Formal Specification and Verification of a Dataflow Processor Array*†

Thomas A. Henzinger Xiaojun Liu Shaz Qadeer Sriram K. Rajamani
EECS Department, University of California at Berkeley, CA 94720-1770, USA
Email: {tah,liuxj,shaz,sriramr}@eecs.berkeley.edu

Abstract. We describe the formal specification and verification of the VGI parallel DSP chip [1], which contains 64 compute processors with ∼30K gates in each processor. Our effort coincided in time with the "informal" verification stage of the chip. By interacting with the designers, we produced an abstract but executable specification of the design which embodies the programmer's view of the system. Given the size of the design, an automatic check that even one of the 64 processors satisfies its specification is well beyond the scope of current verification tools. However, the check can be decomposed using assume-guarantee reasoning. For VGI, the implementation and specification operate at different time scales: several steps of the implementation correspond to a single step in the specification. We generalized both the assume-guarantee method and our model checker MOCHA to allow compositional verification for such applications. We used our proof rule to decompose the verification problem of the VGI chip into smaller proof obligations that were discharged automatically by MOCHA. Using our formal approach, we uncovered and fixed subtle bugs that were unknown to the designers.

1 Introduction

The VGI chip [1] is an array of DSP processors designed to be part of a system for web-based image processing [2]. The VGI chip contains a total of 96 processors and has approximately 6M transistors. Of the 96 processors, 64 are identical 3-stage pipelined compute processors. Each compute processor has about 30,000 logic gates. Data is communicated between the processors by means of FIFO queues. No assumption is made about the relative speeds at which data is produced and consumed in the processors. Hence, to transfer data reliably an elaborate handshake mechanism is used between the sender and the receiver. In addition, the interaction between the control of the pipeline and the control of the communication unit is quite complex.

In this work, we focus on the verification of the 64 compute processors and the communication between them. A single processor was designed partly in VHDL and partly in circuit schematics. We translated the description into the language of Reactive Modules [3], which is the input language to our model checker MOCHA [4]. After a number of discussions with the designers, we produced a formal specification of the design which embodies the programmer's view of the system, also in Reactive Modules. The sheer size of the design together with the well-known state explosion problem precluded the direct use of model checking techniques to verify the implementation against the specification. Existing techniques that flatten the design hierarchy and use BDD-based state exploration [5] can verify designs with at most 50–100 latches reliably. Clearly, the VGI design, which has about 800 latches per compute processor, is well beyond the scope of such tools. We demonstrate how model checking can be scaled up using assume-guarantee reasoning to handle the VGI design. To the best of our knowledge, the largest design that has been ever verified using model checking has been reported by Eiríksson [6]. Compositional techniques used in that effort for decomposing the verification task did not readily apply to the VGI, because the implementation and specification operate on different time scales (several consecutive implementation steps realize a single specification step). We developed novel compositional techniques for decomposing refinement proofs with variable time scales. We

*An abbreviated version of this paper appeared in the *Proceedings of the IEEE/ACM International Conference on Computer-aided Design* (ICCAD 1999), pp. 494–499.

†This research was supported in part by the National Science Foundation CAREER award CCR-01708, by the Defense Advanced Research Projects Agency grant NAG2-1214, and by the Semiconductor Research Corporation contracts 324.041 and 683.003.

then applied these techniques to obtain proof obligations that were small enough to be discharged automatically by MOCHA. In the process, we found several subtle bugs that were unknown to the designers. Three of these bugs will be explained in the discussion in Section 5.

Step 1: formal specification. A significant part of the verification effort was invested in producing a correct specification. Only an informal specification of the design existed in the form of English description and elaborate timing diagrams. The fact that no behavioral description of the design was available (the datapath was designed directly in schematic) made the task of producing the specification even more difficult.

