Operation-Centric Hardware Description and Synthesis

James C. Hoe, *Member, IEEE,* and Arvind, *Fellow, IEEE*

***Abstract*—The *operation-centric* hardware abstraction is useful for describing systems whose behavior exhibits a high degree of concurrency. In the operation-centric style, the behavior of a system is described as a collection of operations on a set of state elements. Each operation is specified as a predicate and a set of simultaneous state-element updates, which may only take effect in case the predicate is true on the current state values. The effect of an operation's state updates is atomic, that is, the legal behaviors of the system constitute some sequential interleaving of the operations. This atomic and sequential execution semantics permits each operation to be formulated as if the rest of the system were frozen and thus simplifies the description of concurrent systems. This paper presents an approach to synthesize an efficient synchronous digital implementation from an operation-centric hardware-design description. The resulting implementation carries out multiple operations per clock cycle and yet maintains the semantics that is consistent with the atomic and sequential execution of operations. The paper defines, and then gives algorithms to identify, *conflict-free* and *sequentially composable* operations that can be performed in the same clock cycle. The paper further gives an algorithm to generate the hardwired arbitration logic to coordinate the concurrent execution of conflict-free and sequentially composable operations. Lastly, the paper evaluates synthesis results based on the TRAC compiler for the TRSPEC operation-centric hardware-description language. The results from a pipelined processor example show that an operation-centric framework offers a significant reduction in design time, while achieving comparable implementation quality as traditional register-transfer-level design flows.**

***Index Terms*—Conflict-free, high-level synthesis, operation-centric, sequentially composable, term-rewriting systems (TRS).**

I. Introduction

MOST hardware-description frameworks, whether schematic or textual, make use of concurrent state machines (or processes) as their underlying computation model. For example, in a register-transfer-level (RTL) language, the design entities are the individual registers (or other state primitives) and their corresponding next-state equations. We refer to this style of descriptions as *state-centric* because the description of behavior is organized by the individual state element's next-state logic. Alternatively, the behavior of a hardware system can be described as a collection of operations, where each operation can atomically update all or some of the state elements. In this paper, we refer to this style of descriptions as *operation-centric* because the description of behavior is organized into individual operations. Our notion of operation-centric models for hardware description is inspired by the term rewriting systems (TRS) formalism [1]. The operation-centric hardware model of computation is also analogous to Dijkstra's guarded commands [7]. Similar models of computation can be found in some parallel programming languages (e.g., UNITY [5]), hardware-description languages for synchronous and asynchronous design synthesis (e.g., synchronized transitions (ST) [13], [16], [17]), and languages for hardware-design verification (e.g., st2fl [12], SINC [14]). In Section VII, we compare our description and synthesis approach to synchronized transitions.

The operation-centric model of computation consists of a set of state variables and a set of state transition rules. Each atomic transition rule represents one operation. An execution is interpreted as a sequence of atomic applications of the state transition rules. A rule can be optionally guarded by a predicate condition such that a rule is applicable only if the system's state satisfies the predicate condition. If several rules are enabled by the same state, any one of the enabled rules can be *nondeterministically* selected to update the state in one step, and afterwards, a new step begins on the updated new state. With predicated/guarded transition rules, an execution is interpreted as a sequence of atomic rule applications such that each rule application produces a state that satisfies the predicate condition of the next rule to be applied. The atomicity of operation-centric state transition rules simplifies the task of hardware description by permitting the designer to formulate each rule/operation under the assumption that the system is not simultaneously affected by other potentially conflicting operations—the designer does not have to worry about race conditions between different operations. This permits an apparently sequential description of potentially concurrent hardware behaviors.

It is important to note that the simplifying atomic and sequential semantics of operations does not prevent a legal implementation from executing several operations concurrently. Implicit parallelism between operations can be exploited by an optimizing compiler that produces a synchronous implementation that executes several operations concurrently in a clock cycle. However, the resulting concurrent implementation must not introduce new behaviors, that is, behaviors which are not producible under the atomic and sequential execution semantics. This paper presents an approach to synthesize such a syn-

Manuscript received March 11, 2003; revised December 19, 2003. This work was supported in part by the Defense Advanced Research Projects Agency, Department of Defense, under Ft. Huachuca Contract DABT63-95-C-0150 and in part by the Intel Corporation. J. C. Hoe was supported in part by an Intel Foundation Graduate Fellowship during this research. This paper was recommended by Associate Editor R. Gupta.

J. C. Hoe is with the Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA 15213 USA (e-mail: jhoe@ece.cmu.edu).

Arvind is with Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA 02139-4307 USA (e-mail: arvind@csail.mit.edu).

Digital Object Identifier 10.1109/TCAD.2004.833614

0278-0070/04$20.00 © 2004 IEEE

chronous hardware implementation from an operation-centric description. The paper provides algorithms for *detecting* opportunities for the concurrent execution of operations and for *generating* a hardwired arbitration logic to coordinate the concurrent execution in an implementation.

This paper is organized as follows. Section II first presents the abstract transition systems (ATS), an simple operation-centric hardware formalism used to develop and explain the synthesis procedures. Section III next describes the basic synthesis procedure to create a reference implementation that is functionally correct but inefficient. The inefficiencies of the reference implementation are addressed in Sections IV and V using two optimizations based on the concurrent execution of *conflict-free* and *sequentially composable* operations, respectively. The resulting optimized implementation incorporates a hardwired arbitration logic that enables the concurrent execution of conflict-free and sequentially composable operations. Section VI presents a comparison of designs synthesized from an operation-centric description versus an state-centric RTL description. Section VII discusses related work, and Section VIII concludes with a summary.

II. Operation-Centric Transition Systems

We have developed an source-level language called TRSPEC to support operation-centric hardware design specification [9]. TRSPEC's syntax and semantics are adaptations of a well-known formalism, TRS [1]. To avoid the complications of a source language, instead, we present a low-level operation-centric hardware formalism called ATS. ATS is developed as the intermediate representation in our TRSPEC compiler. In this paper, we use ATS to explain how to synthesize an efficient implementation from an operation-centric hardware specification.

A. ATS Overview

The structure of ATS is summarized in Fig. 1. At the top level, an ATS is defined as a triple $\langle \mathcal{S}, \mathcal{S}^o, \mathcal{X} \rangle$. $\mathcal{S}$ is a list of explicitly declared state elements, including registers (R), arrays (A) and first-in-first-out (FIFO) queues (F). $\mathcal{S}^o$ is a list of initial values for the elements in $\mathcal{S}$. $\mathcal{X}$ is a set of operation-centric transitions, where each transition is a pair $\langle \pi, \alpha \rangle$. In a transition, π is a boolean predicate expression, and α is a list of simultaneous actions, exactly one action for each state element in $\mathcal{S}$. (An acceptable action for all state-element types is the null action "nil.") In an ATS, if π of T_x is enabled in a state s [abbreviated as $\pi_{T_x}(s) == \text{true}$, or simply $\pi_{T_x}(s)$], then the simultaneous actions prescribed by α_{T_x} are applied atomically to update s to s'. We use the notation $\delta_{T_x}(s)$ to mean the resulting state s'. In other words, δ_{T_x} is the functional equivalent of α_{T_x}.

B. ATS State Elements and Actions

In this paper, we concentrate on the synthesis of ATS with three types of state elements: R, A, and F. The three types of state elements are depicted in Fig. 2 with signals of their supported interfaces. Below, we describe the three types of state elements and their usage.

$$
\begin{aligned}
\mathcal{ATS} &= \langle \mathcal{S}, \mathcal{S}^o, \mathcal{X} \rangle \\
\mathcal{S} &= \langle \mathsf{R}_1,...,\mathsf{R}_{NR}, \mathsf{A}_1,...,\mathsf{A}_{NA}, \mathsf{F}_1,...,\mathsf{F}_{NF} \rangle \\
\mathcal{S}^o &= \langle v^{R_1},...,v^{R_{NR}}, v^{A_1},...,v^{A_{NA}}\ v^{F_1},...,v^{F_{NF}} \rangle \\
\mathcal{X} &= \{ T_1,...,T_M \} \\
T &= \langle \pi, \alpha \rangle \\
\pi &= exp \\
\alpha &= \langle \mathsf{a}^{R_1},...,\mathsf{a}^{R_{NR}}, \mathsf{a}^{A_1},...,\mathsf{a}^{A_{NA}}, \mathsf{a}^{F_1},...,\mathsf{a}^{F_{NF}} \rangle \\
\mathsf{a}^R &= nil \ [] \ set(exp) \\
\mathsf{a}^A &= nil \ [] \ a\text{-}set(\mathsf{exp}_{idx}, \mathsf{exp}_{data}) \\
\mathsf{a}^F &= nil \ [] \ enq(exp) \ [] \ deq() \ [] \ en\text{-}deq(exp) \ [] \ clear() \\
exp &= constant \ [] \ Primitive\text{-}Op(exp_1, ..., exp_n) \\
&\quad [] \ \mathsf{R}.get() \ [] \ \mathsf{A}.a\text{-}get(idx) \\
&\quad [] \ \mathsf{F}.first() \ [] \ \mathsf{F}.notfull() \ [] \ \mathsf{F}.notempty()
\end{aligned}
$$

Fig. 1. ATS summary.

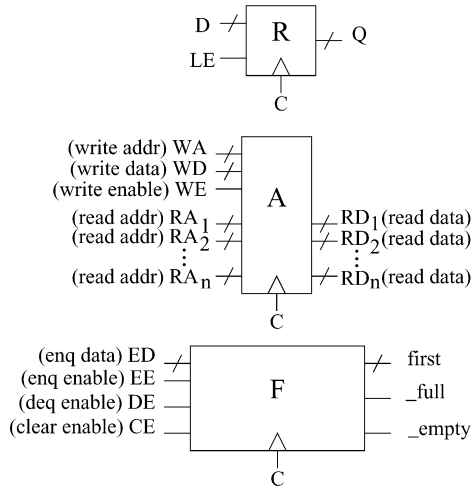


Fig. 2. Synchronous state primitives.

