary in the following sections. In particular, if a given wire w holds the high value in a given set s, then it will still does so in a bigger set:

$$setExtension \vdash highWire(w, s) \Rightarrow highWire(w, (s \cup s_1))$$

It is also possible to prove that any explicit action in the set of actions setting up a wire w_1 to the *high* value can be replaced by a pair of actions, one setting up a wire w_2 to the *high* value and another one to propagate its value into w_1. All these can be done without affecting the validity of the $highWire$ predicate:

$$highWire(w, s \cup \{w_1\}) \Rightarrow highWire(w, s \cup \{w_2\,; w_1 \longleftarrow\!\circ\, w_2\})$$

With these results, we are able to prove (by induction over the semantic predicate) that for any given construct c the prologue in the semantics always sets its finish wire $\pi_f(c)$ to *high*:

$$\begin{aligned}
&smPred(c, (\texttt{Mixed}\ init_c\ seq_c\ link_c)) \Rightarrow highWire(\pi_f(c), link_c) \wedge \\
&smPred(c, (\texttt{Combin}\ init_c\ link_c)) \Rightarrow highWire(\pi_f(c), link_c)
\end{aligned}$$

This is the first result towards proving (a) in the introduction of this section. In order to complete our verification of (a) we introduce a new assertion type $(w_1 \leftrightsquigarrow w_2)^{\perp}$ defined to hold iff $highWire(w_1) \wedge$

$highWire(w_2)$ holds in a given set of combinatoric actions. We then modify our semantic predicate to include this new type of assertions at the places where condition (a) is supposed to hold. As an example, we show the updated version of the semantics for the sequential composition construct:

$$
\begin{array}{l}
(\forall\, c_1, init_{c_1}, seq_{c_1}, link_{c_1}, c_2, init_{c_2}, seq_{c_2}, link_{c_2} \bullet \\
\quad smPred(c_1, (\texttt{Mixed}\ init_{c_1}\ seq_{c_1}\ link_{c_1})) \wedge smPred(c_2, (\texttt{Mixed}\ init_{c_2}\ seq_{c_2}\ link_{c_2})) \Rightarrow \\
\qquad smPred(c_1 \fatsemi c_2, \\
\qquad (\texttt{Mixed}\ init_{c_1} \cup \{\pi_s(c_1) \longleftarrow \pi_s(c_1 \fatsemi c_2)\} \\
\qquad\qquad seq_{c_1} \boxplus (\texttt{CombBlk}\ (link_{c_1} \cup init_{c_2} \cup \{\pi_s(c_2) \longleftarrow \pi_f(c_1)\,;(\pi_f(c_1) \leftrightsquigarrow \pi_s(c_2))^{\perp}\}\ seq_{c_2}) \\
\qquad\qquad (link_{c_2} \cup \{\pi_f(c_1 \fatsemi c_2) \longleftarrow \pi_f(c_2)\}))))
\end{array}
$$

4.2 Assertion satisfiability

Having defined the concept of wire being set to *high*, we need a way to capture the idea of satisfaction of our assertions regarding wires. In particular, we say that all the wire-related assertions in a set are *satisfied* iff the predicate $wireSAT$ holds[4]:

$$wireSAT(s) = \forall\, w \in WireIds \bullet (w)^{\perp} \in s \Rightarrow highWire((w)^{\perp}, s)$$

Having the definition of wire-satisfiability we can prove that if the hardware generated from any given syntactic construct c is started, then all the wire-related assertions in its combinatoric prelude hold:

$$
\begin{array}{l}
smPred(c, (\texttt{Mixed}\ init_c\ seq_c\ link_c)) \Rightarrow wireSAT(init_c \cup \{\pi_s(c)\}) \wedge \\
smPred(c, (\texttt{Combin}\ init_c\ link_c)) \Rightarrow wireSAT(init_c \cup \{\pi_s(c)\})
\end{array}
$$

This lemma is proving consideration (a) from the previous section holds in the compilation: the activation signal is propagated from the parent/previous circuit into the start signal of the current one. In fact, even though the above theorem is just stating that it happens that the right *start/finish* signals get the high value in the appropriate clock cycle, evidence gathered during the proof process showed that the *high* value actually gets propagated between them, proving consideration (a) to its full extent. It is also showing that consideration (b) holds: the start *signal* is given to each circuit at the appropriate time (provided that the time models of the hardware compilation are correct regarding the Handel-C's semantics).

We also proved that the epilogue set of combinatoric actions satisfies $wireSAT$:

$$
\begin{array}{l}
smPred(c, (\texttt{Mixed}\ init_c\ seq_c\ link_c)) \Rightarrow wireSAT(link_c) \wedge \\
smPred(c, (\texttt{Combin}\ init_c\ link_c)) \Rightarrow wireSAT(link_c)
\end{array}
$$

Provided that (a) and (b) hold, the verification of (c) can be reduced to proving that the assertions in the behavioural part of the semantic predicate are satisfied. The rationale behind this affirmation is that the base cases for the $smPred$ trivially satisfy (c) while compound circuits are can be regarded as placeholders linking *start/finish* wires of different components by means of combinatoric actions. The assertions introduced in order to verify (a) and (b) are, in this context, checking that those actions are propagating the right value among the different involved components.

[4]We replace assertions of the form $(w_1 \leftrightsquigarrow w_2)^{\perp}$ by the equivalent set of assertions $\{(w_1)^{\perp}\,;(w_2)^{\perp}\}$, allowing us to use the simple form of satisfiability defined above.

