

Verification Driven Formal Architecture and Microarchitecture Modeling

Yogesh Mahajan, Carven Chan, Ali Bayazit, Sharad Malik*

Department of EE, Princeton University
Princeton, NJ 08544, USA

{yogism, carvenc, abayazit, sharad}@princeton.edu

Wei Qin

ECE Department, Boston University
Boston, MA 02215, USA

wqin@bu.edu

Abstract

Our ability to verify complex hardware lags far behind our capacity to design and fabricate it. We argue that this gap is partly due to the limitations of RTL models when used for verification. Higher level models such as SystemC and SystemVerilog aim to raise the level of abstraction to enhance designer productivity; however, they largely provide for executable but not analyzable descriptions. We propose the use of formally analyzable design models at two distinct levels above RTL: the architecture and the microarchitecture level. At both these levels, we describe concurrent units of data computation termed transactions. The architecture level describes the computation/state updates in the transactions and their interaction through shared data. The microarchitecture level adds to this the resource usage in the transactions as well as their interaction based on shared resources. We then illustrate the applicability of these models in a top-down verification methodology which addresses several concerns of current methodologies.

1 Introduction

Hardware verification is increasingly recognized as a major limiting factor in modern complex designs. This is reflected in the so-called “verification gap” between the growth rate of designs that can be reasonably verified and designs that can be fabricated [4]. While the primary contributor to the verification gap is the complexity of exploring state spaces exponential in the number of state elements, a less recognized contributor is the absence of a widely applicable verification methodology. Though we have a number of verification techniques and tools - functional simulation, emulation and various flavors of formal verification - we lack a useful methodology which lets us efficiently deploy them in a structured manner to provide a useful notion of completeness or even coverage of verification. We argue that a large contributor to this is the inappropriateness of

popular design models and design flows for this purpose.

1.1 Inappropriate Design Modeling

Register-Transfer-Level (RTL) models written in Verilog/VHDL have functional semantics as well as detailed structural and timing information. However, the RTL does not easily yield information on how the high-level computation being carried out by the design gets mapped to the implementation. Given an RTL design, it is also very difficult to reverse-engineer the algorithm being implemented by the design *at the level of computation on units of data*. Having such high-level information is often critical in practice for verifying the correctness of the design.

Since RTL cannot be used successfully for high-level *specification*, some other description must be used for this - generally an informal natural language document. Such non-analyzable specifications render us unable to precisely answer the question: “What do we need to verify?” This is reflected in difficulty in deriving testbenches for designs, as well as in the difficulty of *automating* verification using formal verification tools.

More recently, specification languages such as ForSpec [2], etc. have been developed to specify temporal properties as assertions which can then become part of the specification. However, these are at best partial specifications.

There is a push to move design to levels of abstraction above the RTL, using languages like SystemC [27] and SystemVerilog [26]. This can potentially increase designer productivity. However, high level models in these languages are largely executable models for use in simulation. This has two serious limitations. First, it is difficult to automate the *formal analysis* of the state space of these models. Thus, testbench creation is manual and their use in formal verification is limited. Second, even though SystemC provides for a stack of levels of abstraction with increasing levels of design detail [27], there is no easy way to *automatically relate models at different levels of abstraction*. Thus, while a design with a greater level of detail may be a refinement of a more abstract design, there is no formal connection between the two descriptions.

*This research is funded by MARCO through the Gigascale Systems Research Center.

It is well recognized that *concurrency* is at the heart of verification complexity. Once the various sources of concurrency are separated, formal verification techniques such as model checking [8] hold the promise of detecting subtle interactions between concurrent components through efficient state space traversal. Thus, capturing concurrency cleanly is a key requirement for any verification-directed modeling technique. In this regard, there is a practical distinction to be made between *shared data* and *shared physical resources*. The interactions due to the shared data arise because of the concurrency in the computation algorithm, while the interactions of the shared resources are due to its eventual implementation.

2 Data Computation based Modeling

The previous section argued that the following characteristics are useful in a modeling framework directed towards verification:

- It should be able to describe behaviors/specifications in terms of computation on units of data.
- It should provide for a concurrency description that separates concurrent interactions due to shared data from those due to shared resources.
- It should be easy to relate models across multiple levels of the framework during the design process.
- It should be formally analyzable.

We now describe a modeling framework with these characteristics. It provides for modeling at two separate levels above RTL - the architecture and μ -architecture levels.

While the proposed modeling framework will have some underlying model(s) of computation (MoC), this is not the focus of this work. The emphasis is on the semantic layer above this that encapsulates hardware design and design flow characteristics. It is this that can be exploited in various verification tasks.

2.1 Architecture Level

The architecture level describes the concurrent algorithm. It describes the:

- **Concurrent components:** Each concurrent component, called a *transaction*, performs a single well-defined computation on units of data. Transactions have clear starting and ending points and always execute as indivisible units. Thus, a transaction is a “unit of work” (e.g. [14]) and serves as a partial specification of the computation being done. For example, in a programmable processor, at the ISA level, the instruction is a transaction which describes how architectural processor state is transformed. For a speech/image/video processing algorithm, a transaction may be the algorithm to process a frame.

- **Data interactions:** All interaction between different architectural transactions is through shared data. There may be different forms of these interactions. Consider a producer-consumer interaction in a signal-processing design. This form of decoupled interaction is modeled in known data flow models such as Kahn Process Networks [15] (as well as in our model) using queues. Coupled interactions such as through a shared variable in a multi-threaded program are modeled using general shared memory. A single design may have both coupled as well as decoupled forms of interaction.

The architecture level does *not* model physical resource constraints. Thus, it is free of many details of the algorithm implementation and serves as a functional specification for the design - how the different components transform and share data. This generalizes the notion of architecture descriptions to beyond programmable processors.