A number of features are desirable in the specification for the VGI chip. First, the specification should be at a level of abstraction such that a high degree of confidence in its correctness can be established by informal means such as code review. Specifically, the specification should embody the view that the programmer/compiler has of the VGI chip, which is that of a dataflow architecture with a set of processing elements connected through queues. For this high-level view, every processing element behaves as if each instruction is executed atomically in one step, and the communication between the processors behaves like FIFO queues. The behavior of a program written with this high-level view should not depend on the delay in transferring a data token from one processor to another. Such FIFO queues can be modeled using nondeterministic delay. This makes necessary the availability of nondeterminism in the specification language.

Second, the specification should have an operational as well as a mathematical semantics. Operational semantics permits the execution of specifications; mathematical semantics permits their formal verification. Executability is especially desirable in the case of the VGI processor because the design is part of a bigger system. If all essential features of the design that are necessary for correct interaction with the environment have been captured by the specification, it can be used in place of the actual design for simulating the system.

Third, the design itself (the "implementation") should be describable in the same language as the specification, and a refinement operator should be available for relating the implementation and the specification. In our case, the refinement operator must relate two different time scales. The implementation has a clock signal `clk` with activity on both the `HIGH` and `LOW` phases in different parts of the design. For instance, in the execute phase of the pipeline a bus carries an operand when `clk` is `HIGH` and the result when `clk` is `LOW`. But the specification does not mention `clk` at all. In fact, the whole computation happens in just one step. Thus, one round in the specification is equal to two rounds in the implementation, one with `clk` = `HIGH` and one with `clk` = `LOW`. Therefore, our formal notion of refinement samples the implementation whenever `clk` is `LOW` and checks if the sampled behavior is present in the specification.

Reactive Modules, our modeling language for both specification and implementation, has all the desirable features mentioned above — mathematical semantics, executability, and support for nondeterminism and sampling.

Step 2: formal verification. Since VGI is a very big design, model checking cannot be applied directly. Previously, assume-guarantee methods have been developed for decomposing a refinement verification task into smaller proof obligations that can be discharged automatically with a model checker. In assume-guarantee reasoning [3, 7, 8, 9, 10], the different components of the implementation are verified in isolation by making appropriate assumptions about their environments. The environment assumptions are then discharged separately. In order to keep the sizes of the individual proof obligations within the capacity limits of model checking, it is essential to specify the environment assumptions for implementation components abstractly in terms of specification signals, using "abstraction modules" [10] (also called "refinement maps" [9]).

In the case of VGI, the specification describes the behavior of the implementation only at the sampling instants. Consequently, the abstraction modules specify the values of implementation signals only at those instants. But the correct behavior of implementation components may depend on assumptions about the environment between sampling instants. Hence, for carrying out refinement-based proofs in situations where the time scales of the implementation and specification differ, we (1) introduce a new sampling operator that can sample the signal values of a module with some environment constraint between sampling instants, and (2) generalize the assume-guarantee proof rule to work with the sampling operator. The details of the generalized proof rule can be found in [11]; here we demonstrate its efficacy in verifying the VGI chip. Working with specifications at an abstract level of temporal granularity is not new. While a processor pipeline takes several steps to execute an intruction, its ISA specification executes an instruction atomically in a single step, and the pipeline state can be related to the ISA state by an abstraction function that uses the "pipeline flush-

ing" operation [12]. Clock abstraction on dynamic switch-level circuits [13, 14] generates gate-level circuits without clocks to make their verification easier. Temporal abstraction hierarchies [15] have been used for efficient state space exploration. However, we are not aware of any compositional refinement checks between implementations and specifications that operate at different time scales.

In order to handle the proof obligations that are generated by our new assume-guarantee rule, we extended the model checker MOCHA with the capability for dealing with the sampling operator in refinement checks. We are not aware of any other model checker that currently offers such a capability. Using the enhanced version of MOCHA we discovered several bugs in the VGI design and fixed them. In this process, we found it extremely useful to employ MOCHA as a debugging tool that supports the concurrent activities of (re)design and formal (re)verification: design insights would suggest the definition of refinement maps for model checking, and MOCHA would produce error traces that suggest corrections to the design. In this way, design and formal verification become a single activity ("formal design") that involves similar mental processes, rather than two decoupled activities, one followed by the other with little interaction.

