FORMAL SPECIFICATION AND VERIFICATION OF A DATAFLOW PROCESSOR ARRAY

by

Thomas A. Henzinger, Xiaojun Liu, Shaz Qadeer
And Sriram K. Rajamani

Memorandum No. UCB/ERL M99/14

10 March 1999

COVER

FORMAL SPECIFICATION AND VERIFICATION OF A DATAFLOW PROCESSOR ARRAY

by

Thomas A. Henzinger, Xiaojun Liu, Shaz Qadeer and Sriram K. Rajamani

Memorandum No. UCB/ERL M99/14

10 March 1999

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

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 [STUR98], which contains 96 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. 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 [STUR98] is an array of DSP processors designed to be part of a system for web-based image processing [SSTR97]. The VGI chip contains a total of 96 processors and has approximately 6M transistors. Of the 96 processors, 64 are 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.

The design was described partly in VHDL and partly in the form of circuit schematics. We translated the design into the language of Reactive Modules [AH96], which is the input language to our model checker MOCHA [AHM+98]. 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 [BHSV+96] can verify designs with at most 50-60 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 [Eir98]. 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 single a specification step). We developed novel compositional techniques for decomposing refinement proofs with variable time scales. We 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. This and 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. More 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 circuitry between the processors 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 under consideration is part of a bigger system. Provided 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 whole 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 [Sta85, CLM89, GL94, AL95, AH96, McM97, HQR98], 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 so-called "abstraction modules" (refinement maps, simulation relations) [AL88, Kur94, Lyn96, AH96, McM97, HQR98].

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. Working with specifications at an abstract level of temporal granularity is not new. Previous work on dynamic switch-level circuits encountered similar situations, where it is useful to generate gate-level circuits in which the clock is abstracted out [JBJ95, KSL95]; previous work on reachability checking utilized the efficient exploration of temporal abstraction hierarchies [AHR98]. 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.

The goal of our verification effort is a proof that the implementation of the VGI chip refines its specification. As a result, a programmer who wishes to find out how the VGI chip would behave with his program can simulate his program on the specification of the VGI chip instead. Our refinement proof guarantees that the behaviors produced by the implementation of the VGI chip conforms to behaviors produced by the specification. An interesting, but different question is whether the specification correctly captures the intent of the designer. It is not possible to answer this question formally, because intentions are informal and imprecise. However, it is possible to check if the specification satisfies

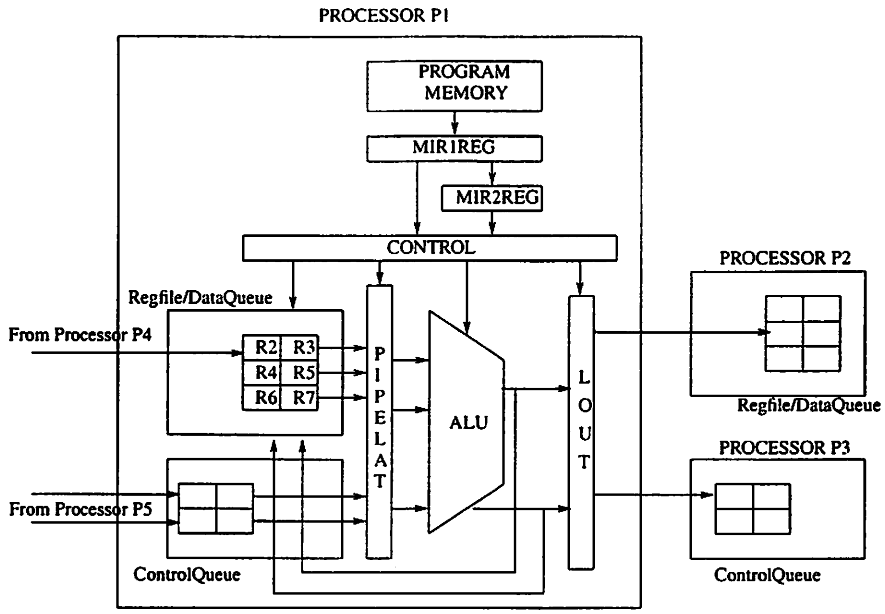

Figure 1: VGI processor configuration with three input and two output queues

specific properties expressed in temporal logics using model checking.