In order to verify the behavioural sequences from the semantic predicate we first need to extend the concept of wire-satisfiability to sequences. We do so by defining the function $wireSAT_{seq}$ as follows:

$$wireSAT_{seq}(\bot) = F \wedge wireSAT_{seq}([]) = T$$
$$wireSAT_{seq}(\texttt{CombBlk}\ a\ s_1) = wireSAT(a) \wedge wireSAT_{seq}(s_1)$$
$$wireSAT_{seq}(\texttt{ClkBlk}\ a\ s_1) = wireSAT(a) \wedge wireSAT_{seq}(s_1)$$

Before being able to use the $wireSAT_{seq}$ function to prove that the wiring is correct in the behavioural sequences generated by the *smPred* predicate we need to prove two lemmas regarding the $\boxplus$ and $\uplus$ preserving the wire-satisfiability property for sequences if it holds true in all their arguments. The case of concatenation is fairly straightforward by induction over the sequences being sequentially composed:

$$wireSAT_{seq}(s_1 \boxplus s_2) \Leftrightarrow wireSAT_{seq}(s_1) \wedge wireSAT_{seq}(s_2)$$

To prove the equivalent result for the parallel-merge operator is a very complicated task given the already mentioned lack of an induction principle capable of handling two sequences as a pair. On the other hand, as we are still within the context of the semantic predicate (and hence, within the subclass of clock-bounded sequences) we can prove the easier goal:

$$wireSAT_{seq}(s_1) \wedge wireSAT_{seq}(s_2) \wedge parMerge(s_1, s_2, res) \Rightarrow wireSAT_{seq}(res)$$

and then use the equivalence between $parMerge$ and $\uplus$ to deduce the equivalent result for $\uplus$. With these two results, we are able to prove the correctness of the wiring for the behavioural sequences produced by *smPred*:

$$smPred(c, (\texttt{Mixed}\ init_c\ seq_c\ link_c)) \Rightarrow wireSAT_{seq}(seq_c)$$

With the three main theorems of this section, it is easy to show that the wiring is correct (regarding our wire-satisfiability criteria) for any given syntactic construct in the core language.

5 Conclusions and future work

The main contributions of this work are an improved semantic model for the hardware components synthesised from Handel-C programs and the verification of the wiring schema used to handle the control flow among those components.

This work is based on a more abstract semantic domain than the one used in previous works [14] and allows a better description of the parallel behaviour exhibited by the hardware components generated by the compilation process. In particular, we have defined our semantic domain in terms of a deep embedding of sequences of state-transformers [4] in Higher Order Logic (HOL). We have also established a partial order relationship over the domain and proved the existence of fix-point solutions to our inductive approximations for recursive constructs.

The synthesis process we are formalising [13, 12] is based on the assumption that no hardware component will be activated unless a precise signal has been given to it through its interface. We have captured this synthesis process by encoding Handel-C's syntactic constructs in HOL and providing a semantic function that associates each construct in the language to its representation in our semantic model for the hardware.

Moreover, the way in which Handel-C's synchronous nature is encoded in the model introduces explicit information about the value held by the wires used to link different components. We have taken advantage of this feature to formally verify the correctness of the wiring schema used in the compilation.

Even though we have proved that each of the hardware components propagate the control token in the right way, we still need to prove that the hardware generated by the compilation rules is *correct* (i.e., semantically equivalent to its original Handel-C code). This correctness proof will also allow us to discharge the only assumption of this work: that the timing model of the generated hardware is consistent with the one for Handel-C. Towards this end, our next step is to prove the existence of an equivalence relationship using the semantic models for Handel-C [2] and the semantics for the generated hardware presented in this paper.

References

[1] A. Butterfield. Denotational semantics for prialt-free Handel-C. Technical report, The University of Dublin, Trinity College, December 2001.

[2] A. Butterfield. A denotational semantics for Handel-C. In *Formal Methods and Hybrid Real-Time Systems*, pages 45–66, 2007.

[3] A. Butterfield and J. Woodcock. Semantic domains for Handel-C. *Electronic Notes in Theoretical Computer Science*, 74, 2002.

[4] A. Butterfield and J. Woodcock. Semantics of prialt in Handel-C. In *Concurrent Systems Engineering*. IOS Press, 2002.

[5] J. Camilleri and T. Melham. Reasoning with Inductively Defined Relations in the HOL Theorem Prover. Technical Report 265, August 1992.

[6] Celoxica Ltd. *DK3: Handel-C Language Reference Manual*, 2002.

[7] R. Floyd. Assigning meaning to programs. In *Mathematical Aspects of Computer Science*, number 19 in Proceedings of Symposia in Applied Mathematics, pages 19–32. American Mathematical Society, 1967.

[8] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 26(1):100–106, 1983.

[9] C.A.R. Hoare and H. Jifeng. *Unifying Theories of Programming*. Prentice-Hall, 1998.

[10] Celoxica Ltd. *The Technology behind DK1*, August 2002. Application Note AN 18.

[11] T. F. Melham. A Package for Inductive Relation Definitions in HOL. In *Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and its Applications, Davis, August 1991*. IEEE Computer Society Press.

[12] I. Page. Constructing hardware-software systems from a single description. *Journal of VLSI Signal Processing*, 12(1):87–107, 1996.

[13] I. Page and W. Luk. Compiling Occam into field-programmable gate arrays. In *FPGAs, Oxford Workshop on Field Programmable Logic and Applications*, pages 271–283. 1991.

[14] J. Perna and J. Woodcock. A denotational semantics for Handel-C hardware compilation. In *ICFEM*, pages 266–285, 2007.

