Formal Verification of Pipeline Control Using Controlled Token Nets and Abstract Interpretation

Pei-Hsin Ho
Advanced Technology Group
Synopsys, Inc.
pho@synopsys.com

Adrian J. Isles
Department of EECS
University of California, Berkeley
aji@eecs.berkeley.edu

Timothy Kam
Strategic CAD Labs
Intel Corporation
tkam@ichips.intel.com

1. ABSTRACT

We present an automated formal verification method that can detect common pipeline-control bugs of logic-design components containing thousands of registers. The method models logic designs using *controlled token nets*. A controlled token net consists of: a *token net* that models the data flow in the datapath using token semantics; a *control logic* that models the control machines using traditional finite state semantics. We provide algorithms to (1) extract a controlled token net from a logic design, (2) minimize the controlled token net, and (3) compute an abstract interpretation of the controlled token net for efficient model checking. We implemented and applied the method to 6 Intel logic-design components containing up to 4500 registers and successfully detected 8 pre-silicon errata.

1.1 Keywords

Pipeline control verification, controlled token net, abstract interpretation, processor verification, model checking, formal verification, functional verification, computer-aided design

2. INTRODUCTION

As microprocessors use more and more complex pipelining, bugs are more likely to arise in pipeline control. Moreover, a subtle pipeline-control bug may only manifest itself with one particular input sequence. Given the low probability of exercising such an input sequence among billions of possible input sequences, pipeline-control bugs are usually hard to catch using simulation. In contrast to the stimulus-based logic simulation, formal verification (FV) tools do not require test vectors, since they analyze all possible input sequences. Thus formal verification is a good candidate for validating pipeline control.

Model checking [13] is the most widely used formal verification technique for verifying properties of VLSI designs. However, there are two basic issues with model checking pipeline control today: capacity and specification. For the capacity issue, existing model checkers cannot consistently verify designs with more than a few hundred registers, while real-life design components contain more than a thousand registers. To verify such a design component, we have to manually simplify the design component into verifiable models. From our experience, this manual abstraction process can be both error-prone and time-consuming, not to mention the level of FV expertise required by the process.

Specification is also an issue as existing temporal logics are interpreted by the evaluation of all state variables. As a result it is hard to separate the pipeline-control verification from the datapath verification. For example, consider the following simple pipeline-control property: given a "valid" input data *a*, the corresponding output data will eventually be produced. We may want to verify the above property to see if a valid data can be mistakenly squashed in the pipeline after some rare combinations of stalls. The pipeline control property may be specified by the following LTL formula:

$$G((input = a \wedge validin) \rightarrow F(output = f(a) \wedge validout))$$

where G and F represent the temporal operators "always" and "eventually" respectively, *input* is the input vector (for example, the combined input vector of the two operands to a pipelined ALU), *validin* is a control signal that indicates if the input data is valid, *output* is the output vector, *validout* is a control signal that indicates if the output data is valid, and f represents the function that the datapath performs. If the function f is very complex, specifying f in a temporal-logic formula would be difficult since the formula can become as complex as the design itself. This example illustrates why we want to specify pipeline-control properties without mentioning the exact function f of the datapath. An exception may be arithmetic circuits where the function of the datapath can be reasonably specified in word-level temporal logic [3]. But even so, verification of the pipeline control and the datapath functionality at the same time may be unnecessarily expensive.

2.1 Previous Work

There are many approaches in the literature that utilize data-abstraction techniques to verify pipelined designs. Theorem-proving techniques, for example, have been successfully adapted to verify pipelined processors [6][16][17]. The disadvantage of these approaches, however, is that they require a great deal of user intervention, especially for verifying control-intensive designs.

Uninterpreted functions have been shown to be useful in abstracting away datapath complexity for pipeline-control verification. In particular, Burch and Dill [1][2] introduced an automated method for verifying a pipelined processor (modeled using uninterpreted functions) against its architectural specification. Their method guarantees a clean correctness criterion and can verify larger designs than could previous approaches. However, to apply this method to design components at the micro-architecture level, one may need to overcome two difficulties. First, since the design components tend to perform low-level functions and heavily interact with other components, a clean formal specification of the design required by the method is very difficult to obtain in practice. Second, the verification capacity of the method is limited by the capacity to construct an inductive invariant of the abstract model, the validity checker and the symbolic simulator. It is very difficult to construct a verifiable abstracted model from a design component with thousands of registers using uninterpreted functions.

Levitt and Olukotun introduced the un-pipelining method [12] that also utilizes abstract models with uninterpreted functions. The advantage of the method is that it does not require a formal specification of the model. But the design components that we encountered often did not fit the pipeline topologies that the un-pipelining method applies.