2.2 μ -Architecture Level

The μ -architecture level describes how the concurrent algorithm is implemented. It describes the:

- **Concurrent components:** As in the architectural model, these are the concurrent transactions. However, due to various resource interactions between transactions, transactions no longer execute as indivisible units, and can get blocked when waiting for resources. Thus, not only is the concurrency constrained by true data dependencies, but also by resource availability.
- **Data interactions:** As at the architecture level, the data interactions can be coupled or decoupled.
- **Resource interactions:** The primary addition at the μ -architecture level is how the transactions use resources during the computation. These resources may be shared by multiple transactions. The model describes how resources are requested and arbitrated. This is done by associating a resource manager (an arbiter) with each resource for which requests are made by the transactions. This is related to the concept of token managers in the OSM model [25].

This generalizes the notion of μ -architecture descriptions to beyond programmable processors.

2.3 RTL

This paper focuses on the models at levels above RTL, specifically the architecture and μ -architecture levels. It is important to review how RTL differs from the above two levels, especially the μ -architecture level. Both may be viewed as “hardware implementations” of the algorithm. However, we claim that while RTL describes the “hardware” it does not describe how this is an “implementation” of the algorithm. It is precisely this gap that the μ -

architecture model fills. It describes the “implementation” of the algorithm in terms of how the individual transactions are implemented, by describing what resources are used in each step of the transaction and how these are shared. Thus, it relates the algorithm to the hardware - something that is not there in either the algorithm or the RTL. *In existing verification methodologies this gap between the algorithm and the hardware is completely bridged by human intervention and this is a major limitation.*

We do see a role for RTL in the design flow, and that is serving as the starting point for logic and physical synthesis. A given μ -architecture model may be compiled to an RTL model by synthesizing the control logic needed for sequencing through its transactions as well as for implementing the resource arbiters. While this will undoubtedly open up new optimization problems, control synthesis has been very well studied and overall this compilation seems manageable.

3 Architecture Model

The architectural level captures the functional/algorithmic aspects of the computation and breaks the computation up into “units of work” called transactions. Transactions execute concurrently and interact via data present in shared architectural state.

A simple pipelined processor will be used as a running example for the purpose of illustration. The instructions are: (i) addition, (ii) conditional branch, and (iii) load and store. The instruction and data address spaces are separate.

3.1 Architectural State and Transactions

Definition 3.1 An architectural state element \mathbf{S} is one of the following: a register \mathbf{r} , an indexed memory $\mathbf{M}[\mathbf{i}]$, or an unbounded queue \mathbf{Q} . A register \mathbf{r} provides for storage and retrieval of a single value. A memory \mathbf{M} is a finite (possibly unbounded) sequence of registers, with $\mathbf{M}[\mathbf{i}]$ being the i^{th} register in the sequence. A queue \mathbf{Q} provides for storage of multiple values using the usual FIFO semantics. In addition, each state element may have a value type, e.g. *bool* or *bounded/unbounded integer*.

More generally, the set of architectural state elements need not be limited to registers, memories or queues, and can be expanded to include other abstract data types. Each type of state element has a set of operations defined on it, e.g. queues support the *push*, *pop* and *top* operations which may update state. Certain combinations of operations may not occur together, e.g. two simultaneous writes to the same register cause a conflict.

In the processor example in Fig. 1(a), the architectural state consists of a register **pc** for a program counter, and indexed memories **Reg**, **Mem_inst**, and **Mem_data** for the

register file, instruction, and data memories, respectively. **Reg** is a bounded memory with 128 entries. The value types in all the state elements are fixed bit-width unsigned integers. Initial values can be assigned to each architectural state element, e.g. here the **pc** is initialized to 0.

Each transaction is modeled as a *transaction graph* which describes the steps of the computation (called ‘transaction steps’) as well as the flow of control between these.

In addition to the architectural state, a transaction can use *local* state elements which have a lifetime equal to the lifetime of the transaction. In the processor example, **op1**, **op2**, **iw** and **result** (Fig. 1(a)) are local transaction states.

Definition 3.2 A transaction step σ is a pair $(\Delta_\sigma, \beta_\sigma)$, where $\Delta_\sigma = \{\delta_1, \delta_2, \dots, \delta_{|\Delta_\sigma|}\}$ is a set of concurrent state updates and $\beta_\sigma = \{p_1, p_2, \dots, p_{|\beta_\sigma|}\}$ is a set of Boolean predicates on state elements.

The execution of a transaction step involves the following sequence of steps

1. The set of concurrent state updates Δ_σ is carried out.
2. The Boolean predicates in β_σ are evaluated. These predicates are used to determine the control flow within the transaction graph.

As an example of a transaction step, consider the step labelled D in the transaction graph shown in Fig. 1(b). There are two parallel state updates (assignments to registers **op1** and **op2**) and four predicate assignments in this step. During the execution of step D , the two operand indices are decoded, accessed from the register file, and assigned to the transaction’s **op1**, **op2** registers. The Boolean predicates *is_add*, *is_branch*, etc. are assigned as per the result of the instruction decoding.

Definition 3.3 A transaction graph $\Gamma = (\Sigma, E)$ is an edge-labeled directed graph. The vertex set Σ is a set of transaction steps with a unique source vertex $\sigma_{start} \in \Sigma$ and a unique sink vertex $\sigma_{end} \in \Sigma$. The edges out of each step σ are $\{e_1, e_2, \dots, e_{|\text{outedges}(\sigma)|}\} \subseteq E$. Each $e_k = (\sigma, \sigma_k)$ is labeled with a predicate $p_k \in \beta_\sigma$. The predicate p_1 is always true for all transaction steps.