[15] J. Perna and J. Woodcock. Proving wire-wise correctness for Handel-C compilation in HOL. Technical report, Computer Science Department, The University of York, 2007.

[16] A.W. Roscoe and C.A.R. Hoare. The laws of occam programming. *Theoretical Computer Science*, 60:177–229, 1988.

[17] D. Scott and C. Strachey. Towards a mathematical semantics for computer languages. In *Proceedings Symposium on Computers and Automata*.

Obsidian: GPU Programming in Haskell

Koen Claessen, Mary Sheeran, Joel Svensson
Chalmers

1 Introduction

Obsidian is a language for data-parallel programming embedded in Haskell. As the Obsidian programs are run, C code is generated. This C code can be compiled for an NVIDIA 8800 series GPU (Graphics Processing Unit), or for other high-end NVIDIA GPUs. The idea is that the style of programming used in Lava for structural hardware design [2] can be applied to data-parallel programming as well. Therefore Obsidian programmers use combinators that have much in common with those used in Lava. However, where Lava generates the netlist for a fixed-size circuit, Obsidian can generate GPU programs that are parametric in input size.

2 GPGPU and Data-parallel programming

GPUs designed to produce the fast-paced graphics in modern games are now interesting for general purpose computations as well. GPUs are designed for graphical computations of highly data-parallel nature. In comparison to CPUs (Central Processing Units), GPUs devote more of their transistor budget to computation, where CPUs need to devote much effort to extracting instruction-level parallelism [14]. The GPGPU (General-Purpose Computations on the GPU) field is driven by the desire to use the computational power of GPUs for general-purpose computations.

GPUs have been successfully applied to several areas such as physics simulation, bioinformatics and computational finance [15]. Sorting is another area where there are success stories [18, 12].

2.1 The NVIDIA 8800 GPU

The NVIDIA 8800 GPU is described as "a set of SIMD multiprocessors" in the CUDA Programming Manual [7]. Each of the multiprocessors in the set consists of 8 SIMD processing elements and the high end GPUs in the 8800 series have 16 such multiprocessors, giving a total of 128 processing elements.

On each of these groups of 8 SIMD processing elements a number of threads can be executed. Such a group of threads is called a thread *block*. Each of the threads in a block is executing an instance of the same program. Up to 512 threads can be executing within a block. A block is divided into smaller groups that are executed in a SIMD fashion; these groups are called *warps* [7]. This means that within a warp, all threads are progressing in lock-step through the program. There is a scheduler that periodically switches warps. However, between warps; SIMD fashion of execution is not maintained, thus thread synchronisation primitives are needed.

3 Programming in Obsidian

Obsidian can be used to describe computations on arrays. Currently there are some limitations on what computations can be described. As an example the computations must be length homogeneous, that is the input and output arrays must be equal in length. Also programs are currently limited to working only with integers. Another limitation is that currently the generated code can operate on arrays of length up to 512 elements. This is because the generated code is run in one block of threads, with at most 512 threads, and each thread is only operating on one element of the array.

The first example program reverses an array and adds one to each element:

```
rev_incr = rev ->- fun (+1)
```

Here `rev` is an index permutation and `fun` applies a function to each element of the array. The combinator `->-` composes its two arguments into one operation, by feeding the outputs of the first into the inputs of the second. The `rev_incr` program can be run on the GPU using the `execute` function or it can be run on the CPU using the `emulate` function:

```
*Obsidian> emulate rev_incr [1..10]
[11,10,9,8,7,6,5,4,3,2]
```

The functions `execute` and `emulate` both take an Obsidian program and a Haskell list as arguments. The C program is generated and the Haskell list is turned into a C array. The resulting C program is then passed to the NVIDIA compiler and turned into GPU code or, in the emulation case, into code for the CPU. The compiled program is executed and the result read back into the Haskell system and presented.

The C code generated from the previous `rev_incr` program looks as follows:

```
__global__ static void rev_incr(int *values, int n)
{
  extern __shared__ int shared[];
  int *source = shared;
  int *target = &shared[n];
  const int tid = threadIdx.x;
  int *tmp;
  source[tid] = values[tid];
  __syncthreads();
  target[tid] = (source[((n - 1) - tid)] + 1);
  __syncthreads();
  tmp = source;
  source = target;
  target = tmp;

  __syncthreads();
  values[tid] = source[tid];
}
```

In this program, the general structure of a program generated by Obsidian is visible. First the array is loaded into shared memory. This is followed by a CUDA `__syncthreads()` statement making sure the entire array is loaded into the shared memory. When the shared memory is set up, the computation described in the Obsidian program commences. The generated C code then ends with another `__syncthreads()` and the storing of the computed results into the array.

An Obsidian program will bear much resemblance to the corresponding Lava program. In some cases the Lava and Obsidian descriptions will be identical. As an example, here is the description of the *shuffle exchange network*:

```
shex n f = rep n (riffle ->- evens f)
```

In the definition of `shex`, the combinators `rep` and `evens` are used along with the index permutation `riffle`. `rep` repeats a program a given number of times and `evens` applies a two-input two-output function to each even numbered input and its direct neighbour. Figure 1 shows a visual representation of the shuffle exchange network.