A register can store an integer value up to a specified maximum word size. The value stored in a register can be referenced using the side-effect-free $\mathrm{get}()$ query and set to v using the $\mathrm{set}(v)$ action (written as $r.\mathrm{get}()$ and $r.\mathrm{set}(v)$, respectively). An entry of an array can be referenced using the side-effect-free $a\text{–}\mathrm{get}(idx)$ query and set to v using the $a\text{–}\mathrm{set}(idx, v)$ action. For brevity, we abbreviate $a.a\text{–}\mathrm{get}(idx)$ as $a[idx]$ in our examples. The oldest value in a FIFO can be referenced using the side-effect-free $\mathrm{first}()$ query, and can be removed by the $\mathrm{deq}()$ action. A new value v can be added to a FIFO using the $\mathrm{enq}(v)$ action. The $\mathrm{en\text{–}deq}()$ action is a compound action that conceptually *first* dequeues the oldest entry before enqueuing a new entry. In addition, the contents of a FIFO can be cleared using the $\mathrm{clear}()$ action. The status of a FIFO can be queried using the side-effect-free $\mathrm{notfull}()$ and $\mathrm{notempty}()$ queries. Conceptually, an ATS FIFO has a bounded but unspecified size. The exact size is only set at implementation time, and hence a specification must be valid for all possible sizes. Applying an $\mathrm{enq}()$ to a full FIFO or applying $\mathrm{deq}()$ to an empty FIFO is illegal. (Applying $\mathrm{en\text{–}deq}()$ action to a full FIFO is legal however.) Any transition that queries or acts on a FIFO must include the appropriate $\mathrm{notfull}()$ or $\mathrm{notempty}()$ queries in its predicate condition.

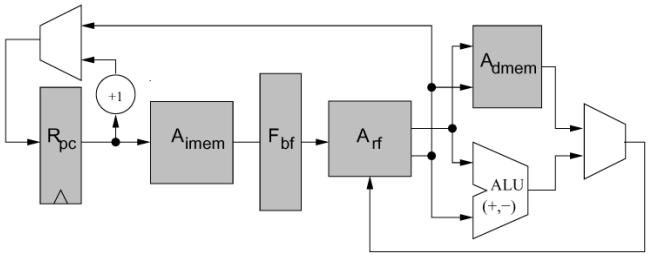


Fig. 3. Simple two-stage pipelined processor example with $\mathcal{S} = \langle \mathsf{R_{pc}}, \mathsf{A_{rf}}, \mathsf{A_{imem}}, \mathsf{A_{dmem}}, \mathsf{F_{bf}} \rangle$.

Besides registers, arrays, and FIFOs, the complete ATS includes register-like state elements for input and output. An input state element (I) is like a register but without the $\mathrm{set}()$ action. A $\mathrm{get}()$ query on an input element returns the value presented on a corresponding external input port. An output state element (O) supports both $\mathrm{set}()$ and $\mathrm{get}()$, and its content is visible to the outside of the ATS on a corresponding output port. For brevity, we omitted the discussion of I and O in this paper; for the most part, I and O are treated as R during synthesis.

C. Pipelined Processor Example

In this example, we describe a two-stage pipelined processor, where a pipeline buffer is inserted between the fetch and execute stages. In an ATS, we use a FIFO, instead of a register, as the pipeline buffer. The buffering property of a FIFO provides the isolation needed to permit the operations in the two stages to be described independently. Although the resulting description reflects an elastic pipeline, our synthesis can infer a legal implementation that operates synchronously and has stages separated by registers (explained further in Section VI-A).

The key state elements and datapath of the two-stage pipelined processor example is shown in Fig. 3. The processor consists of state elements $\mathcal{S} = \langle \mathsf{R_{pc}}, \mathsf{A_{rf}}, \mathsf{A_{imem}}, \mathsf{A_{dmem}}, \mathsf{F_{bf}} \rangle$. The five state elements are: $\mathsf{R_{pc}}$ the program counter, $\mathsf{A_{rf}}$ the register file (an array of integer values), $\mathsf{A_{imem}}$ the instruction memory (an array of instructions), $\mathsf{A_{dmem}}$ the data memory (an array of integer values), and $\mathsf{F_{bf}}$ the pipeline buffer (a FIFO of instructions). In an actual synthesis, the data width of the state elements would be inferred from the type declaration given in the source-level description.

Next, we examine ATS transitions that describe the behavior of this two-stage pipelined processor. Instruction fetching in the fetch stage can be described by the transition $T_{\mathrm{Fetch}} = \langle \pi_{\mathrm{Fetch}}, \alpha_{\mathrm{Fetch}} \rangle$. This transition should fetch an instruction from the current program location in $\mathsf{A_{imem}}$ and enqueuing the instruction into $\mathsf{F_{bf}}$. This transition should be enabled whenever $\mathsf{F_{bf}}$ is not full. In ATS notation

$$\begin{aligned} \pi_{\mathrm{Fetch}} &= \mathsf{F_{bf}}.\mathrm{notfull}() \\ \alpha_{\mathrm{Fetch}} &= \langle \mathsf{a_{Fetch,pc}}, \mathrm{nil}, \mathrm{nil}, \mathrm{nil}, \mathsf{a_{Fetch,bf}} \rangle \\ \mathsf{a_{Fetch,pc}} &= \mathrm{set}\,(\mathsf{R_{pc}}.\mathrm{get}() + 1) \\ \mathsf{a_{Fetch,bf}} &= \mathrm{enq}\,(\mathsf{A_{imem}}\,[\mathsf{R_{pc}}.\mathrm{get}()])\,. \end{aligned}$$

Notice the specification of the T_{Fetch} transition is unconcerned with what happens elsewhere in the pipeline (e.g., if an earlier branch in the pipeline is about to be taken or if the pipeline is about to encounter an exception).

The execution of the different instruction types in the second pipeline stage can be similarly described as separate atomic transitions. First, consider $T_{\mathrm{Add}} = \langle \pi_{\mathrm{Add}}, \alpha_{\mathrm{Add}} \rangle$ for executing an $\mathrm{Add}(rd, r1, r2)$ instruction, where

$$\begin{aligned} \pi_{\mathrm{Add}} &= (\mathsf{F_{bf}}.\mathrm{first}() == \mathsf{Add(rd, r1, r2)}) \wedge \mathsf{F_{bf}}.\mathrm{notempty}() \\ \alpha_{\mathrm{Add}} &= \langle \mathrm{nil}, \mathsf{a_{Add,rf}}, \mathrm{nil}, \mathrm{nil}, \mathsf{a_{Add,bf}} \rangle \\ \mathsf{a_{Add,rf}} &= a - \mathrm{set}\,(\mathsf{rd}, \mathsf{A_{rf}[r1] + A_{rf}[r2]}) \\ \mathsf{a_{Add,bf}} &= \mathrm{deq}(). \end{aligned}$$

T_{Add} is enabled only when the next pending instruction in $\mathsf{F_{bf}}$ is an Add instruction. When applied, its actions carry out the semantics of the Add instruction. Next, consider two separate transitions $T_{\mathrm{BzNotTaken}} = \langle \pi_{\mathrm{BzNotTaken}}, \alpha_{\mathrm{BzNotTaken}} \rangle$ and $T_{\mathrm{BzTaken}} = \langle \pi_{\mathrm{BzTaken}}, \alpha_{\mathrm{BzTaken}} \rangle$ that specify the two possible executions of a $\mathrm{Bz}(rc, ra)$ branch-if-zero instruction

$$\begin{aligned} \pi_{\mathrm{BzNotTaken}} &== (\mathsf{F_{bf}}.\mathrm{first}() = \mathsf{Bz(rc, ra)}) \\ &\quad \wedge (\mathsf{A_{rf}[rc]} \neq 0) \wedge \mathsf{F_{bf}}.\mathrm{notempty}() \\ \alpha_{\mathrm{BzNotTaken}} &= \langle \mathrm{nil}, \mathrm{nil}, \mathrm{nil}, \mathrm{nil}, \mathsf{a_{BzNotTaken,bf}} \rangle \\ \mathsf{a_{BzNotTaken,bf}} &= \mathrm{deq}() \\ \pi_{\mathrm{BzTaken}} &= (\mathsf{F_{bf}}.\mathrm{first}() == \mathsf{Bz(rc, ra)}) \\ &\quad \wedge (\mathsf{A_{rf}[rc]} == 0) \wedge \mathsf{F_{bf}}.\mathrm{notempty}() \\ \alpha_{\mathrm{BzTaken}} &= \langle \mathsf{a_{BzTaken,pc}}, \mathrm{nil}, \mathrm{nil}, \mathrm{nil}, \mathsf{a_{BzTaken,bf}} \rangle \\ \mathsf{a_{BzTaken,pc}} &= \mathrm{set}\,(\mathsf{A_{rf}[ra]}) \\ \mathsf{a_{BzTaken,bf}} &= \mathrm{clear}(). \end{aligned}$$

$\pi_{\mathrm{BzNotTaken}}$ tests for conditions when the next pending instruction is a Bz instruction and the branch condition is not met. The resulting action of $\alpha_{\mathrm{BzNotTaken}}$ is to leave the pipeline state unmodified other than to remove the executed Bz instruction from $\mathsf{F_{bf}}$. On the other hand, when π_{BzTaken} is satisfied, besides setting $\mathsf{R_{pc}}$ to the new branch target, $\alpha_{\mathrm{BzTaken}}$ must also simultaneous clear the content of $\mathsf{F_{bf}}$ because the T_{Fetch} transition given earlier actually follows a simple branch speculation by always incrementing $\mathsf{R_{pc}}$. Thus, if a branch is taken later, $\mathsf{F_{bf}}$ could hold zero or more speculatively fetched wrong-path instructions.