The transaction graph describes the execution of a transaction as follows: Execution of the transaction begins in σ_{start} . After a transaction step σ executes, the values of the predicates in β_σ are used to determine the next step σ_{next} to execute as follows: the step σ_{next} is pointed to by the out-edge e_k of σ where $k = \max\{k : p_k \text{ is true}\}$. Thus, the edges are prioritized and the highest priority edge with a true predicate is taken. Since p_1 is labeled true, this least priority edge will be taken if none of the other predicates evaluates to true. This process is repeated until the execution eventually reaches σ_{end} .

For the transaction graph in the processor example, Fig. 1(b), one possible execution occurs as follows. There

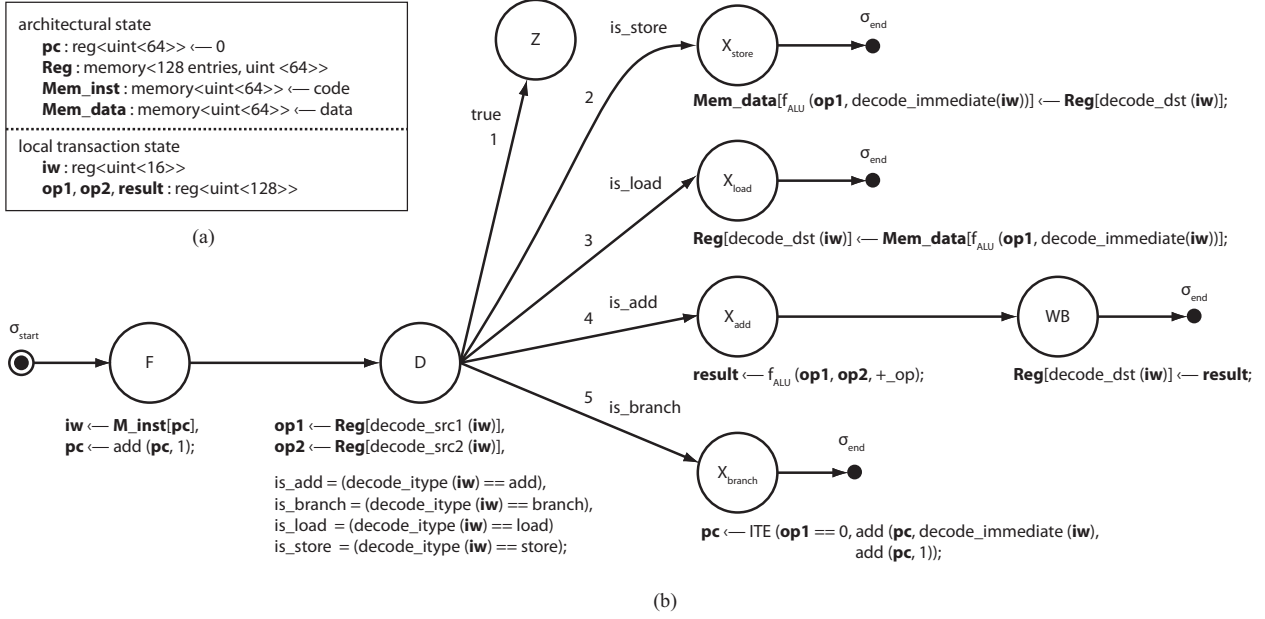


Figure 1. Architectural transaction class for the pipelined processor — (a) state (b) transaction graph

is only one out-edge from σ_{start} which by default has the predicate `true` on it. The executing transaction follows this edge into step F which performs instruction fetch. Another default edge takes us to D which performs the instruction decoding. The next step to execute is determined by the values of the predicates `is_add`, `is_branch`, etc. and the priorities marked on the out-edges of D . (The least priority edge out of D points to an error-handling state Z if the instruction is of unknown type.) Supposing the processor decoded a store instruction, the third transaction step will be X_{store} . From there, the default edge out of X_{store} leads to σ_{end} and the transaction finishes execution.

Definition 3.4 A transaction class T is a tuple (Γ, S) , where Γ is a transaction graph, and S is a set of local state elements which only Γ may update.

A transaction class is merely a template defining a transaction. Every time a transaction executes, a new *transaction instance* is created along with a new set of local transaction states and starts execution from σ_{start} . A transaction instance can make progress out of σ_{start} only when there is at least one true predicate in $\beta_{\sigma_{start}}$ on an edge that leaves σ_{start} (i.e. not a self-loop). Since σ_{start} always executes upon instantiation of new transactions, it should not modify shared state. In Fig. 1(b), the only out edge from σ_{start} has the predicate `true` on it, and the transaction can always make progress from σ_{start} .

3.2 Architecture Model and Semantics

Definition 3.5 An architecture model \mathcal{A} is a pair (S, T) , where S is a set of shared architectural state elements and T is a set of transaction classes.

For the processor example, T contains a single transaction class for an instruction, and S specifies the architectural register and memory state. For multiple processors, the model will consist of one transaction class for each processor, and include the shared memory in the state elements.

Thus far we have described the instantiation of transactions, the semantics of execution of a transaction step, and the flow of control between the transaction steps during the execution of a transaction. We now examine the semantics of the concurrent execution of multiple transactions.

A transaction in the architectural model is indivisible, i.e. if a transaction executes, then it executes all the way to the end. An instantiated transaction, T_1 may execute simultaneously with another instantiated transaction T_2 . In this case, both transactions see the architectural state as it was before the start of both transactions. Thus, any updates by T_1 are not seen by T_2 and vice-versa. Further, T_1 and T_2 cannot execute simultaneously if a state update operation in T_1 conflicts with a state update operation in T_2 e.g. if both write to the same register. Conflicting transactions cause an execution error. For non-conflicting transactions, the architectural state updates at the end of execution of T_1 and T_2 is the union of their updates.