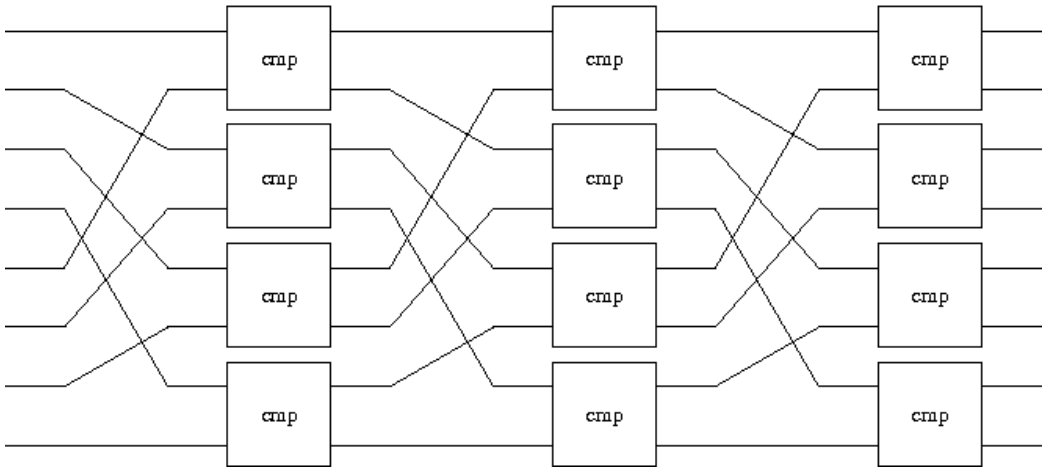


Figure 1: Shuffle exchange network

In Lava as well as Obsidian, `shex` declared above defines the shuffle exchange network. However, the above program will generate a GPU program which is not parametric in the length of the input array, much as Lava generates a netlist of fixed size. This can be fixed by using a different repeat combinator called `repE` that takes an expression as argument instead of an integer:

```
pshex f arr = let n = log2i (len arr)
              in  repE n (riffle ->- evens f) arr
```

This parametric shuffle exchange network will work for any array of length a power of 2. Both `rep` and `repE` result in a **for loop** in the generated C program.

The following example shows the kind of C code that is generated from an Obsidian program using the combinator `repE`. The example program reverses an array as many times as the array is long and hence it is the identity on even length arrays while it reverses arrays of odd length.

```
revs arr = let n = len arr
           in  repE n rev arr
```

The code generated from the example above looks as follows:

```
__global__ static void revs(int *values, int n)
{
  extern __shared__ int shared[];
  int *source = shared;
  int *target = &shared[n];
  const int tid = threadIdx.x;
  int *tmp;
  source[tid] = values[tid];
  __syncthreads();
  for (int i0 = 0;(i0 < n);i0 = (i0 + 1)){
    target[tid] = source[((n - 1) - tid)];
    __syncthreads();
    tmp = source;
    source = target;
    target = tmp;
  }
  __syncthreads();
  values[tid] = source[tid];
}
```

In Lava, it is common to define circuits recursively. Here is an example showing a butterfly network:

```
bfly 1 f  = f
bfly n f  = ilv (bfly (n-1) f)  ->- evens f
```

Figure 2 shows the butterfly network (which forms the merger in Batcher's well-known *bitonic sort* [1]). At the moment there is no good way of dealing with programs such as `bfly`, in Obsidian. Defining an Obsidian function recursively can lead to C programs with deeply nested conditionals, which is very bad for the performance on the target platform. A way to deal with recursive structures is definitely needed and will be explored as future work.

Since we cannot at the moment generate code for recursive structures, the second example program will be a *periodic sorter*, instead of a recursive one. A periodic sorter works by repeatedly applying a so-called periodic merger to the input. The first component needed is a two-sorter. A two-sorter, here called `cmpSwap`, is a two-input two-output function that sorts its two inputs onto the outputs:

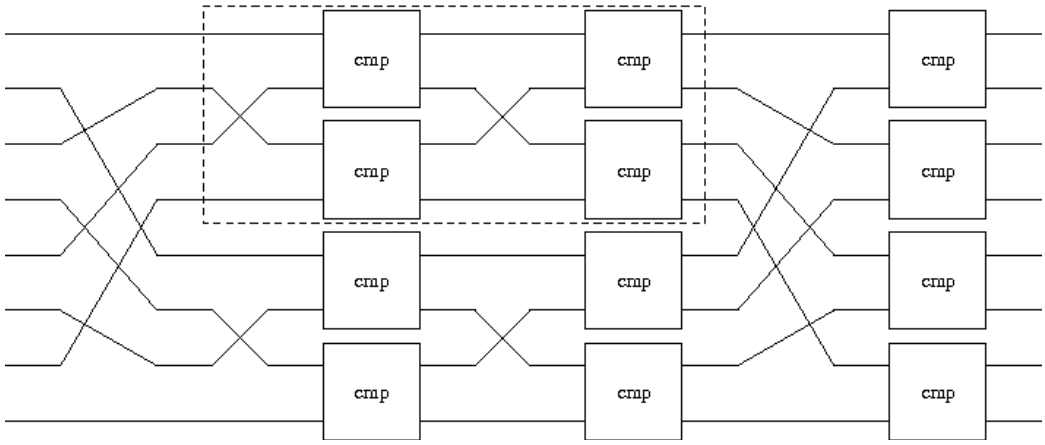


Figure 2: Butterfly network

```
cmpSwap op (a,b) = ifThenElse (op a b) (a,b) (b,a)
```

Now the shuffle exchange network can be used to define a bitonic merger:

```
bmergeIt = pshex (cmpSwap (<*))
```