2 The Problem

A compute processor in the VGI chip has an instruction memory, a register file containing three register pairs, a 3-stage pipelined datapath, a control unit, and three data output buses and one control output bus for sending tokens to other processors. Each register pair can be configured either as a queue or as general purpose registers. Each output bus may or may not be connected to another processor. A processor P can send data to another processor Q if a data output bus of P is connected to a register pair of Q that has been configured as a queue. A handshake protocol is used between P and Q for transferring data reliably. There is a programmable interconnection network that allows any processor to be connected to any other processor. In a typical dataflow computation, programs are loaded into the instruction memory of some subset of the set of processors, and the appropriate data connections between the processors are made by programming the network. Each processor with its own program acts as an "actor" in a data flow network, consuming tokens from its input and producing tokens at its output. In any network of compute processors, each processor is in a certain *configuration* depending on the register pairs

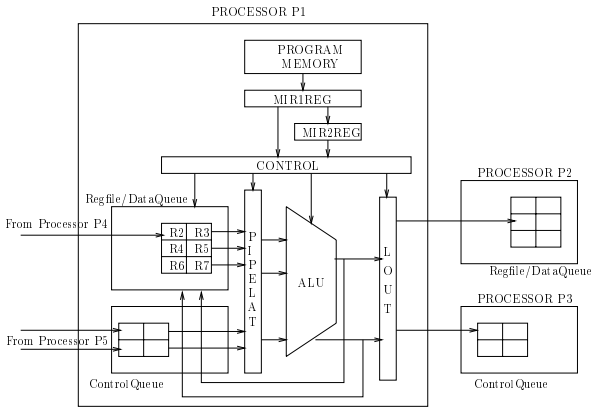

Figure 1: Configuration with three input and two output queues

configured as queues, and the output buses connected to downstream processors. Let $\mathcal{C}_{VGI}$ denote this set of $2^3 \times 2^4 = 128$ configurations. Figure 1 shows a processor configuration where the register pair R2–R3 is configured as a queue, and a data and a control output queue are configured to send out tokens.

Our specification for the processor configuration shown in Figure 1 consists of modules `ISA`, `DataQueue`, and `ControlQueue`.[1] The module `ISA` contains modules such as program memory, register file, control unit, and ALU, and is a specification of the pipelined datapath of processor `P1`. Every instruction gets executed atomically in one round in the `ISA`. The specification for the data output bus of `P1` together with the queue of processor `P2` is a 4-place FIFO buffer `DataQueue`. The four places in `DataQueue` correspond to the two places of the queue of `P2` and the two sets of latches in `lout` and `pipelat`. The module `ControlQueue` is identical to `DataQueue` except for the data width and is the specification for the control output bus of `P1` together with the queue of processor `P3`. Performing verification against the composition of `ISA`, `DataQueue`, and `ControlQueue` will ensure that instructions are executed correctly and data is transferred reliably from `P1` to `P2` and `P3`. We can similarly write specifications for processors `P2` and `P3`. Then the specification for the network of processors `P1`, `P2`, and `P3` can be obtained by composing the specifications of the individual processors. In Figure 2, note that the register pair R2–R3 is missing. Since they have been configured as an input queue, they are part of the distributed output queue of an upstream processor, and is specified in that processor.

[1] The dotted rectangle in the lower portion of Figure 2 shows refinement maps. We defer their description to Section 4.

Our verification methodology, described in the next section, will let us prove that an arbitrary network of compute processors satisfies its specification.