In this two-stage pipeline description, T_{Fetch} in the first stage and a transition in the second stage could become applicable at the same time. Even though conceptually only one transition is to take place in each step, an implementation of this processor description must carry out both fetch and execute transitions in the same clock cycle; otherwise, the implementation does not behave like a pipeline. Nevertheless, the implementation must also ensure that a concurrent execution of multiple transitions produces the same result as a sequentialized execution of the same transitions in some order. In particular, consider the concurrent execution of T_{Fetch} and T_{BzTaken}. Both transitions update $\mathsf{R_{pc}}$ and $\mathsf{F_{bf}}$. In this case, the implementation has to guarantee that these transitions are applied in some sequential order. However, it is interesting to note that the choice of ordering determines how many bubbles are inserted after a taken branch, but it does not affect the processor's ability to correctly execute a program.

III. Synthesizing a Reference Implementation

This section introduces a basic synthesis procedure that maps an operation-centric ATS into an state-centric RTL-level representation. It is assumed that commercial RTL-synthesis tools would complete the synthesis path to a final physical implementation. The synthesis procedure described in this section produces a straightforward implementation that executes only one transition per clock cycle. In this synthesis, the elements of $\mathcal{S}$ are instantiated from a design library to constitute the state elements in the implementation. The transitions in $\mathcal{X}$ are combined to form the next-state logic for the state elements in a three step procedure. The resulting reference implementation is functionally correct but contain many inefficiencies, most particularly its lack of concurrency due to the single-transition-per-cycle restriction. This inefficiency is addressed in Sections IV and V by two increasingly sophisticated optimizations that enables concurrent execution of *conflict-free* and *sequentially composable* transitions.

A. Extraction of π and δ

In this first step, all value expressions in the ATS are mapped to combinational signals evaluated on the current values of the state elements. In particular, this step creates a set of signals, $\pi_{T_1}, \ldots \pi_{T_M}$, that are the π signals of transitions $T_1, \ldots, T_M$ in an M-transition ATS. The logic mapping in this step assumes all required combinational resources are available. Standard RTL optimizations can be applied later to simplify the combinational logic and to share redundant logic.

B. Arbitration Logic

In this step, we create an arbitrator to sequence the execution of enabled transitions. The arbitrator circuit generates the set of arbitrated enable signals $\phi_{T_1}, \ldots, \phi_{T_M}$ based on $\pi_{T_1}, \ldots, \pi_{T_M}$. The block diagram of a generic arbitrator is shown in Fig. 4. Any valid arbitrator must, at least, ensure for any s:

1) $\phi_{T_i} \Rightarrow \pi_{T_i}(\mathsf{s})$;
2) $\pi_{T_1}(\mathsf{s}) \vee \ldots \vee \pi_{T_M}(\mathsf{s}) \Rightarrow \phi_{T_1} \vee \ldots \vee \phi_{T_M}$.

For a single-transition-per-cycle reference implementation, the arbitrator is further constrained to assert at most one ϕ signal in each clock cycle, reflecting the selection of one applicable transition. A priority encoder is a valid arbitrator for the reference implementation.

C. Next-State Logic Composition

Lastly, one *conceptually* begins by creating M independent versions of the next-state logic, each corresponding to one of the M transitions in the ATS. Next, the M versions of next-state logic are merged, state-element by state-element, using the ϕ signals for arbitration at each state element. For example, a particular register may have N transitions that affect it over time. $N \leq M$ because some transitions may not affect the register. The register takes on a new value if any of the N relevant transitions is applicable in a clock cycle. Thus, the register's latch enable is the logical-OR of the ϕ signals from the N relevant transitions. The new value of the register is selected from the N next-state values via a multiplexer controlled by the ϕ signals. Fig. 5 illustrates the merging circuit for a register that can be acted on by two transitions.

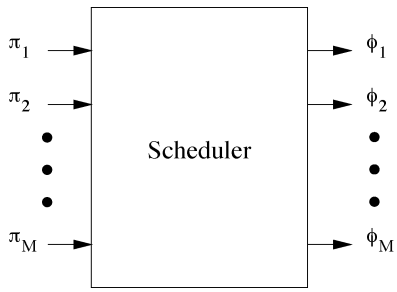


Fig. 4. Monolithic arbitrator for an M-transition ATS.

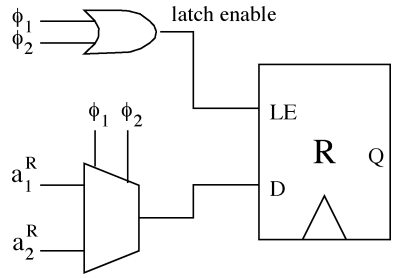


Fig. 5. Circuits for combining two transitions' actions on the same state element.

This merging scheme assumes at most one transition's action is applied to a particular state element in a clock cycle. Furthermore, all the actions of a selected transition must be selected in the same clock cycle at all affected state elements to ensure the appearance of an atomic transition. The two assumptions above are trivially met in a single-transitions-per-cycle implementation. The details of the merging circuit for all three ATS state element types are given next as RTL equations.

For each R, the set of transitions that update R is $\{T_{x_i} | \mathsf{a}^R_{T_{x_i}} == \mathrm{set}(\exp_{x_i})\}$ where $\mathsf{a}^R_{T_{x_i}}$ is the action on R by T_{x_i}. R's data (D) and latch enable (LE) inputs are

$$\mathsf{D} = \phi_{T_{x_1}} \cdot \exp_{x_1} + \ldots + \phi_{T_{x_n}} \cdot \exp_{x_n}$$
$$\mathsf{LE} = \phi_{T_{x_1}} \vee \ldots \vee \phi_{T_{x_n}}$$

For each A, the set of transitions that write A is $\{T_{x_i} | \mathsf{a}^A_{T_{x_i}} == a\text{-}\mathrm{set}(\mathrm{idx}_{x_i}, \mathrm{data}_{x_i})\}$. A's write address (WA), data (WD), and enable (WE) inputs are

$$\mathsf{WA} = \phi_{T_{x_1}} \cdot \mathrm{idx}_{x_1} + \ldots + \phi_{T_{x_n}} \cdot \mathrm{idx}_{x_n}$$
$$\mathsf{WD} = \phi_{T_{x_1}} \cdot \mathrm{data}_{x_1} + \ldots + \phi_{T_{x_n}} \cdot \mathrm{data}_{x_n}$$
$$\mathsf{WE} = \phi_{T_{x_1}} \vee \ldots \vee \phi_{T_{x_n}}.$$

The set of transitions that enqueue a new value into F is $\{T_{x_i} | \mathsf{a}^F_{T_{x_i}} == \mathrm{enq}(\exp_{x_i})) \vee (\mathsf{a}^F_{T_{x_i}} == \mathrm{en\text{–}deq}(\exp_{x_i}))\}$

$$\mathsf{ED} = \phi_{T_{x_1}} \cdot \exp_{x_1} + \ldots + \phi_{T_{x_n}} \cdot \exp_{x_n}$$
$$\mathsf{EE} = \phi_{T_{x_1}} \vee \ldots \vee \phi_{T_{x_n}}.$$

The set of transitions that dequeue from F is $\{T_{x_i} | \, \mathsf{a}^F_{T_{x_i}} == \mathrm{deq}()) \vee (\mathsf{a}^F_{T_{x_i}} == \mathrm{en\text{–}deq}(\exp_{x_i}))\}$

$$\mathsf{DE} = \phi_{T_{x_1}} \vee \ldots \vee \phi_{T_{x_n}}.$$

Similarly, the set of transitions that clear the contents of F is $\{T_{x_i} | \, \mathsf{a}^F_{T_{x_i}} == \mathrm{clear}()\}$

$$\mathsf{CE} = \phi_{T_{x_1}} \vee \ldots \vee \phi_{T_{x_n}}.$$

This conceptual construction based on merging M separate versions of next-state logic would result in a large amount of duplicated logic and resources. In reality, our compiler performs extensive common-subexpression elimination and constant propagation transformations to simplify the synthesized RTL netlist. Furthermore, in the multiple-transition-per-cycle implementations discussed later, our compiler also analyze whether two transitions are mutually-exclusive to enable time-multiplexed reuse of resources such as arithmetic units and interface ports to the state elements. The two primary inefficiencies of this reference implementation are that: 1) only one transitions is permitted per cycle and 2) all transitions must be arbitrated through a centralized arbitration circuit. These two inefficiencies are addressed together by the optimizations in the next two sections.

D. Correctness of the Reference Implementation

The reference implementation given above is not strictly equivalent to the ATS. In the reference implementation, unless the arbitrator employs true randomization, the implementation is always deterministic. In other words, the implementation can only embody one of the behaviors allowed by the ATS. An implementation is said to implement an ATS correctly if: 1) the implementation's sequence of state transitions corresponds to some execution of the ATS and 2) the implementation maintains liveness. Thus, the implementation could enter a livelock if the ATS depends on nondeterminism for forward progress. The reference implementation can use a round-robin priority encoder to ensure *weak-fairness*, that is, a transition is guaranteed to be selected at least once if it remains applicable for a bounded number of consecutive cycles.

IV. Concurrent Execution of Conflict-Free Transitions

Although the semantics of ATS defines an execution in sequential and atomic steps, a hardware implementation can execute multiple transitions concurrently in one clock cycle. However, in a multiple-transitions-per-cycle implementation, the composite state transition taken in each clock cycle must correspond to some sequentialized execution of the constituent ATS transitions. For example, a necessary condition for an implementation in state s to concurrently execute two applicable transitions T_a and T_b is that $\pi_{T_a}(\delta_{T_b}(s)) \vee \pi_{T_b}(\delta_{T_a}(s))$. In other words, after applying the actions of T_b to s, the resulting intermediate state $\delta_{T_b}(s)$ must still satisfy π_{T_a}, or vice versa. Otherwise, concurrent execution of T_a and T_b is not possible since there is not a legal sequential execution of T_a and T_b in two consecutive atomic steps.