For a given component, such as `cmpSwap`, the functional behaviour of the shuffle exchange network is equivalent to that of the butterfly described above. This can be shown by induction, see for example reference [17]. An earlier Lava paper also showed how to exploit the *zero one principle* in automatic proofs about fixed-size sorting networks [6]. Here, we use that idea (and Lava) to show that the bitonic merger and the merger made using the shuffle exchange network have identical behaviour for 8 inputs. The following property states that using the the component `cmpSwapB`, which is a two-sorter on boolean values, the shuffle exchange network and the butterfly network produce the same output given the same input:

```
prop_shex_bfly =
    forAll (list 8) $ \xs ->
        shex 3 cmpSwapB xs  <==> bfly 3 cmpSwapB xs
```

Now this can be verified using Lava and SMV:

```
Main> smv prop_shex_bfly
Smv: ... (t=0.01system) \c
Valid.
```

Below are two test runs using `bmergeIt`. The first input shown is a bitonic sequence (with first half increasing and second half decreasing,) resulting in sorted output. The second input is not a bitonic sequence and the result remains unsorted.

```
*Obsidian> execute pbmergeIt [1,3,5,7,8,6,4,2]
[1,2,3,4,5,6,7,8]

*Obsidian> execute pbmergeIt [1,7,4,2,6,8,3,5]
[1,2,3,7,4,5,6,8]
```

The next question is how to make a periodic merger. Composing several bitonic mergers in sequence does not give a sorter (as the reader might like to verify). However, Dowd et al have not only introduced the periodic balanced merger, but also shown how it relates to the shuffle exchange (or omega) network [8].From the τ permutation that they introduce, we can derive a related permutation, here called `tau1`, that when composed with a bitonic merger gives a network that behaves identically to the balanced periodic merger.[1] Figure 3 shows the permutation defined by tau1 schematically. The `tau1` index permutation is defined as follows:

```
tau1 = unriffle ->- one rev
```

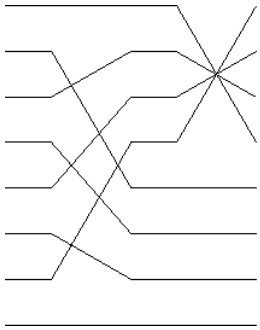


Figure 3: Index permutation defined by `tau1`

Combining `tau1` and bmergeIt results in a merger that has the same behaviour as the balanced periodic merger [8]. Figure 4 shows this merger.

```
dmergeIt = tau1 ->- bmergeIt
```

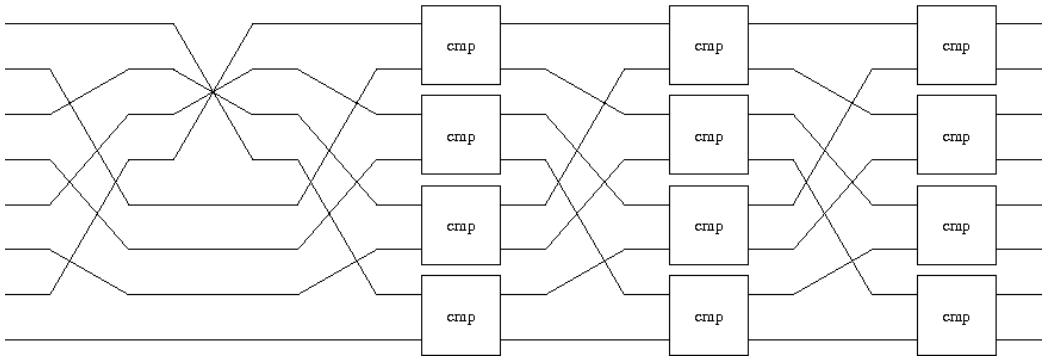


Figure 4: Periodic merger

The sorter called *iterative vsort* is a periodic sorter made from this merger. It is implemented by repeatedly applying `dmergeIt` to the input array.

```
vsortIt arr =
    let n = log2i (len arr)
    in (repE n dmergeIt) arr

*Obsidian> execute vsortIt [8,1,4,2,3,6,7,5]
[1,2,3,4,5,6,7,8]
```

This section showed how programs are written using Obsidian and also what the generated C code looks like.

4 Results

The experiements shown in this section was performed using an earlier version of Obsidian. In that version, a single Obsidian sync point resulted in two `__synchthreads()` in the generated code. The figures presented here are obtained using the following hardware:

```
CPU: Intel Core 2 Duo 2,4GHz
GPU: NVIDIA 8800 GTS 1.2GHz
```

[1]Thanks to Eva Suci (Berrgen) for pointing out that Dowd et al contained exactly the information we needed to be able to make a completely iterative description of a periodic sorter

The dataset used in the tests was 288MB of random data. This dataset was split into batches of 512 32bit elements. Each batch of 512 elements was then sorted individually. Figure 5 shows the running time of a number of sorters generated from Obsidian descriptions as well as a sorter running on the CPU and one implemented directly in CUDA. The GPU sorters are running on one of the multiprocessors, using 8 SIMD processing elements. All running times where obtained using the Unix `time` command. Below are short descriptions of all the sorters used in the comparison:

bitonicCPU is an implementation of bitonic sort for the CPU. The implementation is adapted from one shown in [16].

sortOET Odd Even Transposition sort, is a periodic sorter with depth n.

vsortIt is similar to the sorter described previously, but it is optimized using tables that represent the `tau1` permutation. For 2^n inputs it has depth n^2

vsortIt2 is the same sorter as the above, but with an extra sync inserted into its shuffle exchange network.

vsortHO is a hand optimised version of *vsort*. This version is actually quite different from the different versions of *vsort* generated by Obsidian. In it each swap operation is done by a single thread.

bitonicSort is the implementation of bitonic sort supplied by the CUDA SDK. For 2^n inputs it has depth $n(n+1)/2$.