The architectural model allows for flexibility in the concurrency model. Transaction classes may be synchronous, i.e. all instantiated transactions execute simultaneously, asynchronous, i.e. any subset of instantiated transactions executes simultaneously, or a combination of these timing disciplines with transaction classes partitioned into synchronous sets and asynchronous execution across sets. Strict interleaving semantics correspond to asynchronous execution with subsets of size 1.

Given the model $(\mathcal{T}, \mathcal{S})$, and initial values for all state elements, a simulation procedure for the asynchronous concurrency model is as follows:

```

while true do
   $\mathcal{B} \leftarrow \{t : t = \text{instance}(T), T \in \mathcal{T}\}$ 
  execute step  $\sigma_{start}$  for all  $t \in \mathcal{B}$ 
   $\mathcal{A} \leftarrow \{t : t \text{ can progress from } \sigma_{start}, t \in \mathcal{B}\}$ 
  if  $\mathcal{A} = \phi$  then break
   $\mathcal{A} \leftarrow \text{pick\_non\_empty\_subset}(\mathcal{A})$ 
  execute( $\mathcal{A}$ )
end while

```

In each iteration, the simulation scheduler non-deterministically picks any non-empty subset of the instantiated transactions which can make progress. The procedure `execute(\mathcal{A})` then executes all transactions in \mathcal{A} simultaneously. This allows for all possible interleavings of all possible subsets of enabled transactions.

4 μ -Architecture Model

Concurrency in the architecture model is determined by the availability of data values. Concurrency in an implementation is further constrained by the availability of shared resources. These constraints are captured in the μ -architecture model by modeling physical resources as well as introducing the notion of resource managers which arbitrate the use of the limited shared resources.

4.1 Resources and Resource Managers

Definition 4.1 A resource ρ is a token which represents some aspect of the underlying physical hardware.

Definition 4.2 A resource pool R is an indexed collection of resources. A resource can belong to exactly one resource pool. Typically, the resources in a resource pool control access to the same/related hardware functionality.

The examples given below illustrate these definitions.

- In a pipelined processor, the arithmetic logic unit f_{ALU} is shared by multiple executing transactions (instructions). This function can be used by a transaction only after it has obtained the ρ_{ALU} resource to gain exclusive access to f_{ALU} .

- Consider a register file **Reg** in a processor which has two read ports. Then each state operation *read* on **Reg[i]** has to first acquire either the resource $\rho_{readReg-1}$ or $\rho_{readReg-2}$.

We use the functional notation $\Pi(\mathbf{Reg}[i].read) = \{\{\rho_{readReg-1}, \rho_{readReg-2}\}\}$ to denote that in order for a transaction to carry out operation **Reg[i].read** successfully, it must first obtain some resource from the pool $\{\rho_{readReg-1}, \rho_{readReg-2}\}$. In general, $\Pi(obj)$ returns a set of resource pools, such that owning a resource from each of the resource pools is sufficient for using *obj*.

Definition 4.3 A resource manager M associated with a resource pool R is a finite state machine whose inputs are resource requests and whose outputs are resource grants. M allocates resources in R to transactions requesting R , according to some allocation policy. Each resource manager acts independently of other resource managers. The same resource manager may be associated with more than one resource pool.

During its execution, a transaction acquires a resource ρ by making a request to the resource manager for the resource pool containing ρ .

In the processor example, we use stalling to avoid read-after-write (RAW) hazards between pipelined instructions. For each of the 128 registers in the **Reg** register file, we introduce a resource pool with a single resource ρ_i which is required for any read operation on that register. The collection of resource pools $\{\{\rho_1\}, \{\rho_2\}, \dots, \{\rho_{128}\}\}$ is managed by a register file resource manager G as below:

```

for all  $i$  do
  if  $\rho_i$  is free then grant  $\rho_i$  to requesting  $T$ 
end for

```

The transaction T which wishes to write to register location i must obtain the resource ρ_i during the D step, and ρ_i is retained with T till it finishes writing to **Reg[i]** in the WB write-back step. During the resource arbitration phase, the manager G grants the register resource to the the requesting transaction only if it has not been already allocated. Thus, any following instruction which might cause a RAW hazard will stall in the D stage.

(Note that transactions maintain a *data-stationary* [16] view of computation, i.e. each transaction describes the various computation steps on the same unit of data. The resource managers have a *time-stationary* [16] view and describe the control associated with sharing resources between transactions at each point in time.)

4.2 μ -Architectural State and Transactions

The μ -architecture level uses the same state elements as the architecture level. The main *difference* between the μ -