3 The Methodology

We model both implementations and specifications as Reactive Modules [3]. For the purposes of this discussion, a reactive module comprises a finite set of variables, partitioned into *external* (input) and *interface* (output) variables, and rules for initializing and updating their values in each round of operation. Both the initial value and the update of a variable can depend on another variable with zero-delay. These zero-delay dependencies impose a partial order on the evaluation of the variable values in each round. The parallel composition $P \| Q$ of two modules P and Q is obtained by connecting the variables with the same names and is defined only if (1) the set of interface variables of modules P and Q are disjoint, and (2) there is no zero-delay cycle in the composition. If $P \| Q$ is defined, then P and Q are said to be *compatible*. A state s of a module P is an assignment of values to all its variables. A state s is *initial* if it can result from executing the initializing rules of P. We write $s \rightarrow_P t$ if starting from state s, variables of P can be updated according to the update rules of P to reach the state t. A finite sequence $s_0, s_1, s_2, \ldots, s_n$ of states is a *trace* of P if s_0 is an initial state and for all $i < n$, we have that $s_i \rightarrow_P s_{i+1}$. The *trace language* L_P of a module P is the set of all traces of P. Let P be a module and φ a predicate over the variables of P. The *φ-sample* of a trace τ, denoted by τ_φ, is the subsequence of τ obtained by selecting all states of τ that satisfy φ. We say that P *refines* Q, denoted by $P \preceq Q$, if (1) every variable of Q is a variable of P, (2) every interface variable of Q is an interface variable of P, and (3) the trace language of P projected onto the variables of Q is a subset of the trace language of Q.

When we discuss the refinement check $P \preceq Q$, we refer to P as the implementation and Q as the specification. The implementation and specification we are concerned with have been described earlier in Section 2. We would like to prove that the implementation refines the specification in as automatic a way as possible. Two features of the implementation make this verification task specially daunting.

- The implementation consists of a possible maximum of 64 compute processors. Each processor is quite big with around 800 latches and 1700 variables. The sheer size of the implementation precludes a direct use of model checking and makes compositional reasoning essential. In assume-guarantee reasoning, the different components of the implementation can be verified in isolation by making appropriate assumptions about their environments. These environment assumptions must then be discharged separately. A crucial aspect of this decomposition process is the use of "refinement maps." We illustrate this in an abstract setting in the following way. Consider, for example, an implementation that is the parallel composition of two modules P and Q and let P' and Q' be their respective specifications. We would like to verify the modules P and Q one at a time. The environment of P might contain signals that are not present in the specification. Hence, we write abstract definitions of these implementation signals in terms of specification signals in the form of a module R_P and use it along with Q' to construct the environment $E_P = Q' \| R_P$ of P. A similar approach is taken for module Q to generate its environment E_Q. Then, we use the following proof rule [3, 10]:

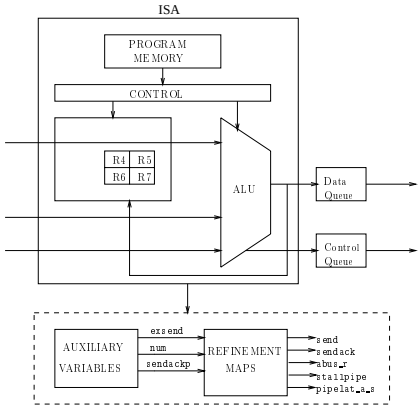

Figure 2: Specification module for refinement check

$$\frac{\begin{array}{lll} P \| E_P & \preceq & E_Q \\ Q \| E_Q & \preceq & E_P \end{array}}{P \| Q \preceq E_P \| E_Q \preceq P' \| Q'} \quad (1)$$

Note that E_P is used in the environment of P to prove E_Q and E_Q is used in the environment of Q to prove E_P. The use of environment assumptions in a circular fashion is crucial for decomposing verification tasks.

- The implementation is based on level-sensitive latches synchronized by a single clock. There are latches of both kind —transparent high and transparent low— and computation is performed in both phases of the clock in different parts of the implementation. Moreover, there are a number of gated latches, i.e., latches whose enabling signals depend on signals

other than the clock. We model these phenomena through an explicit clock variable `clk` that toggles every round. Thus, a round in the implementation corresponds to half a clock cycle. Being at a more abstract level, the specification does not mention the clock at all, and a round in the specification corresponds to two rounds of the implementation. One way to compare an implementation with a specification that operates at a coarser time scale is to sample the values of the implementation signals at appropriate time instants. We would then like to show that every sampled trace of the implementation is a trace of the specification.