The approaches of [4] and [10] perform symbolic state exploration of abstracted models with uninterpreted functions. The usage of uninterpreted functions in the abstract model however makes the reachability analysis undecidable [9].

The self-consistency method introduced in [11] also does not require formal specification of the design. But since the design components often exceed the capacity limit of symbolic simulators, the method in practice often becomes a simulation method rather than a formal-verification method.

In [8], Hojati and Brayton presented an automatic data-abstraction technique to reduce the datapath width to one or very few bits. However, their method only works for datapaths where computations are not performed on data.

The method in this paper does not require the user to provide an abstracted model of the design component with uninterpreted functions nor a complete formal specification of the design component. The method computes an abstract model of the design. Common pipeline-control properties including liveness properties of the design component can be efficiently verified against the abstract model using an off-the-shelf model checker.

2.2 Controlled Token Nets

We introduce *controlled token net* to model pipelined logic designs. A controlled token net consists of two parts: a *token net* that models the data flow in the datapath using a Petri-net-like [14] token semantics, and a *control logic* that models the control machines using a traditional finite state semantics. The data in the datapath are modeled as tokens in the token net. A token is a symbol of an infinite alphabet. The combinational functional blocks in the datapath are modeled as *token union gates* that collect the input tokens. As a result, the exact function of the combinational logic block is abstracted away. The tokens are staged in token latches and steered by token muxes in the token net. The control logic controls the token net through the mux selection signals generated by the control machines.

Controlled token nets enable us to specify pipeline-control properties without having to specify the functionalities of the datapath. We introduced property-specification language Token Linear Time Logic (TLTL) by extending Linear Time Logic (LTL) with token propositions to refer to the set of tokens that are present at a particular signal. For example, the property in the previous example can now be specified as:

$$G((a \in input \wedge validin) \rightarrow F(a \in output \wedge validout))$$

where $a \in input$ and $a \in output$ are token propositions specifying that the token a is a member of the set of the tokens of the token variables *input* and *output*, respectively. The variables *validin* and *validout* are control variables in the control logic. Note that the function f computed by the datapath is no longer mentioned in the property.

We provide algorithms to (1) extract a controlled token net from the logic design, (2) minimize the controlled token net to reduce verification complexity, and (3) compute a finite abstract interpretation [5] of the controlled token net for efficient model checking with

off-the-shelf model checkers. For a particular controlled token net, the minimization and abstract interpretation algorithms are both *sound* (no false positive verification results) and *complete* (no false negative verification results). However, the user needs to provide hints to the extraction algorithm to obtain a meaningful controlled token net model of the logic design.

We applied this method to 6 Intel design components and detected 8 pre-silicon errata in their pipeline control. We found that our symbolic model checker was able to verify the abstract models of design components generated by the minimization and abstraction techniques. The method allows the model checking of general TLTL properties, which were sufficient to specify most pipeline-control properties that we had in mind during the experiments. However, the specification issue is not completely resolved. Without a formal specification of the design, although we can verify many interesting TLTL properties of the pipeline control, we do not know if the set of verified TLTL properties actually "covers" all the functionality of the pipeline control.

2.3 Outline

The rest of the paper is organized as follows. We discuss in the next section the common pipeline-control bugs found in practice. In Section 4 we introduce the controlled token nets for modeling the design and TLTL formulas for modeling the pipeline-control properties. We present the minimization algorithm and the abstract interpretation algorithm in Section 5. The extraction algorithm is presented in Section 6. We will show the results of the experiments of this method on real-world logic designs in Section 7.

3. Common Pipeline-Control Bugs

The motivation of developing the method in this paper is to catch common pipeline-control bugs in design components containing thousands of registers. In our view, a pipeline consists of two parts, the datapath and the control. The datapath contains complex combinational logic to compute some function and wide registers and muxes to store and steer the intermediate data. The control controls the transfer of the data in the datapath by controlling the enable, clock and reset signals of the registers and the selection signals of the muxes in the datapath. We describe common bugs in the control of the pipeline in the following section.

Pipeline control is complex due to pipeline hazards [7] that include structural hazards (stalling due to resource conflicts), control hazards (squashing and flushing due to control dependencies) and data hazards (bypassing due to data dependencies). Although the pipeline control may be very different from design to design, we can informally characterize the pipeline-control bugs by their impact as follows.

1. *Data that is not supposed to be lost in the pipeline is lost in the datapath.* If we analyze the bug at the gate level, a valid data can be mistakenly lost if any one of the following situations (and many others not discussed here) happen.

   First, let *u* and *v* be two edge-triggered flip-flops (each controlled by a clock signal and an enable signal) such that *u* fanouts to the input signal of *v.* If at the same triggering clock edge of both flip-flops, the enable signal of *u* is (mistakenly) asserted while the enable signal of *v* is (mistakenly) de-asserted, then the data that was residing at the flip-flop *u* will not be latched into *v* and will be over-written by some new data.