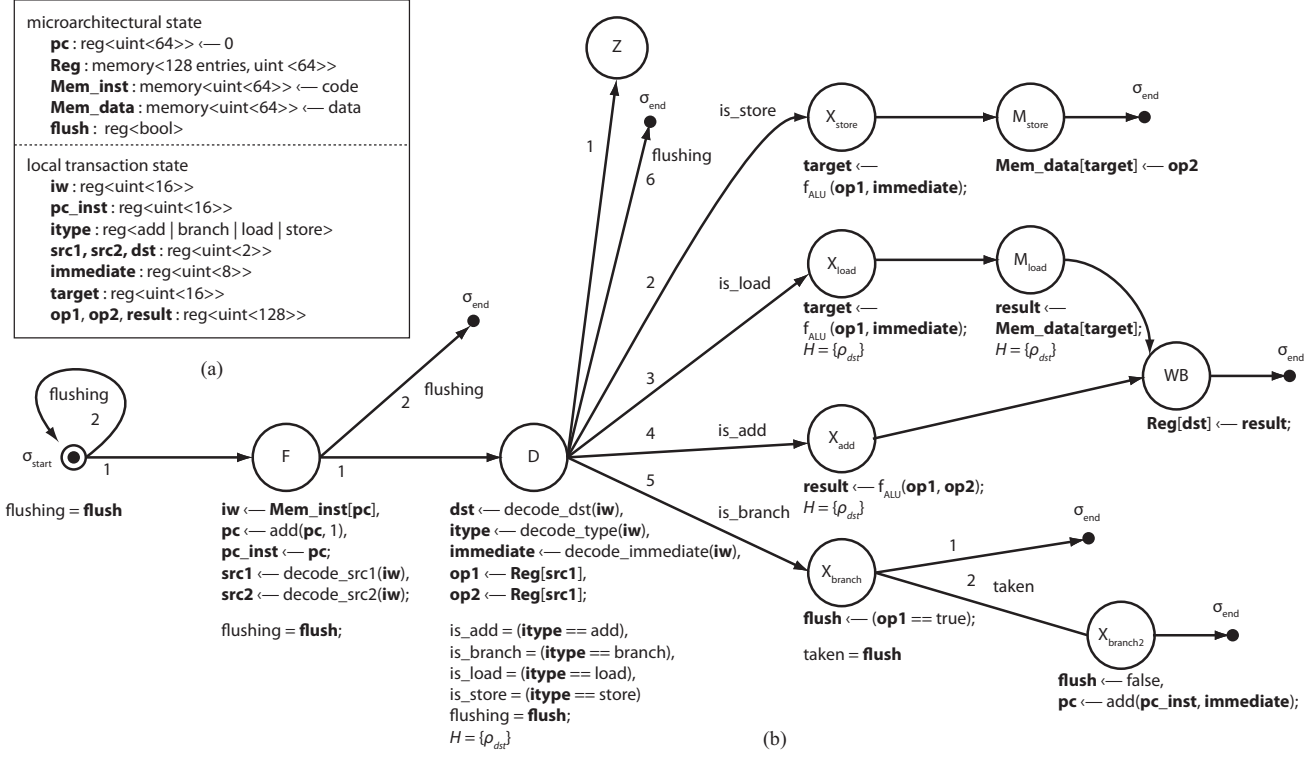


Figure 2. μ -Architectural transaction class for the processor — (a) state (b) transaction graph.

architectural and architectural state is in how the resources associated with the state constrain the *use* of the state elements by μ -architectural transactions. If there is a state resource associated with a state operation, then no transaction step can carry out that operation on the state element unless it possesses the corresponding state resource.

In the processor example in Fig. 2(a), the μ -architecture state consists of **pc**, **Reg**, **Mem_inst**, **Mem_data** and **flush**. This is a superset of the architectural state. The additional register **flush** is used as a signal to other in-flight transactions to conditionally cancel instructions speculatively started in the midst of a branch instruction.

The basic unit of execution at the μ -architectural level is a transaction step.

Definition 4.4 A transaction step σ is a 3-tuple $(\Delta_\sigma, \beta_\sigma, H_\sigma)$, where $\Delta_\sigma = \{\delta_1, \delta_2, \dots, \delta_{|\Delta_\sigma|}\}$ is a set of concurrent state updates, $\beta_\sigma = \{p_1, p_2, \dots, p_{|\beta_\sigma|}\}$ is a set of Boolean predicates and H is a set of resources. All the $p_i \in \beta_\sigma$ involve only local state variables and functions with no associated function-resources. The resources in H_σ are retained after the transaction step completes execution.

Let X be the set of all architectural state operations which are involved in Δ_σ above, and F be the set of all functions referenced in Δ_σ . Then, in order to be able to execute the step σ , the transaction must already have obtained

all the resources in $\Pi(\Delta_\sigma) = \Pi(X) \cup \Pi(F) \cup H_\sigma$. (Other resources such as buses may be similarly acquired.)

Executing a transaction step σ involves the following:

1. Place resource requests for $\Pi(\Delta_\sigma) \setminus P$ where P is the set of resources the step already possesses
2. Wait for the grants from the resource managers
3. Add newly obtained resources to P
4. If $\Pi(\Delta_\sigma) \not\subseteq P$, release all newly acquired resources and end execution. (Execution will be retried later.)
5. If $\Pi(\Delta_\sigma) \subseteq P$, first execute Δ_σ , then evaluate predicates β , and finally release all resources $r \notin H_\sigma$.

Transaction graphs and *transaction classes* are defined as in Defns. 3.3 and 3.4. Since many instances of the same transaction class can execute concurrently, each transaction class also defines an integer *max_instances* which is the maximum number of new instances of type T which can get created at a time. As before, the execution of the transaction begins in σ_{start} . From a step σ , the next step to execute is determined by following edge e_k out of σ where $k = \max\{k : p_k \text{ is true}\}$. This continues until σ_{end} is reached.

Fig. 2(b) illustrates the μ -architectural transaction graph for the processor example. The execution steps for the memory instructions are split into address computation and write-back steps. The f_{ALU} resource pool is accessed during the execution stage of the load, store, and add instructions. There is a register write-back stage common to the

add and load instructions. Branch execution takes two steps because the flush signal (for when the branch is taken) is communicated through shared memory, and needs to be reset. When a branch is determined to be taken, only transactions in the fetch and decode step are speculative and need to be canceled. The box in Fig. 2(a) lists the local transaction state elements which keep instance-specific information.

4.3 μ -Architecture Model and Semantics

Definition 4.5 A μ -architecture model is a 4-tuple $(\mathcal{R}, \mathcal{M}, \mathcal{S}, \mathcal{T})$ where \mathcal{R} , \mathcal{M} , \mathcal{S} and \mathcal{T} are sets containing (in that order) resource pools, resource managers, shared μ -architectural state elements and transaction classes.