Notice that if the implementation and specification have different time scales, the refinement maps will constrain the value of implementation signals only at the sampled time instances. But, sometimes a module in the implementation might depend on the behavior of the environment between sampling points. For example, it might be important that the environment maintains the value of a signal constant from one sampling instant to another. Therefore, the sampling operator might need to constrain the behavior of a module between sampling instants. Let P be a module, T a module compatible with P, and φ a predicate on the variables of module P. Then, we define the following two sampling operators:

- $\texttt{Sample}_{\varphi}(P)$ is a module with the same set of external and interface variables as P, and with the trace language given by the set $\{\tau_{\varphi} | \tau$ is a trace of $P\}$.
- $\texttt{Sample}_{\varphi}(P, T)$ is a module with the same set of external and interface variables as P, and with the trace language given by the set $\{\tau_{\varphi} | \tau$ is a trace of $P \| T\}$.

Note that the module $\texttt{Sample}_{\varphi}(P, T)$ is different from the module $\texttt{Sample}_{\varphi}(P \| T)$. The former module has the same set of interface variables as P while the latter has the same set of interface variables as $P \| T$.

We generalize the assume-guarantee proof rule described above as follows:

$$\frac{\begin{array}{rcl} \texttt{Sample}_{\varphi}(P, T_P) \| E_P & \preceq & E_Q \\ \texttt{Sample}_{\varphi}(Q, T_Q) \| E_Q & \preceq & E_P \\ P \| Q & \preceq & T_P \| T_Q \end{array}}{\texttt{Sample}_{\varphi}(P \| Q) \preceq E_P \| E_Q \preceq P' \| Q'} \quad (2)$$

A formal treatment of the correctness of this proof rule can be found in [11]. The intent behind the first antecedent in the above rule is to prove that $\texttt{Sample}_{\varphi}(P)$ refines E_Q under a "suitable" environment. A suitable environment constrains the inputs to P using the specification component E_P. Since E_P operates at a coarser time scale than P, it can constrain the inputs to P only at the sample points (which are specified by φ). An additional temporal assumption T_P on the inputs to P is needed, which specifies detailed timing assumptions at the finer time scale, about the abstract values supplied by E_P. A similar assumption T_Q is needed to prove that $\texttt{Sample}(Q)$ refines E_P. Finally, it needs to be proved that the implementation $P \| Q$ indeed satisfies the timing assumptions $T_P \| T_Q$. We can further decompose this part of the proof using the assume-guarantee rule in (1) and avoid constructing $P \| Q$. Note that the first two antecedents state a refinement relation at an abstract time scale specified by φ, and the last antecedent states a refinement relation at the detailed time scale.

4 The Proof

Each compute processor in VGI starts a computation in the positive phase of the clock and finishes it in the negative phase of the clock. We decided to sample at the end of each computation. Hence, the sampling predicate φ is `clk` = `LOW`. In the rest of this section, we use φ to refer to `clk` = `LOW`. In Section 2, we showed how to obtain a specification for an arbitrary network of processors. Our goal is to verify that an arbitrary network of processors implements its corresponding specification, using refinement checking. Let $P_1, P_2, \ldots, P_n$ be the compute processors in an arbitrary network, and let $Q_1, Q_2, \ldots, Q_n$ be their respective specifications. For the correct functioning of a processor it is essential that all input signals change only when `clk` is `HIGH`. Let T_i be a module that says that all external signals of P_i change only when `clk` is `HIGH`.

The verification problem is to check

$$\texttt{Sample}_{\varphi}(P_1 \| P_2 \| \ldots \| P_n) \preceq Q_1 \| Q_2 \| \ldots \| Q_n$$

We can apply our new assume-guarantee rule as follows:

$$\frac{\begin{array}{rcl} \texttt{Sample}_{\varphi}(P_i, T_i) & \preceq & Q_i \text{ for all } 1 \leq i \leq n \\ P_1 \| P_2 \| \ldots \| P_n & \preceq & T_1 \| T_2 \| \ldots \| T_n \end{array}}{\texttt{Sample}_{\varphi}(P_1 \| P_2 \| \ldots \| P_n) \preceq Q_1 \| Q_2 \| \ldots \| Q_n}$$