   Second, let *u* and *v* be transparent latches (each controlled only by an enable signal) and the enable signals of *u* and *v* are mistakenly asserted at the same time (race condition).

   Third, a selection signal of a mux is generated incorrectly such that the valid data is not chosen to become the mux output.

   Fourth, the reset signal of a register (flip-flop or latch) is asserted by mistake.

   If we analyze these bugs at the micro-architecture level, a valid data may be mistakenly destroyed if the pipeline control (1) does bubble squashing during the stall of the pipeline (structural hazard); (2) destroy data in a portion of the pipeline during a mispredicted branch (control hazard); or (3) picks only one of several results of speculative computation and loses the rest (data hazard).
2. *Data that is supposed to be destroyed survived in the datapath*. The bug is the dual of the first one and may occur because of the same set of pipeline hazards. For example, let *u* and *v* be the two edge-triggered flip-flops in the very first example. If at the triggering clock edge of both flip-flops, the enable signal of *u* is de-asserted while the enable signal of *v* is asserted, then the data that was residing at the flip-flop *u* will also be latched into *v* and thus there will be two copies of the same data.
3. *FIFO (first in first out) property is violated.* A bug

in the control of an array that implements an FIFO queue can cause the problem.

4. *Wrong source of data is used for some computation*. A bug in the bypassing logic or the selection logic of some speculative computation can be the cause.

Note from the above bug descriptions, it is very difficult to characterize the precise conditions of the clock, enable, reset and mux selection signals that will result in such pipeline-control bugs. So rather than specifying pipeline-control properties with respect to these low-level control signals, we should specify how the data should be transferred in the pipeline. With the controlled token nets introduced in this paper, we can specify how the data should be transferred in the pipeline without mentioning how the data should be computed in the pipeline. Traditional state semantics does not allow us to separate the above two tasks.

4. Controlled Token Nets and TLTL

We model data in the datapath as *tokens*. A token is a symbol of an infinite alphabet *A*. At each execution step, the data at each primary input of the design is modeled as a set containing a single unique token. The set of output tokens of arbitrary logic function is defined to be the *union* of the sets of input tokens. The output token set records the tokens that participate in the computation but not the logic function performed during the computation. We will see in later sections that the information on the participants is sufficient for verifying many pipeline-control properties and the abstraction of the logic function in the datapath enables the minimization of the model of the datapath.

4.1 Controlled Token Nets

Based on the notion of tokens, we use *controlled token nets* to model pipelined designs. A controlled token net consists of two parts: a *token net* modeling the datapath of the design and a *control logic* modeling the control of the design.

Formally, a controlled token net $M = (N,L)$ consists of a *token net N*, and a *control logic L*. Both the token net *N* and the control logic *L* are sets of tuples of the form (ν, T_ν, I_ν), where ν is a *variable*, T_ν is the *transition relation* and I_ν is the *initial condition* of the variable ν. Let the set $V = \{\nu_1, \nu_2, \ldots, \nu_n\}$ be the set of variables of the controlled token net *M*. To specify the characteristic function of a transition relation, we use the set *V* of unprimed variables and the set $V' = \{\nu'_1, \nu'_2, \ldots, \nu'_n\}$ of primed variables to denote the current and next values of the variables.

A variable ν of a controlled token net *M* is either (1) a *primary input* if T_ν (represented by its characteristic function) is *True*, (2) a *latch* if T_ν is of the form $\nu' = \mu$ where the variable μ is called the *input* of the latch ν, (3) a *mux* if T_ν is of the form $\nu' = ((x' \wedge \mu'_1) \vee (\bar{x'} \wedge \mu'_2))$ where the variable *x* is called the *selection variable* of the mux ν and the variables μ_1 and μ_2 are called the *inputs* of the mux ν, or (4) a *gate* if T_ν is of the form $\nu' = f(\mu'_1, \mu'_2, \ldots, \mu'_k)$ where the variables μ_i are called the *inputs* of the gate ν and *f* is a function over the inputs. Note that more complex registers such as edge-triggered flip flops or registers with clock, enable, reset and set signals can be modeled by latches, muxes and primary inputs.

We say that the variable μ *fanouts* to the variable ν or the variable μ is in the *fanin* of the variable ν if the variable μ is an input of the variable ν. A variable is a *primary output* if it does not fanout to any other variables. Informally, combinational loops are not allowed in controlled token nets. More precisely, for any variables $\mu_1, \mu_2, \ldots, \mu_k$ such that μ_1 fanouts to μ_2,..., μ_{k-1} fanouts to μ_k, and μ_k fanouts to μ_1, there must exist a variable μ_i that is a latch.