At the architecture level, a transaction is an indivisible unit, and the semantics describe which transactions can execute concurrently and how concurrent state updates are performed. Since the μ -architecture model deals with the implementation, at this level, transactions may interact at a finer granularity. Thus, at this level, it is more natural to have the transaction step as the indivisible unit.

The μ -architecture model allows for flexibility in the concurrency model. The transaction steps which have obtained the required resources may execute synchronously i.e. they simultaneously perform their state updates, asynchronously i.e. any subset of them may simultaneously perform their state updates, or a combination of these timing disciplines where transaction classes are partitioned into sets and state updates are synchronous within a set and asynchronous across sets. Strict interleaving semantics corresponds to asynchronous execution with subsets of size 1.

When two steps are simultaneously updating their state, the state seen by both steps during the update is the state as of the start of this simultaneous execution, so no changes in the states are visible outside a transaction till the end of the execution of these steps. As with the architecture model, conflicting operations on state elements such as multiple writes to the same location cannot happen simultaneously and cause an execution error.

A simulation procedure for the model $(\mathcal{R}, \mathcal{M}, \mathcal{S}, \mathcal{T})$ is given here for the fully synchronous case. The simulation begins with an initialization phase which schedules the σ_{start} steps of all transaction classes with an empty set of resources associated with them.

```

 $\mathcal{B} \leftarrow \phi$ 
for all  $T \in \mathcal{T}$  do {instantiate initial}
  for  $1 \leq k \leq T.max\_instances$  do
     $step_k \leftarrow \sigma_{start}(instance(T))$ 
     $\mathcal{B} \leftarrow \mathcal{B} \cup step_k$ 
     $resources(step_k) \leftarrow \phi$ 
  end for
end for

```

The main simulation loop has two phases. In the resource arbitration phase, each step requests resources and the resource managers allocate resources. In the state update phase, those steps which succeed in obtaining the required resources (i.e. all $step \in \mathcal{A}$ below) carry out all their state updates in parallel (execute($\Delta_{\mathcal{A}}$) below), and determine the next steps to be scheduled for execution.

```

while  $\mathcal{B} \neq \phi$  do
  for all  $step \in \mathcal{B}$  do {ask for required resources}
     $step.place\_requests(\Pi(\Delta_{step}) \setminus resources(step))$ 
  end for
  for all  $M \in \mathcal{M}$  do
     $M.do\_arbitration()$ 
  end for
   $\mathcal{A} \leftarrow \phi$ 
  for all  $step \in \mathcal{B}$  do
     $N = \{resources\ newy\ granted\ to\ step\}$ 
     $resources(step) \leftarrow resources(step) \cup N$ 
    if  $\Pi(\Delta_{step}) \subseteq resources(step)$  then
       $\mathcal{A} \leftarrow \mathcal{A} \cup step$ 
    else
       $step.release(N)$ 
      continue {retry in next iteration}
    end if
  end for
  execute( $\Delta_{\mathcal{A}}$ ) {synchronous state update}
  for all  $step \in \mathcal{A}$  do
     $step.release(resources(step) \setminus H_{step})$ 
    evaluate all  $p_i \in \beta_{step}$ 
     $k = \max\{i : p_i \in \beta_{step}\ is\ true\}$ 
     $next \leftarrow e_k.tail$ 
    if is_start_step( $step$ ) &  $next \neq step$  then
       $new \leftarrow \sigma_{start}(instance(transaction\_of(step)))$ 
       $resources(new) \leftarrow \phi$ 
       $\mathcal{B} \leftarrow \mathcal{B} \cup new$ 
    end if
     $step \leftarrow next$ 
    if is_end_step( $step$ ) then {end transaction}
       $\mathcal{B} \leftarrow \mathcal{B} \setminus step$ 
    end if
  end for
end while

```

5 Related Work

We now place our modeling framework in the context of other efforts that overlap with some attribute of our work.

Modeling Concurrency Work done on concurrency semantics (also referred to as models of computation [18]) provides a mathematical foundation for the specification and reasoning of the feasible behavior of concurrent components (e.g. Statecharts [12], Synchronous/Reactive sys-

tems [7], and dataflow process networks [15]). Some of these are closer to our work in that they use tokens to model shared objects (e.g. Petri-nets [20], OSM [25]). However, they do not make explicit the important practical distinction between *shared data* and *shared physical resources*. This distinction is at the core of our modeling methodology and is critical to understanding potential sources of errors and requirements for verification. The actual concurrency semantics (synchronous versus asynchronous state updates) can be flexibly selected depending on the application.

Models such as Bluespec [21] (and Synchronized Transitions [11]) describe computation in terms of guarded actions on state. Control is synthesized through model analysis during the synthesis flow from the specification. In contrast, in our work, large parts of the control are made explicit through the specification of the resource requests and managers, as well as the transaction steps. Further, Bluespec has an *operation-centric* focus (a purely time-stationary view), while our work has a *data-centric* focus (a largely data-stationary view), with clearly demarcated data-unit boundaries. Thus, the ability to reason about individual instructions/packets/frames and their interactions based on shared data and resources is not available in Bluespec. Bluespec’s model is driven by a focus on synthesis; ours is driven by a focus on verification.

Multiple Levels of Models The work done on transaction-level-modeling (TLM) [27] in describing end-to-end computation and communication units does provide for modeling at multiple-levels of abstraction. However, the definitions for each level are at best informal (e.g. programmer level) and thus cannot be used in any systematic way for automated derivation of verification tasks. Even when a model-of-computation is embedded to support specific applications [22], the emphasis is on developing executable models, not analyzable ones. Further, while it is possible to reuse testbenches across the different levels, it is not possible to formally relate the models across levels.

Capturing formal invariants during systematic design refinement (e.g. using HOP [10]) is one approach for relating levels of abstraction. Our models allow many ways of specifying refinements as shown in Sec. 6.3.