The chart in figure 5 shows that it is possible to generate a sorter using Obsidian that is close in performance to its hand optimised counterpart. The difference in performance between the very similar sorters *vsortIt* and *vsortIt2* is a result of the much less complicated expressions in the latter.

The most efficient sorter in the comparison is the CUDA Bitonic sort. The difference in running time between *bitonicSort* and *vsortIt2* is explained by the difference in depth between them. The sorter called *vsortIt2* is deeper and therefore slower. For example, for 512 inputs *vsortIt2* has depth 81 while *bitonicSort* has depth 45. This indicates that the generated code is acceptably efficient.

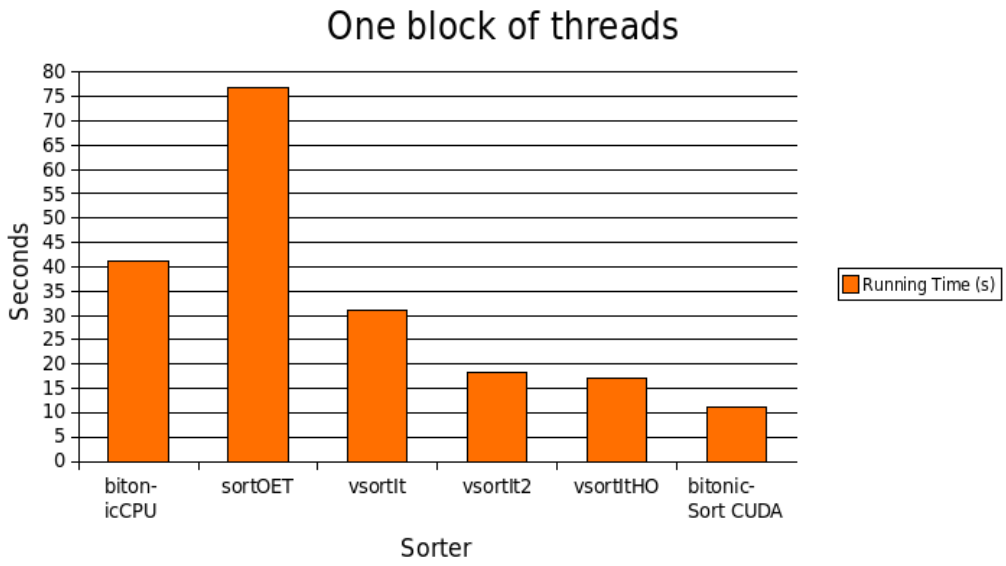


Figure 5: Running time measurements using one block

5 Implementation of Obsidian

Using Obsidian it is possible to describe operations on arrays. To represent an array in Obsidian the type `Arr a` is used:

```
data Arr a = Arr (IxExp -> a, IxExp)
```

An array in Obsidian is a tuple of an indexing function and an expression representing the length of the array. There are a number of helper functions for working with arrays, `mkArray`, `len` and !, for creating new arrays, getting the length of a given array and indexing in an array. These functions are mostly used inside of the library and not by the application writer.

One of the basic operations on arrays provided by Obsidian is the `rev` function. Its use has been shown in the previous section about programming. In the library `rev` is implemented as follows:

```
rev :: Arr a -> W (Arr a)
rev arr =
    let n = len arr
    in  return $ mkArray (\ix -> arr ! ((n - 1) - ix)) n
```

This function takes an array and returns a new one with the indexing function transformed. The type of `rev` is monadic, the monad `W` is a variant of the Writer monad with functionality added for generating loop variables. All Obsidian functions are monadic for consistency even though a function such as rev makes no use of the functionality the `W` monad provides.

To be able to generate C Code from an Obsidian program, a representation of C code is at certain points written into the `W` monad. We call these points *sync* points. These sync points are expressed in Obsidian using the `sync` function:

```
sync :: Syncable a => Arr a -> W (Arr a)
```

When a sync is reached an object of type `IxExp -> AbsC` is written into the W Monad. This function is later applied to an index expression representing a thread's thread Id. The result, `AbsC`, of this application is then used to generate the CUDA C code.

A sync point can be inserted into a program as follows, using the shuffle exchange network example from earlier:

```
shex n f = rep n (riffle ->- sync ->- evens f)
```

The operations before and after the `sync` are now sequenced in the generated code and no longer composed into one operation executed in parallel on the target platform. In some cases this insertion of an extra `sync` can have beneficial effect on the performance of the generated code. This is probably due to the simpler expressions returned by `sync`. Perhaps this effect will diminish when we later introduce optimisation of the generated expressions. Currently no optimisation of the generated code is performed; this will be explored as future work.

Let us return to the `rev_incr` example from the previous section. When passing the program `rev_incr` to `emulate` or `execute`, a `sync` is added to it. So in reality the `rev_incr` program is defined as:

```
rev_incr2 = rev ->- fun (+1) ->- sync
```

When using `execute` or `emulate` the two programs `rev_incr` and `rev_incr2` are the same. Each of the `execute` and `emulate` functions analyses its input and decides whether or not to insert a `sync` statement.

Sync points can also be added automatically in the combinators `rep` and `repE`. This is essentially making the program `rep 3 rev` and `rep 3 (rev ->- sync)` equivalent.