The variables *v* of the token net *N* are called *token variables*. The token variables can be assigned to a set of tokens, subset of the infinite alphabet *A*. The inputs of token variables are also token variables. The selection variables of token variables are variables from the control logic. The transition relation T_v for each gate *v* of the token net is of the form $v' = u'_1 \cup u'_2 \cup \ldots \cup u'_k$ where the token variables u_i are the inputs of the gate *v*. A gate in the token net is called a *token union gate*. The initial condition of the token net requires each latch be assigned the empty token set $\varnothing$.

Variables of the control logic *L* are called *control variables*. Control variables are Boolean variables. The inputs of control variables are also control variables.

4.2 Semantics

The transition relation T_M of the controlled token net *M* is the conjunction of all the transition relations of the variables of *M*. Similarly, the initial condition I_M of the controlled token net *M* is the conjunction of all the initial conditions of the variables of *M*.

A *state* δ of the controlled token net *M* is an evaluation of all variables of *M*. Let $\delta(\nu)$ denote the value of variable ν in state δ. We define the *successor* relation $\rightarrow$ of *M* such that $\delta_0 \rightarrow \delta_1$ if $T_M(\delta_0, \delta_1)$ is *True*, where $T_M(\delta_0, \delta_1)$ is the result of substituting each variable ν of *V* by $\delta_0(\nu)$ and each primed variable ν' of *V'* by $\delta_1(\nu)$ in the transition relation T_M. A *trace* $\tau = (\delta_0, \delta_1, \ldots)$ of *M* is an infinite sequence of states such that (1) the initial state δ_0 satisfies the initial

condition ($I_M(\delta_0)$ is *True*), (2) state δ_{i+1} is a successor of the state δ_i ($\delta_i \rightarrow \delta_{i+1}$) for all *i*, and (3) token primary inputs always get fresh tokens (for any token primary inputs ν and μ, $(\delta_i(\nu) = \delta_j(\mu))$ implies that both $i = j$ and $\nu = \mu$). The *semantics* of *M* is the set of all traces of *M*.

4.3 Token Linear Time Logic

To specify properties of the controlled token nets, we extend *Linear Time Logic* (LTL) to *Token LTL (TLTL)* that includes token propositions. A TLTL proposition p is:

$$p \equiv x \mid (a \in v)$$

where *x* is a variable of the control logic, *v* is a variable of the token net, and *a* is a token. The syntax of TLTL formulas is defined recursively as follows.

$$\varphi \equiv p \mid (\varphi_1 \wedge \varphi_2) \mid \neg\varphi \mid X\varphi \mid \varphi_1 U \varphi_2$$

where p is a TLTL proposition, X and U are respectively the next and strong until temporal operators. We also define $F(\varphi), G(\varphi), \varphi \vee \psi, \varphi \rightarrow \psi$ and $\varphi = \psi$ to be the abbreviations for the formulas $TrueU\varphi, \neg F\neg\varphi, \neg(\neg\varphi \wedge \neg\psi), \neg\varphi \vee \psi$ and $\varphi \rightarrow \psi \wedge \psi \rightarrow \varphi$, respectively. TLTL formulas are interpreted in the usual way on the semantics of controlled token nets.

4.4 Verification of Pipeline Control Using TLTL

Now we demonstrate how each type of pipeline-control bugs mentioned in Section 3 can be verified by model checking TLTL properties.

First, consider bugs of Type 1. Suppose that for a token primary input *u* and a token primary output *v* of the datapath, we want to verify if each valid input data at variable *u* will eventually results in a valid output data at variable *v*, unless some exception occurs. In other words, token will not be lost in the transfer from token variable *u* to token variable *v* unless an exception occurs. This property can be specified as the following TLTL formula:

$$G((a \in u \wedge validin) \rightarrow F((a \in v \wedge validout) \vee exception))$$

where *validin* (*validout*) is a TLTL formula that specifies when the data residing at token variable *u* (token variable *v*) is considered as valid input (output) data, *exception* is a TLTL formula specifying when the exception occurs. Informally the formula specifies that it is always the case that a "valid input" token of *u* will eventually result in a "valid output" token of *v* unless the *exception* happens. This property also applies to designs with multiple token primary inputs and outputs. In those cases, we may want to verify this property for each pair of token primary input *u* and token primary output *v* such that the token at the variable *u* should eventually reach the variable *v*. The following is a possible formula for the TLTL formula *exception*:

$$(a \in w) \wedge reset$$

which says that the *exception* occurs when the *reset* variable is asserted and the data (token) is residing at a token variable *w* of the token net. Note that if the token *a* is not at the token variable *w* when the reset occurs, the token *a* is still supposed to eventually show up at the token primary output variable *v*.