The second antecedent says that the inputs of any processor in the network change only when `clk` is `HIGH`. Since any input to a processor has to be the output of some other processor, this antecedent can be discharged easily by proving that for all $1 \leq i \leq n$,

the outputs of P_i change only when clk is HIGH. This is an easy proof local to each processor and computationally trivial. In the first antecedent, there are n symmetric proof obligations, one for each P_i. For $X \in \mathcal{C}_{VGI}$, let Y be its specification and T_X be the environment constraint that says that all inputs change only when clk is HIGH. If we prove $\texttt{Sample}_{\varphi}(X, T_X) \preceq Y$ for each $X \in \mathcal{C}_{VGI}$, then we have proved that $\texttt{Sample}_{\varphi}(P_i, T_i) \preceq Q_i$ for all $1 \leq i \leq n$. Thus, we have decomposed the proof of an arbitrary network of compute processors to $|\mathcal{C}_{VGI}|$ proofs about individual processor configurations that have 800 latches each. This is still beyond the scope of monolithic model checking. We show how to discharge this proof for a single processor configuration, with further applications of the generalized assume-guarantee rule described earlier. We implemented support for the Sample operator in MOCHA, in order to carry out this refinement check.

We describe the compositional proof for the configuration in Figure 1 whose specification is given in Figure 2. We describe a compute processor in more detail. The processor has a 3-stage pipeline — the fetch stage IF, the execute stage EX, and the communicate stage COM, with pipelat latches between IF and EX, and lout latches between EX and COM. There is feedback from the EX stage to the IF stage. The IF stage is controlled by mir1reg and fetches data from the input queues, the register file, or the feedback. The signal stallempty is asserted if an instruction wants to read from an input queue that is empty. The EX stage contains the ALU and is controlled by mir2reg, a delayed version of mir1reg. The output of the ALU abus_r can be written back to the register file or sent out on one or more queues. For receiving data/control tokens, the downstream processor should have a register pair configured as a 2-place queue. Every data or control token that is computed is latched into lout. If the first send fails, then the COM stage keeps on sending the data in lout until the send succeeds. Signals send and sendack are used for handshake between the sender and the receiver. In the meantime, other instructions might be executing in the EX stage of the pipeline. The pipeline is stalled and a signal stallpipe asserted when the COM stage is trying to send a token and the instruction in the EX stage also wants to send out a token. The invariant that synchronizes the operation of the ISA and the pipeline is that the instruction being executed by the ISA is the instruction in the IF stage of the implementation.

To decompose the proof, we wrote refinement maps for send, sendack, abus_r, stallpipe, and pipelat_a_s as shown in the dotted rectangle in Figure 2. In order to write refinement maps for send and stallpipe, we had to add auxiliary history variables exsend, num, and sendackp. The variable exsend is true whenever the the current instruction in the EX phase wants to send. The variable num keeps track of the number of items in the receiver's 2-place input queue. The variable sendackp predicts the implementation's sendack. The refinement map for abus_r is written in terms of the two stall signals and the output of the ALU in the specification. Using these refinement maps, the proof can be decomposed nicely in the reverse direction of the flow of data in the processor.

1. The output queue is verified using the refinement maps for abus_r, send, and stallpipe. Intuitively, this means that data written into the queue is not lost, no data is written twice, and correct behavior is preserved going into and coming out of stalls (either stallempty or stallpipe).

2. The refinement map for send is verified using the refinement map for sendack.

3. The refinement map for sendack is verified using the refinement maps for stallpipe and send.

4. The refinement map for stallpipe is verified using refinement maps for send and sendack of both the control and data queues.

5. The refinement map for abus_r is verified using the refinement map for the pipelat_a_s signals, which are inputs to the EX stage. Since the bus is generated by the data path of the implementation, this proof amounts to verifying the correctness of the data path. At the time of writing this paper, we have not been able to complete this proof. We believe that this is essentially a combinational verification problem that is amenable to existing techniques geared for combinational verification.