There are two approaches to carrying out the effects of T_a and T_b in the same clock cycle. The first approach adds to the reference implementation in Section III the cascaded combinational logic of the two transitions, in essence introducing a brand new transition that is the composition of T_a and T_b. However, arbitrary cascading is not always desirable since it leads to an explosion in circuit size and a longer cycle time. In our approach, T_a and T_b are executed in the same clock cycle only if the correct final state can be reconstructed from an independent and parallel evaluation of their combinational logic on the same starting state. In other words, the resulting multitransition implementation should only require minimal new hardware and have similar (or even shorter) cycle time as compared to the reference implementation. This section first develops an optimization based on the *conflict-free* relationship ($<>_{\mathrm{CF}}$). Section V next presents another optimization based on the *sequential-composability* ($<_{\mathrm{SC}}$) relationship that can expose additional hardware concurrency.

A. Conflict-Free Transitions

$<>_{\mathrm{CF}}$ is a symmetric relationship that imposes a stronger than necessary requirement for executing two transitions concurrently. However, the symmetry of $<>_{\mathrm{CF}}$ permits a more straightforward implementation than $<_{\mathrm{SC}}$. Given T_a and T_b are both applicable in state s, $T_a <>_{\mathrm{CF}} T_b$ implies that applying T_a first does not revoke the applicability of T_b, and vice versa (i.e., $\pi_{T_b}(\delta_{T_a}(s)) \wedge \pi_{T_a}(\delta_{T_b}(s))$). $T_a <>_{\mathrm{CF}} T_b$ further implies that the two transitions can be applied in either order in two successive steps to produce the same final state, $s' == \delta_{T_b}(\delta_{T_a}(s)) == \delta_{T_a}(\delta_{T_b}(s))$. In order to permit a straightforward implementation, $T_a <>_{\mathrm{CF}} T_b$ further requires that an implementation could produce s' by applying the parallel composition of α_{T_a} and α_{T_b} to the same initial state s. The parallel composition function $\mathrm{PC}()$ takes two action lists α_{T_a} and α_{T_b} and returns a new action list that is their pair-wise union; $\mathrm{PC}(\alpha_{T_a}, \alpha_{T_b})$ is undefined if α_{T_a} and α_{T_b} performs conflicting actions on any state element. The conflict-free relationship and the parallel composition function are defined below formally.

Definition 1—Conflict-Free Relationship: Two transitions T_a and T_b are said to be conflict-free ($T_a <>_{\mathrm{CF}} T_b$) if and only if

$$\forall s. \pi_{T_a}(s) \wedge \pi_{T_b}(s) \Rightarrow \pi_{T_b}(\delta_{T_a}(s)) \wedge \pi_{T_a}(\delta_{T_b}(s)) \wedge (\delta_{T_b}(\delta_{T_a}(s)) == \delta_{T_a}(\delta_{T_b}(s)) == \delta_{PC}(s))$$

where δ_{PC} is the functional equivalent of $PC(\alpha_{T_a}, \alpha_{T_b})$. □

Definition 2—Parallel Composition:

$$PC(\alpha_a, \alpha_b) = \langle pc_R(\mathsf{a}^{R_1}, \mathsf{b}^{R_1}), \ldots pc_A(\mathsf{a}^{A_1}, \mathsf{b}^{A_1}), \ldots pc_F(\mathsf{a}^{F_1}, \mathsf{b}^{F_1}), \ldots \rangle$$

where

$$\alpha_a = \langle \mathsf{a}^{R_1}, \ldots \mathsf{a}^{A_1}, \ldots \mathsf{a}^{F_1}, \ldots \rangle$$
$$\alpha_b = \langle \mathsf{b}^{R_1}, \ldots \mathsf{b}^{A_1}, \ldots \mathsf{b}^{F_1}, \ldots \rangle$$
$$pc_R(\mathsf{a},\mathsf{b}) = \text{case } \mathsf{a}\ \mathsf{b} \text{ of}$$
$$\mathsf{a}, \text{nil} \Rightarrow \mathsf{a}$$
$$\text{nil}, \mathsf{b} \Rightarrow \mathsf{b}$$
$$\ldots \Rightarrow \text{undefined}$$
$$pc_A(\mathsf{a},\mathsf{b}) = \text{case } \mathsf{a}\ \mathsf{b} \text{ of}$$
$$\mathsf{a}, \text{nil} \Rightarrow \mathsf{a}$$
$$\text{nil}, \mathsf{b} \Rightarrow \mathsf{b}$$
$$\ldots \Rightarrow \text{undefined}$$
$$pc_F(\mathsf{a},\mathsf{b}) = \text{case } \mathsf{a},\mathsf{b} \text{ of}$$
$$\mathsf{a}, \text{nil} \Rightarrow \mathsf{a}$$
$$\text{nil}, \mathsf{b} \Rightarrow \mathsf{b}$$
$$\text{enq(exp)}, \text{deq}() \Rightarrow \text{en–deq(exp)}$$
$$\text{deq}(), \text{enq(exp)} \Rightarrow \text{en–deq(exp)}$$
$$\ldots \Rightarrow \text{undefined.}$$

□

Below, Theorem 1 extends the composition of two $<>_{\text{CF}}$ transitions to multiple pairwise $<>_{\text{CF}}$ transitions. Theorem 1 states that it is safe to execute in the same cycle, by parallel composition, any number of applicable transitions that are all pairwise $<>_{\text{CF}}$. Each pair from the transitions selected for concurrent execution must be $<>_{\text{CF}}$, because $<>_{\text{CF}}$ is not a transitive. The resulting composite state transition corresponds to the sequential execution of the selected applicable transitions in any order.

Theorem 1—Composition of $<>_{\text{CF}}$ Transitions: Given a collection of n transitions applicable in state s, if all n transitions are pairwise $<>_{\text{CF}}$, then the following holds for any ordering $T_{x_1}, \ldots, T_{x_n}$:

$$\pi_{T_{x_2}}\big(\delta_{T_{x_1}}(\mathsf{s})\big) \wedge \pi_{T_{x_3}}\Big(\delta_{T_{x_2}}\big(\delta_{T_{x_1}}(\mathsf{s})\big)\Big) \wedge \ldots$$
$$\wedge\, \pi_{T_{x_n}}\Big(\delta_{T_{x_{n-1}}}\big(\ldots \delta_{T_{x_3}}\big(\delta_{T_{x_2}}\big(\delta_{T_{x_1}}(\mathsf{s})\big)\big)\ldots\big)\Big)$$
$$\wedge\, \Big(\delta_{T_{x_n}}\Big(\delta_{T_{x_{n-1}}}\big(\ldots \delta_{T_{x_3}}\big(\delta_{T_{x_2}}\big(\delta_{T_{x_1}}(\mathsf{s})\big)\big)\ldots\big)\Big) == \delta_{PC}(\mathsf{s})\Big)$$

where δ_{PC} is the functional equivalent of the parallel compositions of $\alpha_{T_{x_1}}, \ldots, \alpha_{T_{x_n}}$, in any order. (A proof for Theorem 1 is given in [9].) □

B. Static Determination of $<>_{\text{CF}}$

The arbitrator-synthesis algorithm given later in this section is compatible with a conservative test for $<>_{\text{CF}}$, that is, if the test fails to identify a pair of transitions as $<>_{\text{CF}}$, the algorithm generates a less optimal but still correct implementation. A conservative static determination of $<>_{\text{CF}}$ can be made by comparing the read-set and write-set of the transitions. The read-set of a transition is the set of state elements in $\mathcal{S}$ read by the expressions in either π or α. The write-set of a transition is the set of state elements in $\mathcal{S}$ that are acted on by α. For this analysis, the head and the tail of a FIFO are considered to be separate elements. Using $R\text{–set}()$ and $W\text{–set}()$ that extracts the read-set and write-set of a transition, a sufficient condition that ensures $T_a <>_{\text{CF}} T_b$ is

$$\big(R\text{–set}(T_a) \not\cap W\text{–set}(T_b)\big) \wedge \big(R\text{–set}(T_b) \not\cap W\text{–set}(T_a)\big) \wedge \big(W\text{–set}(T_a) \not\cap W\text{–set}(T_b)\big).$$

In other words, conservatively, two transitions are $<>_{\text{CF}}$ if they do not have data (read-after-write) or output (write-after-write) dependence between them.

If two transitions T_a and T_b never become applicable in the same state (i.e., $\forall \mathsf{s}.\neg(\pi_{T_a}(\mathsf{s}) \wedge \pi_{T_b}(\mathsf{s}))$), then they are said to be mutually exclusive, $T_a <>_{\text{ME}} T_b$. Two transitions that are $<>_{\text{ME}}$ also satisfy the definition of $<>_{\text{CF}}$. An exact test for $<>_{\text{ME}}$ requires determining the satisfiability of the expression $(\pi_{T_a}(\mathsf{s}) \wedge \pi_{T_b}(\mathsf{s}))$. Fortunately, the π expression is usually a conjunction of relational constraints on the current values of the state elements. A conservative test that scans two π expressions for contradicting constraints on any one state element works well in practice.

C. Arbitration of Conflict-Free Transitions

By applying Theorem 1, instead of selecting a single transition per clock cycle, an arbitrator (like the one shown in Fig. 4) can select a number of applicable transitions that are all pairwise $<>_{\text{CF}}$. In other words, in each clock cycle, the ϕ signals should satisfy the condition

$$\phi_{T_a} \wedge \phi_{T_b} \Rightarrow T_a <>_{\text{CF}} T_b$$

where ϕ_T is the arbitrated enable signal for transition T. Given a set of applicable transitions in a clock cycle, many different subsets of pairwise $<>_{\text{CF}}$ transitions could exist. Selecting the optimum subset would require weighing the relative importance of the transitions, which is impossible for the arbitrator synthesis algorithm to discern without user feedback or annotation. An objective selection metric is to simply maximize the number of transitions executed in each clock cycle.

Below, we describe the construction of an efficient arbitrator that selects multiple pairwise $<>_{\text{CF}}$ transitions per clock cycle. In a partitioned arbitrator, transitions in $\mathcal{X}$ are first partitioned into as many disjoint arbitration groups, $\mathcal{P}_1, \ldots, \mathcal{P}_k$, as possible such that

$$(T_a \in \mathcal{P}_a) \wedge (T_b \in \mathcal{P}_b) \Rightarrow T_a <>_{\text{CF}} T_b.$$

This partitioning ensures that a transition is $<>_{\text{CF}}$ with any transition not in the same group and, hence, each arbitration group can be arbitrated independently of other groups. Below, we first describe the formation of $<>_{\text{CF}}$ groups and then the arbitration within individual groups.