System-level design frameworks such as Metropolis [5] and CoFluent [24] do provide for multiple levels of functional abstraction. Metropolis has a general notion of a meta-model that can accommodate other models such as ours. The emphasis in CoFluent is on mapping applications to processors and thus the lower level models are processor blocks.

Processor Description Languages/Tools Descriptions such as ISP [6] and nML [9] focus on the architecture level,

specifying the semantics of individual instructions for use in instruction set simulation and code generation. Descriptions such as Flowcharts [28] and LISA [23] are at the microarchitecture level and used for designing the control logic and cycle-accurate processor simulators. Overall these efforts have a much narrower focus than our work.

6 Verification using the Proposed Models

In this section we examine how the architecture and μ -architecture models are used in verification.

6.1 Functional Simulation

The models suggest notions of simulation coverage and can be used to generate testbenches (automated) that cover:

- every path through every transaction graph (architecture level): This provides computation coverage for each distinct data category, e.g. ALU instructions, conditional branch, etc. for the microprocessor example
- every relative ordering of transactions (architecture level) for access to shared data: This provides coverage for the subtle interactions between the concurrent transactions which are through shared data
- every request made by a transaction for a resource (μ -architecture level), for every possible state of the resource manager

Note that while current methodologies strive for similar coverage goals, they are at a disadvantage because the data, resources and execution paths are not explicit and accessible for automated analysis at the RT level. This gap is filled in partially through human expertise, which is expensive, and backed up with biased random simulation which is only probabilistically successful.

6.2 Formal Verification of Models

We can automatically derive finite-state-models for the transactions (and resource managers at the μ -architecture level) for specific concurrent interactions and use them in proof-techniques techniques such as model checking. While a similar step is performed in current RTL-based methodologies, the abstractions needed to extract a practically usable finite-state-model need greater human input to identify the relevant state variables and to identify the cone of influence for the logic. Further, the transactional description allows for direct specification of properties that deal with sequencing of data combined with temporal ordering of events. For example, for the processor case, checking for a RAW hazard is specified as a transaction i reading from state element s before transaction j writes to s and j precedes i in sequence. Here “precedes” is a sequencing relationship and “before” is a temporal relationship. In

existing RTL based model checking, the “precedes” relationship is not available directly in the model and would need to be captured by some logic and state added to the model through human intervention. In the proposed model, this can be completely automated. (This is the subject of current work.)

6.3 Formal Verification Across Levels

While the architecture and μ -architecture models are useful in themselves, there is additional value to be gained when both are provided for a design. In that case, a correspondence can be stated between objects at the two levels when such a correspondence exists. Let c be a function mapping the objects of an architecture model (α) to the corresponding objects in a μ -architecture model (μ).

- Functions: $c(f^\alpha) = f^\mu$ indicates that f^α at the architectural level is implemented by f^μ at the μ -architecture level.
- State elements: $c(S^\alpha) \subseteq S^\mu$
- Transaction sets: $c(T^\alpha) = T^\mu$.

Given such a correspondence, α can serve as a specification for μ . The problem of verifying that μ is an implementation of α will involve the following tasks.

- Equivalence check functions f^α and $c(f^\alpha)$
- Verify that state s^α is an abstraction of $c(s^\alpha)$, e.g. a queue abstracts a finite sized buffer.
- Check end-to-end correctness of each transaction - assuming that a transaction gets all the resources it requests, verify that the execution path between $\sigma_{start}^\mu \rightsquigarrow \sigma_{end}^\mu$ is functionally equivalent to the path $\sigma_{start}^\alpha \rightsquigarrow \sigma_{end}^\alpha$, for the same data input values. This check can use the properties proved in the previous two steps as lemmas.
- Check data interactions between transactions - verify that each transaction reads proper values (hazard-free).

Such an approach has been shown to be feasible for the case of processor verification [14] with a manual construction of the verification tasks. There, the ISA serves as a specification for the RTL and the desired relation between the two is captured by a “correctness statement” [1]. Note that even for a relatively simple pipelined processor, it takes significant human effort and ingenuity to come up with a “correctness statement” and in verifying it, e.g. using the “flushing” approach [13]. We believe that our models can serve as a framework for both automating and generalizing such approaches beyond processor verification.

6.4 Verifying the RTL

As discussed in Section 2.3, in the proposed methodology the RTL is synthesized from the μ -architecture model by synthesizing:

- for each transaction the sequencer for its steps
- for each resource manager the finite state machine implementing its allocation policy
- the logic for the physical resources
- the interconnections between the resources

Thus, verifying the RTL reduces to verifying that each synthesis task produced correct results. Since each of these synthesis tasks has a much narrower scope, the verification tasks are much simpler than verifying general RTL.

In current design flows, RTL is synthesized to gate level circuits and the check between these two levels reduces to a combinational equivalence check. Similarly, when synthesizing the RTL from the μ -architecture as proposed, the check between these levels is largely an equivalence check.

6.5 Runtime Validation

More recently, an alternative attack on the verification problem has been proposed in the form of runtime validation [3, 19]. In this approach, on-chip logic is used for runtime error detection and correction.

The μ -architecture model has some salient features which make it highly suitable for adding online monitoring/checking and recovery techniques. Transactions provide a clear notion of units of computation which sets a natural scope not only for checking but also for forward/backward error recovery [17], both of which can be implemented as function preserving transformations on the μ -architectural model. Examples of checking techniques are spatial and temporal redundancy. Examples of recovery techniques are: forward recovery techniques such as delayed commits, backward recovery techniques such as check-pointing/rollback and serialization of transactions.