6. The refinement map for pipelat_a_s is verified using the refinement map for abus_r. This lemma amounts to verifying the correctness of feedback from the EX stage to the register file and the pipelat_a_s registers.

In each lemma described above, the part of the implementation under investigation was sampled at clk equal to LOW under some timing assumptions on the inputs between sampling instants. For example, in Lemma 1, it was assumed that the send signal does not change value when clk changes from LOW to HIGH,

and all signals at the receiver end (such as `read` and `save_d`) change values only when `clk` is `HIGH`. All such assumptions were discharged separately. Notice the circular dependencies between Lemmas 1, 2, 3, and 4, and also Lemmas 5 and 6. For Lemmas 2, 3, 4, 5, and 6, we also wrote supporting refinement maps for `mir1reg` and `mir2reg`. These supporting refinements were verified separately. In total, about 35 lemmas needed to be proved. In every lemma except Lemma 5, we used symmetry arguments [16] to reduce the datapath width to just 1 bit. In Lemma 5, the symmetry is broken because of arithmetic operations and hence the full datapath width of 16 bits needs to be considered. Thus, assume-guarantee reasoning provides a clean separation between the verification of the datapath and control of the processor. It is clear in the overall proof that the datapath width is irrelevant in verifying the control that is moving data around. This also suggests that compositional reasoning provides a formal framework under which combinational verification of the datapath and FSM verification of the control can coexist. None of the individual lemmas took more than a few minutes on a 625 MHz DEC Alpha 21164.

5 Discussion

In this section, we describe the bugs we found in the design. We fixed all the bugs and verified our fixes with MOCHA.

1. If the sending processor writes two successive values into the queue and the receiving processor waits for one cycle and then does two successive reads, the second read returns an incorrect value.

2. Suppose `stallempty` is asserted in cycle n but released in cycle $n+1$. Also, suppose send to an output queue fails in cycle $n+1$. Then although `stallpipe` should be asserted in cycle $n+2$, it is not and as a result the instruction in `EX` stage gets clobbered.

3. A particular sequence of events involving 4 sends and 4 reads interleaved in a specific way, with a stall at a precise moment clobbers the data in the `lout` register. This results in the loss of an output token. The error trace that led to the discovery of this bug had ten steps.

We now describe the process by which we found these bugs and the insights we gained about the interaction between design and verification. We found all these bugs while doing the proof of Lemma 1, the lemma stating the correctness of the data transfer between the sender and the receiver. Recall that we needed refinement maps for the environment signals `abus_r`, `send` and `stallpipe`. Initially, we tried to write the refinement maps based on the definitions of these signals in the implementation. But, we got error traces. We kept on strengthening the maps to increasingly constrain the environment until the lemma was proved. At this point, we had correct abstract definitions of these environment signals that we could translate down to definitions in terms of implementation signals. These design fixes were quite complicated and we actually had to do some logic design ourselves. In this way, MOCHA can be used as a debugging tool which tests a proposed design fix by looking at all possible sequences of events. If an error trace is generated then it can be examined to further refine the fix. Thus, the distinction between verifying and designing gets blurred and actually both activities proceed in parallel. We believe that design and verification are symbiotic activities in the sense that the designer's intuition embodied in refinement maps aids verification and the model checker aids the designer by testing that a proposed solution is correct under all possible situations. We believe that the mental processes involved in doing verification exist when the design is being created and therefore, given the right interface to a verification tool, it is not a big burden to do "formal design."

We have shown by our verification of the VGI chip that compositional model checking under the assume-guarantee paradigm can scale to "real" designs. We also believe that it is a general technique not just limited to DSP chips. In our proof, the first step that decomposes the proof obligation on a network of processors to one on a single processor relies on the symmetry inherent in VGI. But the second step involving proof decomposition with the aid of refinement maps is quite general and applicable to a variety of large and complex designs [6, 10, 16].

References

[1] V. Srini, J. Thendean, S. Ueng, and J. Rabaey, "A parallel DSP with memory and I/O processors," in *Proceedings of the SPIE Conference 3452*, pp. 2–13, 1998.