To generate the C code from an Obsidian program the first step is to apply the Obsidian program to a symbolic array defined as follows:

```
symArray :: Arr (Exp Int)
symArray = (\ix -> (index (variable ‘‘source’’ Int) ix),
            variable ‘‘n’’ Int)
```

If all goes well an object of type `IxExp -> AbsC` is created. This object is applied to an index expression representing thread id:

```
threadID = variable ‘‘tid’’ Int
```

The `AbsC` type describes the abstract syntax of a subset of C. For example it contains for loops, if statements, variable declarations and assignments. From this representation a CUDA C source file is generated.

6 Related Work

This project touches a number of different areas, such as embedded languages, data-parallel programming and the GPGPU area.

Lava is an example of an embedded language written in Haskell. It is from Lava that the programming style for Obsidian is derived. In Lava combinators are used to describe hardware [2].

Pan is an embedded language for image synthesis developed by Conal Elliot. Because of the computational complexity of image generation, C code is generated. This C code can then be compiled by an optimising compiler. Pan is described in the paper [9]. Many ideas from the paper “Compiling Embedded Languages”, describing the implementation of Pa,n were used in the implementation of Obsidian [11].

The two languages above are those that had a more direct impact on this project. The programming style using combinators has much in common with Lava, while the implementation is in debt to Pan.

NESL is a functional data-parallel language developed at Carnegie Mellon university. NESL offers a kind of data-parallelism known as nested data-parallelism. Nested data-parallelism allows a parallel function to be applied over nested data structures, such as arrays of arrays, in parallel. NESL is compiled into a intermediate language called VCode that in turn can be used to generate code for numerous parallel architecture [3]. NESL is described in [4].

Data Parallel Haskell takes the ideas from NESL and incorporates them into the Glasgow Haskell Compiler. Data Parallel Haskell adds a new built-in type of parallel arrays to Haskell. Data parallel programs are expressed as operations on objects of this type. The implementation of Data Parallel Haskell is not complete, but is showing promise [5].

In both NESL and Data Parallel Haskell, the data-parallel programming model is implemented in a functional setting. Both implement nested data parallelism.

PyGPU is a language for image processing embedded in Python. PyGPU uses the introspective abilities of Python and is in that way bypassing the need to implement new loop structures and conditionals for the embedded language. In Python it is possible to access the bytecode of a function and from that extract information about loops and conditionals [13]. Programs written in PyGPU can be compiled and run on a GPU.

Vertigo is another embedded language by Conal Elliot. Vertigo is a language for 3D graphics that targets the DirectX 8.1 shader model. Vertigo can be used to describe geometry, shaders and to generate textures. Each sublanguage is given formal semantics [10]. From programs written in Vertigo assembly language programs are generated for execution on a GPU.

Like Obsidian, PyGPU and Vertigo generate code that can be run on GPUs. Though PyGPU and Vertigo are aimed at graphics applications, not GPGPU applications as Obsidian.

7 Future Work

Currently the generated code operates on array with a maximum length of 512 elements. Breaking this barrier in combination with exploiting the full

GPU is high priority for future work.

Even though the results show that the generated code is reasonably efficient, it must be said that implementing the efficient sorter presented in the results demdanded much care in choosing what combinators to use and exactly how they should be implemented. Optimising the generated code might result in less work needed from the application implementor in order to assure efficiency. At the moment no optimisation of the generated code is performed. There are a number of techniques for optimising expressions similar to those generated by Obsidian in "Compiling Embedded Languages" [11].

Recursive Obsidian functions are not recommended, as the expressions generated by such functions tend to be large. Because of this, Obsidian is lacking some expressive power. In the future some control structure to replace recursion is needed. A possible approach is to use combinators that capture common recursive patterns, such as a divide-and-conquer combinator.

Another possible path to investigate in the future is nested data parallelism. Nested data parallelism allows the implementation of divide-and-conquer algorithms [4]. Perhaps this would offer a solution to the previously stated shortcoming as well. It looks as though a limited form of nested data parallelism could be implemented rather easily using the "block of threads" structure supported by CUDA. However, that direct approach would probably mean that the kinds of parallel functions that can be applied to a nested data structure are limited.

8 Acknowledgement

This research is supported by the Swedish Basic Research Agency, Vetenskapsrådet. We would also like to thank our colleague Ulf Assarsson for giving us an introduction to GPU programming.

References

[1] Kenneth E. Batcher. Sorting networks and their applications. In *AFIPS Spring Joint Computing Conference*, pages 307–314, 1968.

[2] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: Hardware Design in Haskell. In *International Conference on Functional Programming, ICFP*, pages 174–184. ACM, 1998.

[3] G. E. Blelloch and S. Chatterjee. VCODE: A data-parallel intermediate language. In *Proceedings of the 3rd Symposium on the Frontiers of Massively Parallel Computation*, pages 471–480, 1990.

[4] Guy E. Blelloch. NESL: A Nested Data-Parallel Language. Technical Report CMU-CS-93-129, CMU Dept. of Computer Science, April 1993.

[5] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon P. Jones, Gabriele Keller, and Simon Marlow. Data parallel haskell: a status report. In *DAMP '07: Proceedings of the 2007 workshop on Declarative aspects of multicore programming*, pages 10–18, New York, NY, USA, 2007. ACM Press.

[6] Koen Claessen, Mary Sheeran, and Satnam Singh. Using lava to design and verify recursive and periodic sorters. *STTT*, 4(3), 2003.

[7] *NVIDIA CUDA Compute Unified Device Architecture: Programming Guide Version 1.0, www.nvidia.com/cuda.*

[8] Martin Dowd, Yehoshua Perl, Larry Rudolph, and Michael Saks. The periodic balanced sorting network. *J. ACM*, 36(4):738–757, 1989.