Outline. In Section 2, we describe the implementation and specification of the VGI chip in more detail. In Section 3, we describe our notion of refinement based on a sampling operator and introduce the corresponding verification methodology based on assume-guarantee reasoning. We show that the verification of an arbitrary network of compute processors can be reduced to the verification of a finite set of configurations of a single processor. But even a single processor is too large to be handled by a model checker directly. In Section 4, we describe the compositional verification of a single processor using assume-guarantee reasoning and MOCHA. We conclude with a discussion of the bugs found and insights gained in Section 5.

2 The Problem

2.1 Design implementation

We briefly describe the architecture of the VGI chip. The VGI chip is a DSP processor array comprising 64 compute processors, 16 memory processors and 16 I/O processors, connected by a statically programmable hierarchical communication network. The processors are arranged in 16 clusters with 4 compute processors, 1 memory processor and 1 I/O processor in each cluster. There is a single clock variable `clk` for the chip. To enhance performance, the design uses a two phase clocking scheme, level sensitive latches of either polarity and gated clocks. In this work, we focus on the verification of compute processors and the data communication among them.

A compute processor in the VGI chip has a pipelined datapath unit, a control unit and a 16-word instruction memory. There are 6 registers organized in pairs. Each pair could either be configured as two general purpose registers or as a 2-place input queue. A processor can output data and control tokens to multiple downstream processors. The interconnection between the processors is programmed statically at the beginning before any computation starts. Each compute processor can be in one of a finite number of configurations depending on how its input and output are configured. At the input, each register pair can be configured either as a queue or as general purpose registers. At the output, each output bus can be used or unused. For instance, Figure 1 shows the configuration when the register pair R2-R3 is configured as a queue, and one data output queue and the control output queue are configured to send data and control tokens respectively to other processors. We do not model the configuration logic that configures the network, and the scan logic that is used by the hardware debugger. After these simplifications, each compute processor model has 1700 variables, of which 800 are latch variables.

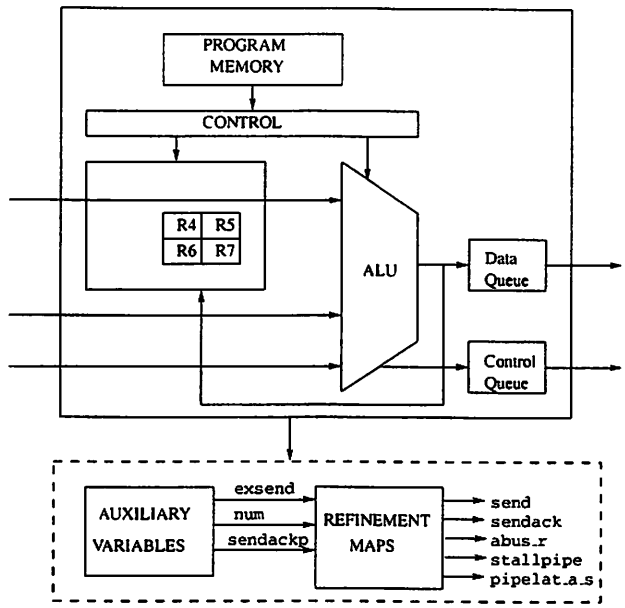

Figure 2: Specification module for refinement check

A data flow computation is performed by connecting the compute processors in a network. The instruction memory of each compute processor is programmed individually, and each processor functions as an "actor" in a data flow network, consuming data and control tokens from its input and producing data and control tokens to its output. Suppose we want to configure two processors so that one sends data to another. Then, one of the register pairs of the receiver has to be configured as an input queue and an output bus of the sender should be connected to the input bus of the receiver by programming the network. No assumption is made about the relative speeds at which data is being produced and consumed. Hence, an elaborate handshake mechanism is used to transfer data reliably between the sender and the receiver. Since an instruction can read from multiple input queues and send to multiple output queues, care must be taken to ensure that data-flow semantics is preserved. For example, if an instruction reads data from two input queues and data is available only on the first queue and the second queue is empty, then the read from the first queue has to be delayed until data is also available in the second queue. Similarly, if an instruction wants to send to two output queues and the second queue is full, then the data is sent to the first queue and the processor stalls until the second queue has space. Irrespective of the relative speeds of the processors, data should not be lost and the same data should not be sent multiple times.