Step 1) $<>_{\text{CF}}$-**Group Partitioning**: $\mathcal{X}$ can be partitioned into $<>_{\text{CF}}$-arbitration groups by finding the connected components of an undirected conflict graph whose nodes are transitions $T_1, \ldots, T_M$ and whose edges are $\{(T_i, T_j) | \neg(T_i <>_{\text{CF}} T_j)\}$. Each connected component in the conflict graph is a $<>_{\text{CF}}$-arbitration group. For a given conflict graph, the partitioning into $<>_{\text{CF}}$-arbitration groups is unique. For example, the undirected graph (a) in Fig. 6 depicts the $<>_{\text{CF}}$ relationships in an ATS with six transitions. Graph (b) in Fig. 6 gives the corresponding conflict graph. The conflict graph has

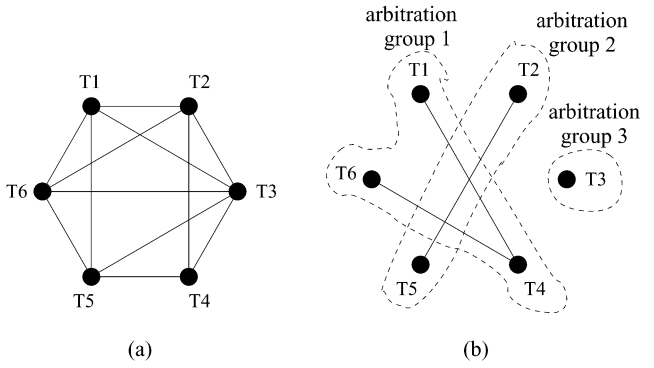


Fig. 6. Analysis of conflict-free transitions: (a) Conflict-free graph and (b) Corresponding conflict graph and its connected components.

three connected components, corresponding to the three $<>_{\mathrm{CF}}$-arbitration groups.

Step 2) $<>_{\mathrm{CF}}$-**Group Arbitrator**: For a given $<>_{\mathrm{CF}}$-arbitration group containing $T_{x_1}, \ldots, T_{x_n}$, $\phi T_{x_1}, \ldots, \phi T_{x_n}$ can be generated locally within the group from $\pi_{T_{x_1}}(s), \ldots, \pi_{T_{x_n}}(s)$ using a priority encoder. However, additional concurrency within a $<>_{\mathrm{CF}}$-arbitration group is possible.

D. Enumerated Group Arbitrator

For example, Arbitration Group 1 in Fig. 6 contains three transitions $\{T_1, T_4, T_6\}$, such that $T_1 <>_{\mathrm{CF}} T_6$, but neither T_1 nor T_6 is $<>_{\mathrm{CF}}$ with T_4. Although the three transitions cannot be arbitrated independently of each other, T_1 and T_6 can be selected together as long as T_4 is not selected in the same clock cycle. In general, for a given state s, each group arbitrator can independently select multiple pairwise-$<>_{\mathrm{CF}}$ transitions within its arbitration group.

For a $<>_{\mathrm{CF}}$-arbitration group with transitions $T_{x_1}, \ldots, T_{x_n}$ and $\phi_{T_{x_1}}, \ldots, \phi_{T_{x_n}}$ can be computed locally by a combinational function that is equivalent to a $2^n \times n$ lookup table indexed by $\pi_{T_{x_1}}(s), \ldots, \pi_{T_{x_n}}(s)$. The binary data value $d_1, \ldots, d_n$ at the table entry with binary index $b_1, \ldots, b_n$ can be determined by finding a maximal clique in an undirected graph whose nodes $\mathcal{N}$ and edges $\mathcal{E}$ are defined as follows:

$$\mathcal{N} = \{T_{x_i} | b_i \text{ is asserted}\}$$
$$\mathcal{E} = \left\{ \left(T_{x_i}, T_{x_j}\right) | \left(T_{x_i} \in \mathcal{N}\right) \wedge \left(T_{x_j} \in \mathcal{N}\right) \wedge \left(T_{x_i} <>_{\mathrm{CF}} T_{x_j}\right) \right\}.$$

This is the conflict-free graph of the T_{x_i} subset that corresponds to the asserted positions in the binary index $b_1, \ldots, b_n$. For each T_{x_i} that is in the selected clique, assert d_i. For example, Arbitration Group 1 from Fig. 6 can be arbitrated using the enumerated lookup table in Fig. 7. The lookup table selects T_1 and T_6 when they are applicable together. This lookup table also reflects an arbitrary decision that T_1 should be selected instead of T_4 if T_1 and T_4 are both applicable in the same cycle. A similar decision is imposed between T_4 and T_6. This freedom in $<>_{\mathrm{CF}}$-group arbitrator generation highlights the fact that, whereas the partitioning of independent $<>_{\mathrm{CF}}$ arbitration groups is unique, the maximal asserted subset of ϕ's for a given combination of

π_{T_1}	π_{T_4}	π_{T_6}	ϕ_{T_1}	ϕ_{T_4}	ϕ_{T_6}
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	0
0	1	1	0	1	0
1	0	0	1	0	0
1	0	1	1	0	1
1	1	0	1	0	0
1	1	1	1	0	1

Fig. 7. Enumerated arbitration lookup table for Arbitration Group 1 in Fig. 6.

$\pi_{T_{x_1}}(s), \ldots, \pi_{T_{x_n}}(s)$ is not unique. In general, the generation of $<>_{\mathrm{CF}}$-group arbitrator can benefit from user input in selecting the most "productive" subset (which may not be the maximal subset).

E. Performance Gain

When $\mathcal{X}$ can be partitioned into arbitration groups, the individual partitioned arbitrators are smaller and faster than the single monolithic encoder in the reference implementation from Section III. The partitioned arbitrators also reduce wiring cost and delay since π and ϕ signals of unrelated transitions are not brought together for arbitration.

The property of the parallel composition function ensures that transitions are $<>_{\mathrm{CF}}$ only if their actions do not conflict at any state element. Hence, the state update logic from the reference implementation can be used unchanged in $<>_{\mathrm{CF}}$-arbitrated implementations. Consequently, combinational delay of the next-state logic is not increased by the $<>_{\mathrm{CF}}$ optimization. All in all, a $<>_{\mathrm{CF}}$-arbitrated implementation should achieve better performance than its corresponding reference implementation by allowing more transitions to execute in a clock cycle without increasing the cycle time.

V. Concurrent Execution of Sequentially Composable Transitions

As noted in Section IV-A, the $<>_{\mathrm{CF}}$ relationship is a sufficient but not necessary condition for concurrent execution of multiple transitions in the same clock cycle. This section presents a more exact requirement to extract additional concurrency from within a $<>_{\mathrm{CF}}$-arbitration group.

A. Sequentially Composable Transitions

Consider the following ATS example:

$$\mathcal{S} = \langle \mathsf{R}_1, \mathsf{R}_2, \mathsf{R}_3 \rangle$$
$$\mathcal{X} = \{ \langle \mathrm{true}, \langle \mathrm{set}(\mathsf{R}_2 + 1), \mathrm{nil}, \mathrm{nil} \rangle \rangle, \langle \mathrm{true}, \langle \mathrm{nil}, \mathrm{set}(\mathsf{R}_3 + 1), \mathrm{nil} \rangle \rangle, \langle \mathrm{true}, \langle \mathrm{nil}, \mathrm{nil}, \mathrm{set}(\mathsf{R}_1 + 1) \rangle \rangle \}.$$

In this ATS, all transitions are always applicable. Furthermore, $\mathrm{PC}(\alpha_{T_a}, \alpha_{T_b})$ and its functional equivalent δ_{PC} are well-defined for any choice of two transitions T_a and T_b. Nevertheless, the $<>_{\mathrm{CF}}$ arbitrator proposed in the previous section would not permit T_a and T_b to execute in the same clock cycle because

$\delta_{T_a}(\delta_{T_b}(s)) \neq \delta_{T_b}(\delta_{T_a}(s))$ in general. A more careful re-examination should reveal that, for all s, $\delta_{\mathrm{PC}}(s)$ is always consistent with at least one ordering of sequentially executing T_a and T_b. Hence, the concurrent execution of any two transitions above can be supported efficiently in an implementation. On the other hand, the concurrent execution of all three transitions in a parallel composition is not possible due to circular data dependencies among the three transitions. These considerations are captured by the sequential composability relationship ($<_{\mathrm{SC}}$).

$<_{\mathrm{SC}}$ is a relaxation of $<>_{\mathrm{CF}}$. In particular, $<_{\mathrm{SC}}$ is not symmetric. The intuition behind $<_{\mathrm{SC}}$ is that concurrent execution of a set of transitions does not need to produce the same result as all possible sequential execution of those transitions, just one such sequence. Hence, given T_a and T_b that are applicable in state s, $T_a <_{\mathrm{SC}} T_b$ only requires the concurrent execution of T_a and T_b on s to correspond to $\delta_{T_b}(\delta_{T_a}(s))$, but not necessarily to $\delta_{T_a}(\delta_{T_b}(s))$. Similarly, $T_a <>_{\mathrm{CF}} T_b$ only implies $\pi_{T_b}(\delta_{T_a}(s))$ and does not require $(\pi_{T_b}(\delta_{T_a}(s)) \wedge \pi_{T_a}(\delta_{T_b}(s)))$.