[9] Conal Elliott. Functional images. In *The Fun of Programming*, Cornerstones of Computing. Palgrave, March 2003.

[10] Conal Elliott. Programming graphics processors functionally. In *Proceedings of the 2004 Haskell Workshop*. ACM Press, 2004.

[11] Conal Elliott, Sigbjørn Finne, and Oege de Moor. Compiling embedded languages. *Journal of Functional Programming*, 13(2), 2003.

[12] Alexander Greß and Gabriel Zachmann. GPU-ABiSort: Optimal parallel sorting on stream architectures. In *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Rhodes Island, Greece, 25–29 April 2006.

[13] Calle Lejdfors and Lennart Ohlsson. Implementing an embedded gpu language by combining translation and generation. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 1610–1614, New York, NY, USA, 2006. ACM.

[14] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krger, Aaron E. Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[15] Matt Pharr and Randima Fernando. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley Professional, 2005.

[16] Robert Sedgewick. *Algorithms in C: Parts 1-4, Fundamentals, Data Structures, Sorting, and Searching*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[17] Mary Sheeran. Sorts of butterflies. In *IVth Workshop on Higher Order (Banff,1990), ed. Birtwistle*, Workshops in Computing. Springer, 1991.

[18] Erik Sintorn and Ulf Assarsson. Fast parallel gpu-sorting using a hybrid algorithm. In *First Workshop on General Purpose Processing on Graphics Processing Units*, October 2007.

Parametric Verification of Industrial Cache Protocols

MURALI TALUPUR SAVA KRSTIĆ JOHN O'LEARY MARK R. TUTTLE

Strategic CAD Labs, Intel Corporation
Contact: murali.talupur@intel.com

Distributed protocols like cache coherence protocols form the bedrock on which modern multi-processor systems are built. Such distributed protocols are typically designed *parametrically*, that is, independent of the precise number of processors involved. Given that distributed programs are hard to reason about for humans and that no amount of testing/simulation can cover all scenarios, it becomes necessary that we find methods to formally and parametrically verify the correctness of such systems.

In this talk we will relate our practical experience with parameterized verification of an on-die cache coherence protocol for a many-core microprocessor design in progress at Intel. The protocol contains complexity that is not present in standard examples such as the FLASH or German protocols. To give an idea: the standard academic version of the German protocol has 7 different messages [4]; the FLASH protocol has 16 different messages and only 2 or 3 methods have ever been successful in verifying it parametrically. The Intel protocol with 54 different types of messages is vastly more complex. Moreover, the number of caching agents in the systems we are interested in is large enough to make parameterized verification a must: we found that conventional (non-parameterized) model checking techniques ran out of gas when confronted with more than four or five agents.

The verification technique we used is based on a method first described by McMillan [3] and subsequently elaborated by Chou, Mannava and Park [1] and Krstić [2]. This method, which we call the CMP method, is based on circular compositional reasoning and uses model checkers as proof assistants. Briefly, a parameterized system containing a directory and N caching agents is abstracted to a system containing a directory, two caching agents, and a third, highly nondeterministic, process representing "all the other agents". The user then supplies a series of lemmas that refine the behavior of the third agent; these lemmas are used mutually to prove one another and also the final property of interest. Coming up with these lemmas is a time-consuming process requiring a deep understanding of the protocol. It took us about a month and 25-odd lemmas to prove the cache coherence of the protocol. As far as we are aware this is the first time a protocol of this size and complexity has been verified parametrically.

The next step of the project was to make the CMP method easier to use by automating as much of it as possible. The method has three stages: (i) creating the inital abstraction, (ii) running the model checker and coming up with lemmas after examining counterexample traces, and (iii) refining the abstract model in light of the new lemmas. While discovering the lemmas requires ingenuity, the other two parts can be automated. We have built a tool that creates an initial abstraction and refines the abstract model with user-supplied lemmas automatically, and our talk will also describe the principles behind this tool.

The most important limitation of the CMP method is that it does not deal with systems in which processes are ordered by their indices. Commonly occurring algorithms break symmetry between processes by ordering them either linearly (as in the bakery algorithm), or placing them on a ring, grid or other network topology. The third element of our talk will explain our ongoing work to extend the CMP method to handle such asymmetric systems as well. Considering asymmetric systems led us to discover what we believe are new circular compositional reasoning principles. Besides enabling the CMP method to handle asymmetric systems, we anticipate that these new principles will also allow to formulate the CMP method itself much more intuitively and succinctly.

References

[1] C.-T. Chou, P. K. Mannava, and S. Park. A simple method for parameterized verification of cache coherence protocols. In A. J. Hu and A. K. Martin, editors, *Proc. Conf. on Formal Methods in Computer-Aided Design (FMCAD04)*, volume 3312 of *Lecture Notes in Computer Science*, pages 382–398. Springer, 2004.

[2] S. Krstić. Parameterized system verification with guard strengthening and parameter abstraction. In *Fourth Int. Workshop on Automatic Verification of Finite State Systems*, 2005. To appear in *Electronic Notes in Theoretical Computer Science*.

[3] K. L. McMillan. Parameterized verification of the FLASH cache coherence protocol by compositional model checking. In *Correct Hardware Design and Verification Methods (CHARME'01)*, volume 2144 of *LNCS*, pages 179–195, 2001.

[4] A. Pnueli, J. Xu, and L. D. Zuck. Liveness with (0, 1, ∞)-counter abstraction. In *CAV*, pages 107–122, 2002.