[2] V. Srini and J. Rabaey, "An architecture for web-based image processing," in *Proceedings of the SPIE Conference 3166*, pp. 90–101, 1997.

[3] R. Alur and T. Henzinger, "Reactive modules," in *Proceedings of the 11th Annual Symposium on*

Logic in Computer Science, pp. 207–218, IEEE Computer Society Press, 1996.

[4] R. Alur, T. Henzinger, F. Mang, S. Qadeer, S. Rajamani, and S. Tasiran, "MOCHA: Modularity in model checking," in *CAV 98: Computer Aided Verification* (A. Hu and M. Vardi, eds.), Lecture Notes in Computer Science 1427, pp. 521–525, Springer-Verlag, 1998.

[5] R. Brayton, G. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S.-T. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. Ranjan, S. Sarwary, T. Shiple, G. Swamy, and T. Villa, "VIS: A system for verification and synthesis," in *CAV 96: Computer Aided Verification* (R. Alur and T. Henzinger, eds.), Lecture Notes in Computer Science 1102, pp. 428–432, Springer-Verlag, 1996.

[6] Á. Eíriksson, "The formal design of 1M-gate ASICs," in *FMCAD 98: Formal Methods in Computer-Aided Design* (G. Gopalakrishnan and P. Windley, eds.), Lecture Notes in Computer Science 1522, pp. 49–63, Springer-Verlag, 1998.

[7] E. Stark, "A proof technique for rely/guarantee properties," in *Proceedings of the 5th Conference on Foundations of Software Technology and Theoretical Computer Science*, Lecture Notes in Computer Science 206, pp. 369–391, Springer-Verlag, 1985.

[8] M. Abadi and L. Lamport, "Conjoining specifications," *ACM Transactions on Programming Languages and Systems*, vol. 17, no. 3, pp. 507–534, 1995.

[9] K. McMillan, "A compositional rule for hardware design refinement," in *CAV 97: Computer Aided Verification* (O. Grumberg, ed.), Lecture Notes in Computer Science 1254, pp. 24–35, Springer-Verlag, 1997.

[10] T. Henzinger, S. Qadeer, and S. Rajamani, "You assume, we guarantee: methodology and case studies," in *CAV 98: Computer Aided Verification* (A. Hu and M. Vardi, eds.), Lecture Notes in Computer Science 1427, pp. 440–451, Springer-Verlag, 1998.

[11] T. Henzinger, S. Qadeer, and S. Rajamani, "Assume-guarantee refinement between different time scales," in *CAV 99: Computer Aided Verification* (N. Halbwachs and D. Peled, eds.), Lecture Notes in Computer Science 1633, pp. 208–221, Springer-Verlag, 1999.

[12] J. Burch and D. Dill, "Automatic verification of pipelined microprocessor control," in *CAV 94: Computer Aided Verification* (D. Dill, ed.), Lecture Notes in Computer Science 818, pp. 68–80, Springer-Verlag, 1994.

[13] S. Jain, R. Bryant, and A. Jain, "Automatic clock abstraction from sequential circuits," in *Proceedings of the 32nd Design Automation Conference*, pp. 707–711, 1995.

[14] A. Kuehlmann, A. Srinivasan, and D. LaPotin, "Verity — A formal verification program for custom CMOS circuits," *IBM Journal on Research and Development*, vol. 39, no. 1-2, pp. 149–165, 1995.

[15] R. Alur, T. Henzinger, and S. Rajamani, "Symbolic exploration of transition hierarchies," in *TACAS 98: Tools and Algorithms for Construction and Analysis of Systems* (B. Steffen, ed.), Lecture Notes in Computer Science 1384, pp. 330–344, Springer-Verlag, 1998.

[16] K. McMillan, "Verification of an implementation of Tomasulo's algorithm by compositional model checking," in *CAV 98: Computer Aided Verification* (A. Hu and M. Vardi, eds.), Lecture Notes in Computer Science 1427, pp. 110–121, Springer-Verlag, 1998.