Second, consider bugs of Type 2. Suppose we want to make sure that if the exception condition occurs then the data originated from token variable *u* will always be destroyed. Then we verify the following formula:

$$G(((a \in u) \wedge F(exception)) \rightarrow G\neg((a \in v) \wedge validout))$$

that basically specifies if the *exception* ever happens (*reset* is asserted when the unique token resides at variable *w*) then there will never be a valid output token appearing at *v*.

Third, consider bugs of Type 3. Suppose that we want to verify the FIFO property of a queue (maybe implemented as an array with read and write pointers) with input *u* and output *v*:

$$G((a \in u \wedge F(b \in u)) \rightarrow$$
$$(\neg(b \in v)U(a \in v)) \wedge F(b \in v))$$

which says that it is always the case that if we send token *a* and later the token *b* to the token variable *u*, then we will not see the token *b* appearing at the token variable *v* until we have seen the token *a* appearing at the token variable *v*, and we will eventually also see that the token *b* appears at the token variable *v*.

Fourth, consider bugs of Type 4. The formula

$$G(((t[0\ldots31] = w[0\ldots31]) \wedge (a \in u)) \rightarrow X(a \in v))$$

is an example of a simple by-passing property that says if the address fields $t[0\ldots31]$ and $w[0\ldots31]$ are the same, then in the next cycle the token variable *v* gets the token from token variable *u*.

Note that if we consider $t[0\ldots31]$ and $w[0\ldots31]$ as datapath variables then they will be modeled as token variables in the token net. However, TLTL formulas do not allow for comparisons to be made between token variables. But since the pipeline control only "monitors" the datapath through *feedback signals*, signals from the datapath to the control, we can usually express the above property as a TLTL formula by replacing datapath variables in the formula with

feedback signals. For example, we can replace decoder or comparator inputs (such as $t[0\ldots31]$ and $w[0\ldots31]$) by decoder or comparator outputs (some feedback signals). In the above example, the design is likely to contain a comparator in the datapath that computes $t[0\ldots31] = w[0\ldots31]$ and asserts/de-asserts a feedback signal *match* according to the result. The feedback signal *match* should be a control variable. Thus we can specify the property as:

$$G((match \wedge (a \in u)) \rightarrow X(a \in v))$$

which is a TLTL formula.

5. Minimization and Abstract Interpretation

Since all the gates in the token net are token union gates, many token latches are "equivalent" in the sense that they are always assigned the same set of tokens. We present an iterative algorithm that can identify these equivalent latches and compute a minimized token net that is easier to model check. We can obtain sound and complete results of verifying TLTL properties of a controlled token net by verifying the minimized TLTL property of the minimized controlled token net. Our minimization algorithm has a flavor of the bisimulation partitioning algorithm --- given a token net N and an initial partition of the variables, the algorithm iteratively refines the partition until the partition is "stable".

5.1 Minimization of Token Nets

We say that variables u and v are *control-compatible* if each variable does not fanout to the other variable and any one of the following conditions holds:

- token variables u and v are both latches, or
- token variables u and v are both muxes that have the same selection variable and disjoint sets of inputs, or
- token variables u and v are both token union gates.

The initial partition of the variables is the partition of the token net into minimum number of classes such that the variables in each class are control-compatible. The variables u and v are *control-equivalent* with respect to the current partition if

- the variables are control-compatible, and
- if $\bar{u}$ fanouts to u for some variable $\bar{u}$ in a class c, then there is a variable $\bar{v}$ in the class c such that $\bar{v}$ fanouts to v, and
- if $\bar{v}$ fanouts to v for some variable $\bar{v}$ in a class c, then there is a variable $\bar{u}$ in the class c such that $\bar{u}$ fanouts to u.

A class is *stable* if and only if all the variables in that class are *control-equivalent.* The algorithm continues dividing the unstable classes into stable classes until every existing class is stable. Then the set C of the stable classes is called the *quotient* of the token net N.

In the second step, we compute a minimized token net from the quotient C as follows. We define a representative for each class c of the quotient C. Let π be the function that maps each control variable x to x itself and maps each token variable v to the representative of the class c that contains the variable v. Define $\pi(T_v)$ to be the result of substituting each variable μ in T_v by $\pi(\mu)$. Then the minimized token net consists of the set of representatives:

$$\pi(N) = \{(c, \pi(T_c), c = \varnothing) \mid c \in C\}$$

The minimized controlled token net $\pi(M)$ is defined as $(\pi(N),L)$. For a TLTL property φ, we define the minimized property $\pi(\varphi)$ to be the TLTL property obtained from the property φ by replacing each variable ν with the variable $\pi(\nu)$. Theorem 1 says that verifying the minimized controlled token net against the minimized TLTL property is the same as verifying the original controlled token net against the original TLTL property.