These techniques are well known, but implementing them in RTL is trickier due to the complexity of control logic. In the proposed methodology, μ -architecture modeling backed by a quality synthesis path to RTL can raise the overall level of abstraction and automation of the design process.

7 Conclusions

In this paper we call attention to the role of inappropriate hardware modeling in the growing verification gap between the complexity of designs that can be reasonably verified and those that can be fabricated. We highlight the limitations of the prevalent RTL-centric design methodology, and then present an alternative comprising of models for two distinct levels above RTL.

We then briefly illustrate the use of these models in a verification methodology, highlighting what aspects of verification be performed at each level and how these are facilitated by the information available in these models. While

significant work still needs to be done to tap the full potential here, we believe that the ideas presented in this paper serve as a strong foundation for a disciplined design methodology with significant benefits for verification.

References

- [1] M. Aagaard, B. Cook, N. A. Day, and R. B. Jones. A framework for microprocessor correctness statements. In *CHARME '01: Proceedings of the 11th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 433–448, London, UK, 2001. Springer-Verlag.
- [2] R. Armoni, L. Fix, A. Flaisher, R. Gerth, B. Ginsburg, T. Kanza, A. Landver, S. Mador-Haim, E. Singerman, A. Tiemeyer, M. Y. Vardi, and Y. Zbar. The ForSpec temporal logic: A new temporal property-specification language. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 296–311, 2002.
- [3] T. M. Austin. DIVA: A reliable substrate for deep sub-micron microarchitecture design. In *MICRO 32: Proceedings of the 32nd Annual ACM/IEEE international Symposium on Microarchitecture*, pages 196–207, Washington, DC, USA, 1999. IEEE Computer Society.
- [4] B. Bailey. A new vision for scalable verification. *EE Times*, February 18th 2004.
- [5] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli. Metropolis: An integrated electronic system design environment. *Computer*, 36(4):45–52, 2003.
- [6] M. Barbacci, C. Bell, and D. Siewiorek. ISP: A notation to describe a computer's instruction sets. *IEEE Computer*, Vol.6, Iss.3, pages 22–24, Mar 1973.
- [7] G. Berry and A. Benveniste. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, 1991.
- [8] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.
- [9] A. Fauth, J. Van Praet, and M. Freericks. Describing instruction set processors using nML. In *European Design and Test Conference, 1995. ED&TC 1995, Proceedings.*, pages 503–507, Paris, Mar. 1995.
- [10] G. Gopalakrishnan and R. Fujimoto. Design and verification of the rollback chip using HOP: A case study of formal methods applied to hardware design. *ACM Trans. Comput. Syst.*, 11(2):109–145, 1993.
- [11] M. Greenstreet, K. Li, and J. Straunstrup. Synchronized transitions on multiprocessors. In *System Sciences, 1989. Vol.II: Software Track, Proceedings of the Twenty-Second Annual Hawaii International Conference on*, Vol.2, Iss., 3-6, pages 789–797, Jan 1989.
- [12] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.
- [13] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In David L. Dill, editor, *Proceedings of the Sixth International Conference on Computer-Aided Verification CAV*, volume 818, pages 68–80, Stanford, California, USA, 1994. Springer-Verlag.
- [14] R. Jhala and K. L. McMillan. Microarchitecture verification by compositional model checking. In *CAV '01: Proceedings of the 13th International Conference on Computer Aided Verification*, pages 396–410, London, UK, 2001. Springer-Verlag.
- [15] G. Kahn. The semantics of a simple language for parallel programming. In *Information Processing*, pages 471–475. North-Holland, 1974.
- [16] P. Kogge. *The Architecture of Pipelined Computers*. McGraw-Hill, New York, 1981.
- [17] P. K. Lala, editor. *Self-checking and fault-tolerant digital design*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.
- [18] E. A. Lee and A. Sangiovanni-Vincentelli. A framework for comparing models of computation. In *TCAD, vol. 17, iss. 12*, pages 1217–1229, 1998.
- [19] S. Malik. Keynote: A case for runtime validation of hardware. In *First International Haifa Verification Conference, Revised Selected Papers, LNCS 3875*. Springer, 2006.
- [20] T. Murata. Petri Nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.
- [21] R. S. Nikhil. Bluespec System Verilog: efficient, correct RTL from high-level specifications. In *MEMOCODE*, pages 69–70, 2004.
- [22] H. D. Patel and S. K. Shukla. Towards a heterogeneous simulation kernel for system level models: a SystemC kernel for synchronous data flow models. In *GLSVLSI '04: Proceedings of the 14th ACM Great Lakes symposium on VLSI*, pages 248–253, New York, NY, USA, 2004. ACM Press.
- [23] S. Pees, A. Hoffmann, V. Zivojnovic, and H. Meyr. Lisa-machine description language for cycle-accurate models of programmable dsp architectures. In *DAC '99: Proceedings of the 36th ACM/IEEE conference on Design automation*, pages 933–938, New York, NY, USA, 1999. ACM Press.
- [24] V. Perrier. System architecting complex designs. <http://www.cofluentdesign.com/>, Feb 2004.
- [25] W. Qin and S. Malik. Flexible and formal modeling of microprocessors with application to retargetable simulation. In *DATE '03: Proceedings of the Conference on Design, Automation and Test in Europe*, pages 556–561, Washington, DC, USA, 2003. IEEE Computer Society.
- [26] D. I. Rich. The evolution of SystemVerilog. *IEEE Design and Test of Computers*, 20(04):82–84, 2003.
- [27] S. Swan. SystemC transaction level models and RTL verification. In *DAC '06: Proceedings of the 43rd Annual Conference on Design Automation*, pages 90–92, New York, NY, USA, 2006. ACM Press.
- [28] N. Tredennick. How to flowchart for hardware. *IEEE Computer*, 14(12):87–102, 1981.