2.2 Design specification

Our goal is to come up with a specification for an arbitrary (but statically) programmed network of the compute processors. We do this by writing a specification for the computation and communication parts of the compute processor. The module `ISA` is a very simple specification of the computation — data values read either from a register or a queue, results are computed and output is written to the output queues and/or written back to a register, as a result of executing an instruction. To write the specification of the communication, we observe that in the implementation, each queue is actually distributed between the sender and the receiver with handshake being performed between the two. We specify the distributed queue as a simple 4-place FIFO buffer. Performing verification against this specification will ensure that the handshake transfers data reliably from the sender to the receiver. The modules `DataQueue` and `ControlQueue` are descriptions of a 4-place FIFO queue different only in the data width.

The specification for any particular processor configuration can be obtained by composing the component specifications of its computation and communication units. For example, the specification for the configuration shown in Figure 1 can be obtained by appropriately composing a module `ISA`, one instantiation of the module `DataQueue` and one instantiation of the module `ControlQueue`. This is shown in Figure 2[1]. The specification of a network of processors can be obtained by composing the specifications of the individual processors. In Figure 2, note that the register pair R2-R3 is

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

missing. Since they have been configured as an input queue, they are part of the distributed output queue of an upstream processor, and will be specified in that processor. 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 [AH96]. 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 them 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 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 in 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 environment. These environment assumptions can then be discharged separately. A crucial aspect of this decomposition process is the use of "refinement maps". We can 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 can use the following proof rule.

$$\frac{\begin{array}{rcl} P\|E_P & \preceq & E_Q \\ Q\|E_Q & \preceq & E_P \end{array}}{\begin{array}{rcccl} P\|Q & \preceq & E_P\|E_Q & \preceq & P'\|Q' \end{array}}$$

  The use of circular environment assumptions as shown in the proof template above 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 $\{\tau_{\varphi} | \tau \text{ is a trace of } P\}$.
- $\texttt{Sample}_{\varphi}(P, T)$ is a module with the same set of external, interface variables as P, and with the trace language given by the set $\{\tau_{\varphi} | \tau \text{ is a trace of } P \| T\}$.

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 generalized the assume-guarantee proof rule described above as follows.

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

A formal treatment of the correctness of this proof rule can be found in [HQR99]. 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 assumptions $T_P \| T_Q$. We can further decompose this proof using traditional assume-guarantee reasoning [HQR98] 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.

Each compute processor in VGI starts computation in the positive phase of the clock and finishes the computation 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 the previous section 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 to it 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 here is to check

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

We can prove the above by applying 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. Each P_i can be in any one of a finite number of configurations. Moreover, if P_j and P_k are in the same configuration, then the jth and kth proofs are identical except for variable renaming. Let $C_{\texttt{VGI}}$ be the finite set of configurations for the compute processors. For $X \in C_{\texttt{VGI}}$, let Y be its specification and T_X be the environment constraint that says that all inputs change only when clk is HIGH. Then, it suffices to prove that for each $X \in C_{\texttt{VGI}}$, we have that $\texttt{Sample}_{\varphi}(X, T_X) \preceq Y$. Thus, we decompose the proof of a 64 processor network to proofs about individual processor configurations that have 800 latches each. This is still beyond the scope of monolithic model checking. In the next section, we show how we discharged 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.

4 The Proof

In this section, we describe the compositional proof for the configuration in Figure 1. The block diagram for the 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 `COM` stage is trying to send a token and the instruction in `EX` stage also wants to send out a token.

The specification of the computation is an `ISA`– in a single cycle, data values are read either from a register or a queue, results are computed and output is written to the output queues and/or written back to a register. The invariant that synchronizes the operation of the specification and implementation is the following: the instruction being executed by the `ISA` is the instruction in the `IF` stage of the implementation. The specification for data communication is a 4-place FIFO buffer.

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`. `exsend` is true whenever the the current instruction in the `EX` phase wants to send. `num` keeps track of the number of items in the receiver's two place input queue. `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 `pipelat_a_s` signals, which are an input 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 and existing techniques geared for it can easily prove it.

6. The refinement map for `pipelat_a_s` refinement 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 send signal does not change value when clk changes from LOW to HIGH, and all signals at the receiver end such as read, 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 need to be proved. In every lemma except Lemma 5, we used symmetry arguments [McM98] 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 very clean separation between the verification of the datapath and control of the processor. It is very 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. Our fixes are currently being reviewed by the designers.

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 in it.

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 till we got no error trace. 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. At this point, we were using MOCHA as a debugging tool that would test our proposed fix by throwing at it all possible sequences of events. If it generated an error trace then we could look at it and 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 his proposed solution by throwing at it 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 the verification tool, it is not a big burden to do "formal design".