Theorem 1 *Controlled token net M satisfies TLTL property φ if and only if the minimized controlled token net $\pi(M)$ satisfies the TLTL property $\pi(\varphi)$.*

5.2 Minimization of Token Primary Inputs

For a subset of TLTL formulas, the minimization algorithm can achieve a greater reduction in the size of the token net by first merging some of the primary inputs of the token net into one primary input. In fact, all the TLTL properties discussed in Section 4.4 belong to this subset.

We call this subset of TLTL properties the *transmission properties*. Transmission properties are of the form $\varphi = G(\varphi_1 \rightarrow \varphi_2)$, where the antecedent φ_1 is a conjunction of formulas of the form p, $X(p)$, $F(p)$, or $G(p)$, where p is a proposition. In addition, we require that the tokens appearing in the formula φ_2 is a subset of the tokens appearing in φ_1. Note that negation and disjunction are not allowed in the formula φ_1.

Let M be a controlled token net and let the token variables $u_1, u_2, \ldots, u_k$ be the set of token primary inputs that are not in the transitive fanin of the token variables that appear in the formula φ_1. The *input-minimized controlled token net M'* is obtained from M by merging the token primary inputs $u_1, u_2, \ldots, u_k$ into one token primary input. Note that the merged token primary inputs can be in the transitive fanin of token

variables that appear in the formula φ_2. We have the following result.

Theorem 2 *Controlled token net M satisfies transmission property* φ *if and only if the input-minimized controlled token net M' satisfies the same transmission property* φ.

Therefore, if we are verifying a transmission property, we can first compute the input-minimized controlled token net *M'* and then apply the minimization algorithm in Section 5.1 to compute the minimized controlled token net from *M'*.

5.3 Abstract Interpretation of Controlled Token Nets

In order to verify a controlled token net M against a TLTL formula φ using an off-the-shelf model checker, we construct an abstract controlled token net $\hat{M}$ of the controlled token net M and an abstract property $\hat{\varphi}$ of the property φ. The abstract controlled token net M is a control logic of Boolean variables and can be verified against the abstract property $\hat{\varphi}$ by an ordinary model checker. The verification result is sound and complete with respect to the original controlled token net M and the original property φ.

Given a TLTL property φ, we define the finite *abstract alphabet* $A_\varphi \subseteq A$ to be the union of all tokens that appear in the property φ. The abstract interpretation of a set of token is the intersection of the set of tokens with the abstract alphabet A_φ. Suppose that the size of the abstract alphabet A_φ is *n*. The abstract interpretation of each set of tokens can be encoded by a bit vector of *n* bits.

We now define the abstract token net $\hat{N}$. In the abstract controlled token net $\hat{N}$, each token union gate of the token net *N* is modeled as a bitwise-OR gate of *n* bits. Each token latch and each token mux of the token net *N* are modeled as *n* binary latches and muxes respectively. Each token primary input of the token net *N* is modeled as an *n*-bit input vector. The abstract token net $\hat{N}$ also includes a state machine that guarantees that all tokens in the abstract alphabet can appear at most once at any input vectors at all time.

The abstract controlled token net $\hat{M}$ is the union of the abstract token net $\hat{N}$ and the original control logic *L*. The abstract property $\hat{\varphi}$ is obtained by translating each token proposition $a \in v$ into the binary variable that represents the token *a*.

Note that for TLTL properties containing only one token constant, The abstract token net can be obtained by replacing token union gates by binary OR gates. Since most TLTL properties contain only 1 or 2 tokens, the number of token latches of the abstract token net is the same or twice as much as the number of token latches in the original token net. In addition, an abstract token net for an abstract alphabet of size *k* work for all TLTL properties with *k* tokens.

Theorem 3 says that verifying the abstract controlled token net with respect to the abstract property is the same as verifying the original controlled token net with respect to the original property.

Theorem 3 *Controlled token net* M *satisfies TLTL property* φ *if and only if the abstract token net* $\hat{M}$ *satisfies the abstract TLTL property* $\hat{\varphi}$.

6. Extraction of Controlled Token Nets from Logic Designs

We can extract a token net from a subset of a control logic by converting each control input into a token input, each control latch into a token latch, each control mux to a token mux, and each control gate into a token union gate. In addition, we also merge a token union gate *v* with all the token union gates in the fanin of the token union gate *v* to become a single token union gate to facilitate the minimization process.

A *logic design* is a controlled token net $(\varnothing, L)$ with control logic only. To verify the pipeline control of the logic design, we want to extract a meaningful controlled token net from the logic design so that the datapath of logic design is modeled by the token net. If there is a natural partition of the logic design into datapath and control, we can easily extract a controlled token net by translating the datapath into a token net and make the control to be the control logic.