Concurrent execution of sequentially composable transitions requires an asymmetric composition function SC() to combine the action lists from two transitions. Like PC(), SC() returns a new action list by composing actions on the same element from two action lists. The state-specific composition functions $\mathrm{sc}_R()$, $\mathrm{sc}_A()$ and $\mathrm{sc}_F()$ are the same as $\mathrm{pc}_R()$, $\mathrm{pc}_A()$ and $\mathrm{pc}_F()$ except in two cases where SC() allows conflicting actions to be sequentialized. First, $\mathrm{sc}_R(\mathrm{set}(\mathrm{exp}_a), \mathrm{set}(\mathrm{exp}_b))$ is $\mathrm{set}(\mathrm{exp}_b)$ since the effect of the first action is overwritten by the second in a sequential application. Second, $\mathrm{sc}_F(\mathsf{a}, \mathrm{clear}())$ returns $\mathrm{clear}()$ since regardless of a, applying $\mathrm{clear}()$ leaves the FIFO emptied.

Below, the sequential composability relationship and the sequential composition function are defined formally.

Definition 3—Sequential Composability: Two transitions T_a and T_b are said to be sequentially composable ($T_a <_{\mathrm{SC}} T_b$) if and only if

$$\forall s. \pi_{T_a}(s) \wedge \pi_{T_b}(s) \Rightarrow \pi_{T_b}\big(\delta_{T_a}(s)\big) \wedge \Big(\delta_{T_b}\big(\delta_{T_a}(s)\big) == \delta_{\mathrm{SC}}(s)\Big)$$

where δ_{SC} is the functional equivalent of $\mathrm{SC}(\alpha_{\mathrm{T_a}}, \alpha_{\mathrm{T_b}})$. □

Definition 4—Sequential Composition:

$$\mathrm{SC}(\alpha_a, \alpha_b) = \big\langle \mathrm{sc}_R(\mathsf{a}^{R_1}, \mathsf{b}^{R_1}), \ldots \mathrm{sc}_A(\mathsf{a}^{A_1}, \mathsf{b}^{A_1}), \ldots \mathrm{sc}_F(\mathsf{a}^{F_1}, \mathsf{b}^{F_1}), \ldots \big\rangle$$

where

$$\alpha_a = \langle \mathsf{a}^{R_1}, \ldots \mathsf{a}^{A_1}, \ldots \mathsf{a}^{F_1}, \ldots \rangle$$
$$\alpha_b = \langle \mathsf{b}^{R_1}, \ldots \mathsf{b}^{A_1}, \ldots \mathsf{b}^{F_1}, \ldots \rangle$$

$$\mathrm{sc}_R(\mathsf{a}, \mathsf{b}) = \text{case } \mathsf{a}, \mathsf{b} \text{ of}$$
$$\mathsf{a}, \mathrm{nil} \Rightarrow \mathsf{a}$$
$$\ldots \Rightarrow \mathsf{b}$$

$$\mathrm{sc}_A(\mathsf{a}, \mathsf{b}) = \text{case } \mathsf{a}, \mathsf{b} \text{ of}$$
$$\mathsf{a}, \mathrm{nil} \Rightarrow \mathsf{a}$$
$$\mathrm{nil}, \mathsf{b} \Rightarrow \mathsf{b}$$
$$\ldots \Rightarrow \text{undefined}$$

$$\mathrm{sc}_F(\mathsf{a}, \mathsf{b}) = \text{case } \mathsf{a}, \mathsf{b} \text{ of}$$
$$\mathsf{a}, \mathrm{nil} \Rightarrow \mathsf{a}$$
$$\mathrm{nil}, \mathsf{b} \Rightarrow \mathsf{b}$$
$$\mathrm{enq}(\mathrm{exp}), \mathrm{deq}() \Rightarrow \mathrm{en\text{–}deq}(\mathrm{exp})$$
$$\mathrm{deq}(), \mathrm{enq}(\mathrm{exp}) \Rightarrow \mathrm{en\text{–}deq}(\mathrm{exp})$$
$$\mathsf{a}, \mathrm{clear}() \Rightarrow \mathrm{clear}()$$
$$\ldots \Rightarrow \text{undefined}.$$

□

Based on the above definitions, a sufficient condition for $T_a <_{\mathrm{SC}} T_b$ is

$$\big(R\text{–set}(T_b) \not\cap W\text{–set}(T_a)\big) \wedge \big(\mathrm{SC}(\alpha_{T_a}, \alpha_{T_b}) \text{ is defined}\big)$$

where R–set() and W–set() extract the read-set and write-set of a transition as explained in Section IV-B. Notice, unlike in the conservative condition for $<>_{\mathrm{CF}}$, $<_{\mathrm{SC}}$ permits $W\text{–set}(T_a) \cap W\text{–set}(T_b)$ to be nonempty as long as $\mathrm{SC}(\alpha_{T_a}, \alpha_{T_b})$ is defined.

Like in the discussion of $<>_{\mathrm{CF}}$, the $<_{\mathrm{SC}}$ relationship for two transitions can be extended to enable multiple transitions to be composed. More specifically, Theorem 2 below extends sequential composition to multiple transitions that are all pair-wise $<_{\mathrm{SC}}$ and whose transitive closure on $<_{\mathrm{SC}}$ is ordered. The ordering requirement is necessary to ensure the selected transitions are not circularly dependent as described in the opening paragraphs of this section.

Theorem 2—Composition of $<_{\mathrm{SC}}$ Transitions: Given a sequence of n transitions, $T_{x_1}, \ldots, T_{x_n}$, that are all applicable in state s, if $T_{x_j} <_{\mathrm{SC}} T_{x_k}$ for all $j < k$, then

$$\pi_{T_{x_2}}(\delta_{T_{x_1}}(s)) \wedge \pi_{T_{x_3}}\Big(\delta_{T_{x_2}}\big(\delta_{T_{x_1}}(s)\big)\Big) \wedge \ldots$$
$$\wedge\, \pi_{T_{x_n}}\Big(\delta_{T_{x_{n-1}}}\big(\ldots \delta_{T_{x_3}}(\delta_{T_{x_2}}(\delta_{T_{x_1}}(s)))\ldots\big)\Big)$$
$$\wedge \Big(\delta_{T_{x_n}}\big(\delta_{T_{x_{n-1}}}(\ldots \delta_{T_{x_3}}(\delta_{T_{x_2}}(\delta_{T_{x_1}}(s)))\ldots)\big) == \delta_{\mathrm{SC}}(s)\Big).$$

where δ_{SC} is the functional equivalent of the nested sequential composition $\mathrm{SC}(\ldots SC(\mathrm{SC}(\alpha_{T_{x_1}}, \alpha_{T_{x_2}}), \alpha_{T_{x_3}}), \ldots)$. (A proof for Theorem 2 is given in [9].) □

B. Arbitration of $<_{\mathrm{SC}}$ Transitions

By applying Theorem 2, the arbitrator of a $<>_{\mathrm{CF}}$-arbitration group (from Step 2, Section IV-C) can select in each state s the set of applicable transitions such that there exists an ordering of those transitions $T_{y_1}, \ldots, T_{y_m}$, where $T_{y_j} <_{\mathrm{SC}} T_{y_k}$ if $j < k$.

Similar to the formation of the $<>_{\mathrm{CF}}$-arbitration group in Section IV-C, the construction of a partitioned $<_{\mathrm{SC}}$ arbitrator involves finding $<_{\mathrm{SC}}$ arbitration subgroups as the connected components in the further relaxed (with fewer edges) conflict graph of a $<>_{\mathrm{CF}}$-arbitration group. In the conflict graph for $<_{\mathrm{SC}}$ analysis, the nodes are $T_{x_1}, \ldots, T_{x_n}$, and the edges are

$$\Big\{ \big(T_{x_i}, T_{x_j}\big) \,|\, \Big(\big(T_{x_i}, T_{x_j}\big) \notin \mathcal{E}_{\mathrm{SC}} \Big) \wedge \Big(\big(T_{x_j}, T_{x_i}\big) \notin \mathcal{E}_{\mathrm{SC}} \Big) \wedge \neg \big(T_{x_i} <>_{\mathrm{CF}} T_{x_j}\big) \Big\}$$

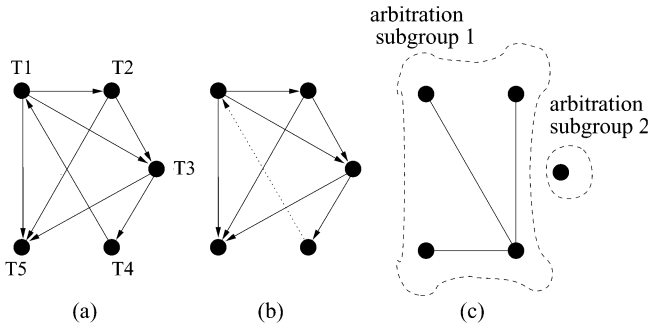


Fig. 8. Analysis of sequentially composable transitions. (a) Directed $<_{\mathrm{SC}}$ graph. (b) Corresponding acyclic directed $<_{\mathrm{SC}}$ graph. (c) corresponding conflict graph and its connected components.

where $\mathcal{E}_{\mathrm{SC}}$ is

$$\left\{\left(T_{x_i}, T_{x_j}\right) \mid \left(T_{x_i} <_{\mathrm{SC}} T_{x_j}\right) \wedge \neg \left(T_{x_i} <>_{\mathrm{CF}} T_{x_j}\right)\right\}.$$

Two nodes are connected if their corresponding transitions are related by neither $<>_{\mathrm{CF}}$ nor $<_{\mathrm{SC}}$. As an added complication, $\mathcal{E}_{\mathrm{SC}}$ considered in this analysis must be acyclic to satisfy the SC-ordering requirement in Theorem 2 (to avoid the concurrent execution of circularly dependent transitions). Therefore, the conflict graph must be based on a version of $\mathcal{E}_{\mathrm{SC}}$ that has been made acyclic by removing cycle-inducing edges. Given an acyclic $\mathcal{E}_{\mathrm{SC}}$, the ordering assumed in Theorem 2 agrees with any topological sort of the corresponding acyclic $<_{\mathrm{SC}}$ graph. (We refer to this ordering as the SC-ordering below.)