References

[AH96] R. Alur and T.A. Henzinger. Reactive modules. In *Proceedings of the 11th Annual Symposium on Logic in Computer Science*, pages 207–218. IEEE Computer Society Press, 1996.

[AHM+98] R. Alur, T.A. Henzinger, F.Y.C. Mang, S. Qadeer, S.K. Rajamani, and S. Tasiran. MOCHA : Modularity in model checking. In A. Hu and M. Vardi, editors, *CAV 98: Computer Aided Verification*, Lecture Notes in Computer Science, pages 521–525. Springer-Verlag, 1998.

[AHR98] R. Alur, T.A. Henzinger, and S.K. Rajamani. Symbolic exploration of transition hierarchies. In B. Steffen, editor, *TACAS 98: Tools and Algorithms for Construction and Analysis of Systems*, Lecture Notes in Computer Science 1384, pages 330–344. Springer-Verlag, 1998.

[AL88] M. Abadi and L. Lamport. The existence of refinement mappings. In *Proceedings of the 3rd Annual Symposium on Logic in Computer Science*, pages 165–175. IEEE Computer Society Press, 1988.

[AL95] M. Abadi and L. Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–534, 1995.

[BHSV+96] R.K. Brayton, G.D. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S.-T. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R.K. Ranjan, S. Sarwary, T.R. Shiple, G. Swamy, and T. Villa. VIS: A System for Verification and Synthesis. In R. Alur and T.A. Henzinger, editors, *CAV 96: Computer Aided Verification*, Lecture Notes in Computer Science 1102, pages 428–432. Springer-Verlag, 1996.

[CLM89] E.M. Clarke, D.E. Long, and K.L. McMillan. Compositional model checking. In *Proceedings of the 4th Annual Symposium on Logic in Computer Science*, pages 353–362. IEEE Computer Society Press, 1989.

[Eir98] A.T. Eiríksson. The formal design of 1M-gate ASICs. In G. Gopalakrishnan and P. Windley, editors, *FMCAD 98: Formal Methods in Computer-Aided Design*, Lecture Notes in Computer Science 1522, pages 49–63. Springer-Verlag, 1998.

[GL94] O. Grumberg and D.E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, 1994.

[HQR98] T.A. Henzinger, S. Qadeer, and S.K. Rajamani. You assume, we guarantee: methodology and case studies. In A. Hu and M. Vardi, editors, *CAV 98: Computer Aided Verification*, Lecture Notes in Computer Science, pages 440–451. Springer-Verlag, 1998.

[HQR99] T.A. Henzinger, S. Qadeer, and S.K. Rajamani. Assume guarantee refinement between different time scales. In *CAV 99: Computer Aided Verification (to appear)*. 1999.

[JBJ95] S. Jain, R.E. Bryant, and A. Jain. Automatic clock abstraction from sequential circuits. In *Proceedings of the 32nd Design Automation Conference*, pages 707–711, 1995.

[KSL95] A. Kuehlmann, A. Srinivasan, and D.P. LaPotin. Verity - a formal verification program for custom CMOS circuits. *IBM Journal on Research and Development*, 39(1-2):149–165, 1995.

[Kur94] R.P. Kurshan. *Computer-aided Verification of Coordinating Processes*. Princeton University Press, 1994.

[Lyn96] N.A. Lynch. *Distributed Algorithms*. Morgan-Kaufmann, 1996.

[McM97] K.L. McMillan. A compositional rule for hardware design refinement. In O. Grumberg, editor, *CAV 97: Computer-Aided Verification*, Lecture Notes in Computer Science 1254, pages 24–35. Springer-Verlag, 1997.

[McM98] K.L. McMillan. Verification of an implementation of Tomasulo's algorithm by compositional model checking. In A. Hu and M. Vardi, editors, *CAV 98: Computer-Aided Verification*, Lecture Notes in Computer Science. Springer-Verlag, 1998.

[SSTR97] V.P. Srini, Spartan-Team, and J.M. Rabaey. An architecture for web-based image processing. In *Proceedings of the SPIE Conference 3166*, 1997.

[Sta85] E.W. 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, pages 369–391. Springer-Verlag, 1985.

[STUR98] V.P. Srini, J. Thendean, S.Z. Ueng, and J.M. Rabaey. A parallel DSP with memory and I/O processors. In *Proceedings of the SPIE Conference 3452*, 1998.