If the logic design does not have separated datapath and control, we can easily obtain a controlled token net model of the logic design that consists of the token net that is converted from the entire logic design and the control logic that is the entire logic design. The disadvantage of the this model is that the extracted controlled token net is twice as complex as the logic design.

6.1 Datapath and Control Separation

To compute a meaningful controlled token net of the right size, we provide an automated separation algorithm to partition the logic design into datapath and control. Then a controlled token net can be computed based on this partition. The separation algorithm consists of three steps. The first step identifies the *seed control signals.* The second step separates the primary inputs and latches, and the third step separates the remaining gates.

In the first step, all the mux selection variables are

considered as seed control variables. The user may also want to specify additional seed control variables in order to fine-tune the partitioning. For example, if there is a pipeline of valid bits, the entire valid-bit pipeline should be specified as seed control variables.

Second, we mark the latches and the primary inputs as data or control as follows. We first compute the *data transitive fan-in* of the *datapath primary outputs* specified by the user. The data transitive fan-in is computed by not tracing back through seed control variables while we compute the transitive fan-in of the primary datapath outputs. All the variables in this data transitive fan-in are marked as data variables. The input variables and latches not in the transitive fan-in are marked as control inputs and control latches. Note that the gates not in the data transitive fan-in have not been marked yet at this step.

In the last step we mark the remaining gates. An unmarked gate ν is marked as control if any variable in the fanin of the gate ν has been marked as control. Otherwise the gate ν is marked as data. The marking of the gates determines the set of feedback signals from the datapath to the control. The feedback signals are then cut to become primary inputs to the control. This marking rule will usually identify the decoder output variables or the comparator output variables as feedback signals automatically. Note that all selection variables of the muxes will be in the control.

7. Experiments

The method was implemented as a tool in a functional language that is very similar to standard ML. An overview of the implemented tool is as follows.

7.1 Overview of Implementation

If the given logic design does not have a natural partition of the datapath and control, the tool applies the separation algorithm in Section 6 to partition the logic design into datapath and control, as illustrated in Figure 1.

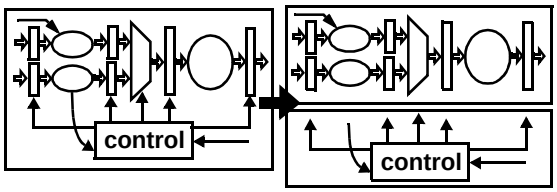

Figure 1. Datapath and control separation

Second, the tool translates the datapath into a token net, minimizes it, and computes from the minimized token net an abstract token net for an abstract alphabet of size one, as illustrated in Figure 2. Third, the tool produces the abstract controlled token net by reconnecting the control logic and the abstract token net together. In the last step, the tool invokes a LTL model checker to verify the abstract controlled token net.

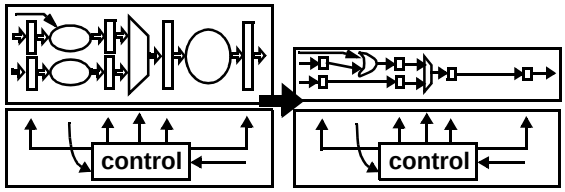

Figure 2. Minimization and Abstraction

We applied the tool on 6 Intel design components and detected 8 pre-silicon design errata. The performance figures are shown in Table 1.

7.2 Performance Figures

Table 1: Experiment Figures

Design Alias	No. Regis-ters in Design	Reduction Ratio	Abstraction Time	Verification Time
C1	~2500	~70	5 mins	10 secs
C2	~2000	~70	6 mins	2 secs
C3	~4500	~40	12 mins	30 mins
C4	~1000	~20	6 mins	1 sec
C5	~1000	~35	2 mins	2 secs
C6	~1500	~15	5 mins	5 mins

The experiments were performed on an HP workstation with 256MB memory. The first column shows aliases of Intel design components. The design components were chosen to cover different portions of microprocessor designs. The second column shows, to the nearest multiple of 500, the rounded number of registers in the original model. The third column shows, to the nearest multiple of 5, the rounded reduction ratio of the data abstraction techniques. The average model reduction rate is about 40x. The fourth column shows the time taken for computing the abstract controlled token net. We verified four pipeline control properties with one token constant for each design component. The properties are sufficient to detect 8 logic errata in these design components. The fifth column shows the time that the model checker took to verify these properties on the abstract controlled token net. The verification time does not include the time for generating tableaus from the linear-time temporal logic formulas.

7.3 False Negatives

We did observe false-negative verification results during the experiments. Our tool can associate error traces found on the minimized extracted token model to an error trace on the token model. These false negatives were then avoided by strengthening the antecedent of the properties. Overall, the false negatives do not seem to be a major issue on the examples that we ran.