Fig. 8 provides an example of $<_{\mathrm{SC}}$ analysis. The directed graph (a) in Fig. 8 depicts the original sequential composability relationships in an $<>_{\mathrm{CF}}$-arbitration group with five transitions. A directed edge from T_a to T_b implies $T_a <_{\mathrm{SC}} T_b$. Edges between T_1, T_3, and T_4 form a cycle., Graph (b) in Fig. 8 shows the acyclic $<_{\mathrm{SC}}$ graph when the edge from T_4 to T_1 is removed. Graph (b) yields the SC-ordering of T_1, T_2, T_3, T_4, and T_5. (The order of T_4 and T_5 can be reversed also.) For a given cyclic $<_{\mathrm{SC}}$ graph, multiple acyclic derivations are possible depending which edge is removed from a cycle. Other possibilities in this example would be to remove the edge from T_1 to T_3 or from T_3 to T_4.

Graph (c) in Fig. 8 gives the corresponding conflict graph. The connected components in the conflict graph form the $<_{\mathrm{SC}}$-arbitration subgroups. Transitions in different $<_{\mathrm{SC}}$-arbitration subgroups are either conflict-free or sequentially-composable, and each subgroup can be arbitrated independently. ϕ's for the transitions in a $<_{\mathrm{SC}}$-arbitration subgroup can be generated locally within the subgroup using a priority encoder. For example, conflict graph (c) in Fig. 8 has two connected components, corresponding to two $<_{\mathrm{SC}}$-arbitration subgroups. In the unary Arbitration Subgroup 2, $\phi_{T_3} = \pi_{T_3}$ without any arbitration. The ϕ signals for transitions in Arbitration Subgroup 1 can be generated using a priority encoding of their corresponding π's. More transitions in Arbitration Subgroup 1 can be selected concurrently using an enumerated combinational lookup table logic similar to the one described in Section IV-D. Notice, if the cycle in the original SC graph (a) had not been broken, T_1 would be a third separate component in the conflict graph; this could lead to a incorrect condition where T_1, T_3, and T_4 are enabled together.

C. Complications to the State-Update Logic

When $<_{\mathrm{SC}}$ transitions are allowed in the same clock cycle, the register-update logic cannot assume that at most one transition acts on a register in each clock cycle. When multiple actions are enabled for a register, the register update logic should ignore all except for the latest action with respect to the SC-ordering of a $<>_{\mathrm{CF}}$-arbitration group. (It can be shown that all transitions that could update the same register in the same cycle are in the same $<>_{\mathrm{CF}}$-arbitration group.) For each R in $\mathcal{S}$, the set of transitions that update R is $\{T_{x_i} |\ \mathsf{a}_{T^R_{x_i}} == \mathrm{set}(\exp_{x_i})\}$. The register's latch enable signal remains

$$\mathsf{LE} = \phi_{T_{x_1}} \vee \ldots \vee \phi_{T_{x_n}}.$$

However, a new data-input signal must be used with an $<_{\mathrm{SC}}$ arbitrator to observe SC-ordering. The new signal is

$$\mathsf{D} = \phi_{T_{x_1}} \cdot \psi_{T_{x_1}} \cdot \exp_{x_1} + \ldots + \phi_{T_{x_n}} \cdot \psi_{T_{x_n}} \cdot \exp_{x_n}$$

where $\psi_{T_{x_i}} = \neg(\phi_{T_{y_1}} \vee \phi_{T_{y_2}} \vee \ldots)$. The expression $\psi_{T_{x_i}}$ contains $\phi_{T_{y_i}}$'s from the set of transitions

$$\{T_{y_i} | \mathsf{R} \in W\text{–set}\left(\alpha_{T_{y_i}}\right) \wedge T_{x_i} \text{ comes before } T_{y_i} \text{ in the SC–ordering} \wedge \neg(T_{x_i} <>_{\mathrm{ME}} T_{y_i})\}.$$

In essence, the register's data input (D) is selected through a prioritized multiplexer that gives higher priority to transitions later in the SC-ordering. Under the current definition of $\mathrm{SC}()$, the update logic for arrays and FIFOs can remain unchanged from Section III.

VI. Evaluation and Results

The synthesis procedures outlined in the previous sections have been implemented in the term architectural complier (TRAC). TRAC accepts TRSPEC descriptions and outputs synthesizable RTL descriptions in the Verilog hardware description language. This section discusses the results from applying TRSPEC and TRAC to the design and synthesis of a five-stage pipelined implementation of the MIPS R2000 ISA [10].

A. Synchronous Pipeline Synthesis

As in the processor from Section II-C, the MIPS processor is described as a pipeline whose five stages are separated by FIFOs. The exact depth of the FIFOs is not specified in the description and, hence, TRAC is allowed to instantiate one-deep FIFOs (basically registers) as pipeline buffers. Flow-control logic is added to ensure the single-register FIFOs are not overflowed or underflowed by enqueue and dequeue actions. In

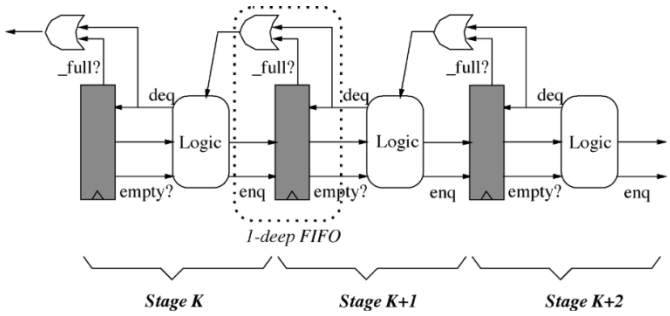


Fig. 9. Synchronous pipeline with combinational multistage feedback flow control.

a naive construction, the single-register FIFO is full if its register holds valid data; the FIFO is empty if its register holds a *bubble*. With only local flow control between neighboring stages, the overall pipeline would contain a bubble in every other stage in the steady state. For example, if pipeline buffer K and $K+1$ are occupied and buffer $K+2$ is empty, the operation in stage $K+1$ would be enabled to advance at the clock edge, but the operation in stage K is held back because buffer $K+1$ appears full during the same clock cycle. The operation in stage K is not enabled until the next clock cycle when buffer $K+1$ is empty.

In TRAC-synthesized implementations, to allow simultaneous enqueue and dequeue actions, a single-register FIFO is considered empty both when it is actually empty or when it is enabled for dequeuing at the next clock edge. Simultaneous enqueue and dequeue actions are permitted as a sequential composition where the dequeue action is considered to have taken place before the enqueue action. In hardware terms, this creates a combinational multistage feedback path for FIFO flow control that propagates from the last stallable pipeline stage to the first pipeline stage. The cascaded feedback scheme shown in Fig. 9 allows stage K to advance both when pipeline buffer $K+1$ is actually empty and when buffer $K+1$ is going to be dequeued at the coming clock edge. This scheme allows the entire pipeline to advance synchronously on each clock edge. A stall in an intermediate pipeline stage causes all up-stream stages to stall at once. A caveat to this scheme is that the multistage feedback path could become the critical path, especially in a deeply pipelined design. In this case, one may want to break the feedback path at select stages by inserting two-deep FIFOs with local flow control. A cyclic feedback path can also be broken by inserting a two-deep FIFO with local flow control.

B. Synthesis Results

The synthesis results are presented for a TRSPEC description that implements the MIPS R2000 integer ISA with the exception of multiple/divide, partial-word or nonaligned load/stores, coprocessor interfaces, privileged instructions and exception modes. Memory load instructions and branch/jump instructions also deviate from the ISA by not obeying the required *delayed-execution* semantics. (The complete TRSPEC description of the five-stage pipelined processor model is given in [9].)

The processor description is compiled by TRAC into a synthesizable Verilog RTL description, which is subsequently compiled by the {Synopsys Design Compiler} to target both Synopsys CBA and LSI Logic 10K Series technology libraries. Table I summarizes the prelayout area and speed estimates reported by Synopsys. The row labeled "TRSPEC" characterizes the implementation synthesized from the TRSPEC description. The row labeled "Hand-coded RTL" characterizes the implementation synthesized from a hand-coded Verilog description of the same five-stage pipelined microarchitecture. The results indicate that the TRSPEC description produces an implementation that is competitive in size and speed to the implementation resulting from the hand-coded Verilog description. This similarity should not be surprising because, after all, both descriptions are describing the same microarchitecture, albeit following very different design abstractions and methodologies. The same conclusion has also been reached on comparisons of other designs and when we targeted the designs for implementation on FPGAs.

TABLE I
SUMMARY OF MIPS CORE SYNTHESIS RESULTS

	CBA tc6a		LSI 10K	
version	area (cell)	speed (MHz)	area (cell)	speed (MHz)
TRSPEC	9059	96.6	34674	41.9
Hand-coded RTL	7168	96.0	26543	42.1

The TRSPEC and the hand-coded Verilog descriptions are similar in length (790 versus 930 lines of source code), but the TRSPEC description was developed in less than one day (eight hours), whereas the hand-coded Verilog description required nearly five days to complete. The TRSPEC description can be translated in a literal fashion from an ISA manual. Whereas, the hand-coded Verilog description faces a greater representation gap between the ISA specification and RTL. The RTL designer also needs to manually inject implementation-level information that is not part of the ISA-level specification. In a TRSPEC design flow, the designer can rely on TRAC to correctly complete the design with suitable implementation decisions.

C. Current Synthesis Limitations

The synchronous hardware synthesis procedure presented in this paper has two important limitations. First, the procedure always maps the entire effect of an operation into a single clock cycle. Second, the procedure always maximizes hardware concurrency. In many hardware applications, there are restrictions on the amount and the type of resources available. Under those assumptions, it may not be optimal or even realistic to execute an operation in one clock cycle. In future work, we are developing another synthesis approach, where the effect of an operation can be executed over multiple clock cycles, while also being overlapped with the execution of other multiple-cycle operations. The key issue in this future work also hinges on how to ensure the resulting implementation correctly maintains the atomic and sequential execution semantics of the operation-centric hardware abstraction.