8. Summary

We have presented a domain-specific abstraction technique for model checking pipeline control. We implemented the method as an automatic tool and successfully tested the tool by verifying real-world logic designs that contain up to 4500 registers.

To address the specification and capacity issue of model checking pipeline control, we introduced the controlled token nets to model the logic designs. For the specification issue, controlled token nets enable the verification of pipeline-control properties without mentioning the functionality of the datapath.

For the capacity issue, we present algorithms that can compute a minimized controlled token net from a logic design. The techniques can reduce the number of registers in the model by 40x on average. This significant reduction in model size implies big verification capacity improvement. In addition, the techniques operate on the netlist of the logic design and thus the techniques scale better than the abstraction techniques that operate on the state space of the design. Common bugs found in pipeline control can be caught by verifying the abstract model. We detected pre-silicon errata in real-world designs in our experiments.

Comparing with previous pipeline-control verification techniques, we concluded that the key advantages to this method are that it provides significantly higher capacity, automation and usability. Although, this method does not provide a guarantee of “total” correctness of the pipeline control, it is an effective and capable formal debugging tool for pipeline control.

9. Acknowledgments

We thank Carl Seger for helping us improve the performance of the algorithm for separating datapath and control by 5x. We thank Freddy Mang for implementing some missing pieces of the system and for assisting some of the experiments. We thank Lauryn Bryght, Rob Gerth, Limor Fix, John O’Leary and ICCAD reviewers for providing comments to the drafts of this paper.

10. References

[1] J.R. Burch, Techniques for verifying superscalar microprocessors, In Proceedings of The 33th ACM/IEEE DAC, 1996.

[2] J.R. Burch, and D.L. Dill. Automatic verification of pipelined processor control. In Proceedings of CAV94, Lecture Notes in Computer Science 818, pages 68-80. Springer-Verlag, 1994.

[3] Y.-A. Chen, E. Clarke, P.-H. Ho, Y. Hoskote, T. Kam, M. Khaira, J. O’Leary, and X. Zhao. Verification of all circuits in a floating-point unit using word-level model checking. In Proceedings of FMCAD96, Lecture Notes in Computer Science 1166, pages 19-48. Springer-Verlag, 1996.

[4] F. Corella, Z. Zhou, X. Song, M. Langevin, E. Cerny, Multiway decision graphs for automated hardware verification, IBM technical report RC19676, July 1994.

[5] P. Cousot and R. Cousot. Abstract Interpretation: a unified lattice model for the static analysis of programs by construction or approximation of fixpoints. In Proceedings of the Fourth Annual Symposium on Principles of Programming Languages. ACM Press, 1977.

[6] D. Cyrluk. Microprocessor verification in PVS: A methodology and simple example. Technical Report SRI-CSL-93-12, SRI Computer Science Laboratory, Dec. 1993.

[7] J. Hennessy and D. Paterson. Computer Architecture: A quantitative Approach, 2nd ed. Morgan Kaufman, 1996.

[8] R. Hojati and R.K. Brayton. Automatic datapath abstraction of hardware systems. In Proceedings of CAV95. Lecture Notes in Computer Science 939, pages 95-113. Springer-Verlag, 1995.

[9] R. Hojati, A.J. Isles, D. Kirkpatrick, and R.K. Brayton. Verification using uninterpreted functions and finite instantiations. In Proceedings of FMCAD96, Lecture Notes in Computer Science 1166, pages 218-232. Springer-Verlag, 1996.

[10] A.J. Isles, R. Hojati and R.K. Brayton. Computing reachable control states of systems modeled with uninterpreted functions and infinite memory, In Proceedings of CAV98, Vancouver, Canada, 1998.

[11] R.B. Jones, C.-J. H. Seger, and D.L. Dill. Self-consistency checking. In Proceedings of FMCAD96, Lecture Notes in Computer Science 1166, pages 159-171. Springer-Verlag, 1996.

[12] J. Levitt, K. Olukotun. Verifying correct pipeline Implementation for microprocessors, In Proceedings of ICCAD97, pages 162-169, IEEE 1997.

[13]K.L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem.* Kluwer Academic Publishers, 1993.

[14]J.L. Peterson, *Petri Nets*, Vol. 9, ACM Computing Surveys, No. 3, Sept. 1977.

[15]A. Pnueli, The temporal logic of programs. Proc. 18th IEEE Symp. on Fund. of Computer Science, 46-57, 1977.

[16]J. Sawada, W.A. Hunt, Jr. Trace table based approach for pipelined microprocessor verification. In Proceedings of CAV97, Lecture Notes in Computer Science 1254, pages 364-375. Springer-Verlag, 1997.

[17]M. Srivas and M. Bickford, Formal verification of a pipelined microprocessor, IEEE Software, 7(5), pages 52-64, Sept. 1990.