VII. RELATED WORK

In comparison to operation-centric abstractions, traditional state-centric abstractions more closely reflect the true nature

(e.g., explicit concurrency and synchronization) of the underlying hardware implementation. Hence, they are relatively simpler to synthesize into hardware implementations, and they afford designers greater control over the details of the synthesized outcome. On the other hand, the additional details exposed by state-centric abstractions also place a greater design burden on the designers. In a synchronous state-centric framework, a designer must explicitly manage the exact cycle-by-cycle interactions between multiple concurrent state machines. Design mistakes are common in coordinating interactions between two state machines because one cannot directly couple related transitions in different state machines. It is also difficult to design or modify one state machine in a system without considering its, often implicit, interactions with the rest of the system. The atomic and sequential execution semantics of the operation-centric abstraction removes these low-level design issues from the designer; instead the abstraction allows these low-level decisions to be offloaded to an optimizing compiler.

The operation-centric ATS formalism in this paper is similar in semantics to guarded commands [7], synchronized transitions with *asynchronous combinators* [13] and the UNITY language [5]. ATS extends these earlier models with support for hardware state primitives, such as arrays and FIFOs. In particular, the use of FIFOs simplifies the description of pipelined designs (as explained in Section VI-A). ATS serves as an intermediate representation for the compilation of source-level descriptions in the TRSPEC language [9]. TRSPEC is an adaptation of TRS [1] for describing a finite-state transition system. In addition to TRSPEC, other language syntax can also be layered on top of the basic ATS abstraction. For example, the *Bluespec* language is an operation-centric hardware description language that has a Haskell-like syntax [15]. In addition to a functional language syntax, Bluespec also leverages sophisticated functional language features in the declaration of state, operations and module interfaces.

Staunstrup and Greenstreet describe in [17] the synthesis of both synchronous and asynchronous (delay insensitive) circuits from a synchronized-transitions program. In their synchronous synthesis, the entire effect of a transition is mapped into a single clock cycle, and all enabled transitions are allowed to execute in the same clock cycle. To guarantee the atomic semantics of synchronized transitions is not violated, they enforce the CREW (concurrent-read-exclusive-write) requirement on a synthesizable set of synchronized transitions. Under the CREW requirement, a pair of transitions must be mutually exclusive in their predicate condition if they have data (read-after-write) or output (write-after-write) dependence between them. If initially two transitions violate CREW, one of their predicate conditions must be augmented by the negation of the other so the two transitions become mutually exclusive. In this way, the final synthesized system exhibits a deterministic behavior that is acceptable according to the original nondeterministic system. Dhaussy *et al.* describe a similar approach in their synthesis of UCA, a language derived from UNITY, where a program with conflicts is transformed into a program without conflicts prior to synthesis [6].

The Staunstrup and Greenstreet's CREW relationship is similar to our conflict-free relationship, but it is more restrictive than both the conflict-free and the sequential composability requirements given in Theorems 1 and 2. In particular, the sequential composability optimization enables significantly more parallelism to be exploited in an implementation than CREW. However, in the case when only the conflict-free optimization is enabled in the TRAC compiler, TRAC's conservative static test for conflict-free relationships (discussed in Section IV-B) yields essentially the same scheduled behavior as CREW. Nevertheless, our approach further differs from Staunstrup and Greenstreet in how the statically determined conflict resolutions are realized in hardware. Resolving conflicting transitions by statically transforming the predicate conditions is less efficient than the partitioned arbitrator approach in this paper because augmenting predicate conditions can lead to a lengthy final predicate expression if a transition conflicts with many transitions. The complexity of the predicate expressions can directly impact an implementation's size and critical path delay.

There is also some tangential relationship between the ATS formalism and synchronous languages [2], [3], exemplified by Esterel [4], Lustre [8], and Signal [11]. In both models, state is updated atomically at a discrete times step but the abstract concept of time is different in the two formalisms. Synchronous languages are based on a calculus of time and require explicit expression of concurrency with deterministic behavior. On the other hand, an operation-centric abstraction employs a sequential execution semantics with nondeterminism. Operation-centric hardware synthesis must automatically discover and exploit the implicit parallelism that exist in a sequentially conceived and interpreted description. Hence, as presented in Sections IV and V, the key to synthesizing an efficient implementation from an operation-centric description lies precisely in how to take advantage of the nondeterminism to select an appropriate subset of enabled transitions for concurrent execution in each cycle. Furthermore, synchronous languages have been used primarily for describing the control part of a design and thus, it is difficult to compare the two models using a processor-like example which employs rich datapaths.

VIII. Conclusion

Ultimately, the goal of a high-level description is to provide a design representation that is easy for a designer to comprehend and reason about. Although a concise notation is helpful, the utility of a *high-level* description framework has to come from the elimination of some *lower-level* details. It is in this sense that an operation-centric design framework can offer an advantage over traditional RTL design frameworks. In RTL languages, hardware concurrency must be expressed and managed explicitly. Whereas in an operation-centric language, parallelism and concurrency are implicit at the source-level, where they are later discovered and managed by an optimizing compiler.

This paper develops the theories and algorithms necessary to synthesize an efficient synchronous hardware implementation from an operation-centric description. To facilitate analysis and synthesis, this paper defines the ATS formalism, which is an operation-centric intermediate representation when mapping TRSPEC and other operation-centric source-level languages to

hardware implementations. This paper explains how to implement an ATS as a synchronous finite-state machine. The crux of the synthesis problem lies in finding a valid composition of the ATS transitions in a coherent finite-state machine that carries out as many ATS transitions concurrently as possible. This paper first presents a straightforward reference implementation and then offers two optimizations based on the parallelization of conflict-free and sequentially-composable transitions. The synthesis results from a pipelined processor design example show that an operation-centric framework allows a significant reduction in design time and effort while achieving comparable implementation quality as traditional register-transfer level design flows.

REFERENCES

[1] F. Baader and T. Nipkow, *Term Rewriting and All That*. Cambridge, U.K.: Cambridge Univ. Press, 1998.

[2] A. Benveniste and G. Berry, "The synchronous approach to reactive and real-time systems," *Proc. IEEE*, vol. 79, pp. 1270–1282, Sept. 1991.

[3] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone, "The synchronous languages twelve years later," *Proc. IEEE*, vol. 91, pp. 64–83, Jan. 2003.

[4] G. Berry, "The Foundations of Estevel," in *Proof, Language and Interaction: Essays in Honor of Robin Milner*. Cambridge, MA: MIT Press, 2000, pp. 425–454.

[5] K. M. Chandy and J. Misra, *Parallel Program Design*. Reading, MA: Addison-Wesley, 1988.

[6] P. Dhaussy, J.-M. Filloque, and B. Pottier, "Global control synthesis for an MIMD/FPGA machine," in *Proc. IEEE Workshop FPGAs Custom Comput. Mach.*, 1994, pp. 72–81.

[7] E. W. Dijkstra, "Guarded commands, nondeterminacy, and formal derivation of programs," *Commun. ACM*, vol. 18, no. 8, pp. 453–487, 1975.

[8] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data flow programming language LUSTRE," in *Proc. IEEE*, vol. 79, Sept. 1991, pp. 1305–1320.

[9] J. C. Hoe, "Operation-Centric Hardware Description and Synthesis," Ph.D. thesis, Dept. Elect. Eng. Comput. Sci., Massachusetts Inst. Technol., Cambridge, June 2000.

[10] G. Kane, *MIPS R2000 RISC Architecture*. Englewood Cliffs, NJ: Prentice-Hall, 1987.

[11] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire, "Programming real-time applications with SIGNAL," *Proc. IEEE*, vol. 79, pp. 1321–1336, Sept. 1991.

[12] T. W. S. Lee, C. J. Seger, and M. R. Greenstreet, "Automatic verification of asynchronous circuits," *IEEE Design Test Comput.*, vol. 12, pp. 24–31, Spring 1995.

[13] A. P. Ravn and J. Staunstrup, "Synchronized Transitions," Dept. Comput. Sci., Univ. Aarhus, Aarhus, Denmark, Tech. Rep. AAR-219, 1987.

[14] V. M. Rodrigues and F. R. Wagner, "Synchronous transitions and their temporal logic," in *Proc. Workshop Métodos Formais*, 1998, pp. 84–89.

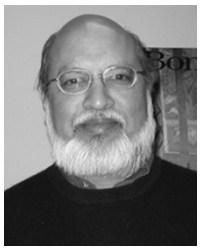
[15] A brief description of Bluespec, Sandburst Corporation. Available: http://www.bluespec.org/description.html [Online]

[16] J. Staunstrup and M. R. Greenstreet, "From high-level descriptions to VLSI circuits," *BIT*, vol. 28, no. 3, pp. 620–638, 1988.

[17] ——, "Synchronized Transitions," in *Formal Methods for VLSI Design*. Amsterdam, The Netherlands: Elsevier, 1990, pp. 71–128.

James C. Hoe (S'91–M'00) received the B.S. degree in electrical engineering and computer science from the University of California, Berkeley, in 1992 and the M.S. and Ph.D. degrees in electrical engineering and computer science from the Massachusetts Institute of Technology, Cambridge, in 1994 and 2000, respectively.

Since 2000, he has been an Assistant Professor of Electrical and Computer Engineering at Carnegie Mellon University, Pittsburgh, PA. His research interest includes many aspects of computer architecture and digital hardware design. His present focus is on developing high-level hardware description and synthesis technologies to simplify hardware development. He is also working on innovative processor microarchitectures to address issues in security and reliability.



Arvind (SM'85–F'95) received the B.Tech. degree in electrical engineering from the Indian Institute of Technology, Kanpur, in 1969 and the M.S. and Ph.D. degrees from the University of Minnesota, Twin Cities, in 1972 and 1973, respectively.

He is the Johnson Professor of Computer Science and Engineering at the Massachusetts Institute of Technology, Cambridge, where he has taught since 1979. He has contributed to the development of dynamic dataflow architectures, and together with Dr. R. S. Nikhil published the book *Implicit Parallel Programming in pH* (San Francisco, CA: Morgan Kaufmann, 2001). His current research interest is in high-level specification, modeling, and the synthesis and verification of architectures and protocols.