Synthesis of Operation-Centric Hardware Descriptions

James C. Hoe
Dept. of Electrical and Computer Engineering
Carnegie Mellon University
jhoe@ece.cmu.edu

Arvind
Laboratory for Computer Science
Massachusetts Institute of Technology
arvind@lcs.mit.edu

Abstract

Most hardware description frameworks, whether schematic or textual, use cooperating finite state machines (CFSM) as the underlying abstraction. In the CFSM framework, a designer explicitly manages the concurrency by scheduling the exact cycle-by-cycle interactions between multiple concurrent state machines. Design mistakes are common in coordinating interactions between two state machines because transitions in different state machines are not semantically coupled. It is also difficult to modify one state machine without considering its interaction with the rest of the system.

This paper presents a method for hardware synthesis from an "operation centric" description, where the behavior of a system is described as a collection of "atomic" operations in the form of rules. Typically, a rule is defined by a predicate condition and an effect on the state of the system. The atomicity requirement simplifies the task of hardware description by permitting the designer to formulate each rule as if the rest of the system is static.

An implementation can execute several rules concurrently in a clock cycle, provided some sequential execution of those rules can reproduce the behavior of the concurrent execution. In fact, detecting and scheduling valid concurrent execution of rules is the central issue in hardware synthesis from operation-centric descriptions. The result of this paper shows that an operation-centric framework offers significant reduction in design time, without loss in implementation quality.

1 Introduction

1.1 Operation-Centric Hardware Descriptions

Digital hardware designs inherently embody highly concurrent behaviors. Any non-trivial design invariably consists of a collection of cooperating finite state machines (CFSM). Hence, most hardware description frameworks, whether schematic or textual, use CFSM as the underlying abstraction. In a CFSM framework, a designer explicitly manages the concurrency by scheduling the exact cycle-by-cycle interactions between multiple concurrent state machines. Design mistakes are common in coordinating interactions between two state machines because transitions in different state machines are not semantically coupled. It is also difficult to modify one state machine without considering its interaction with the rest of the system.

This paper presents a method for hardware synthesis from an "operation centric" description, where the behavior of a system is described as a collection of "atomic" operations in the form of rules. Typically, a rule is defined by a predicate condition and an effect on the state of the system. In an execution, a rule "reads" the state of the system in one step, and if enabled, the effect of the rule updates the state in the same step. If several rules are enabled at the same time, any one of the rules can be nondeterministically selected to update the state in one step, and afterwards, a new step begins with the updated state. The atomicity requirement simplifies the task of hardware description by permitting the designer to formulate each rule as if the rest of the system is static.

Describing the instruction reorder buffer (ROB)[1] of a modern out-of-order microprocessor poses a great challenge if concurrency needs to be managed explicitly. An operation-centric description captures the behavior of an ROB more perspicuously as a collection of rules for operations like *dispatch*, *complete*, *commit*, etc. [1]. For example, the *dispatch* operation is specified to take place if there exists an instruction that has all of its operands and is waiting to execute, and furthermore, the execution unit needed by the instruction is available. The effect of the *dispatch* operation is to send the instruction to the execution unit. The rule specification of the *dispatch* operation does not have to include information about how to resolve potential conflicts arising from the concurrent execution with other operations.

The sequential and atomic interpretation of a description does not prevent a legal implementation from executing several rules concurrently in a clock cycle, provided some sequential execution of those rules can reproduce the behavior of the concurrent execution. In fact, *detecting and scheduling valid concurrent execution of rules is the central issue in hardware synthesis from operation-centric descriptions.*

1.2 Comparison to Other High-level Frameworks

Behavioral descriptions typically describe hardware, or hardware/software systems, as multiple threads of computation that communicate via a message-passing or shared-memory paradigm [13, 4, 14, 17, 5]. As in CFSM frameworks, designers of behavioral descriptions still need to manage the interactions between concurrent computations explicitly. In reconfigurable computing, both sequential and parallel programming paradigms have been used to capture functionalities for hardware implementation. Transmagrifier-C [6] and HardwareC [16] are specification languages based on C syntax plus additional constructs to convey hardware-related information such as clocking. Sequential C and Fortran programs have been automatically parallelized to target an array of configurable structures [3]. Data-parallel C languages

[1] Refer to [9] for background information.

have been used to program an array of FPGA's in Splash 2 [8] and CLAy [7]. More formal representations have also been used to describe hardware for verification. Windley uses the language from the HOL theorem proving system to describe a pipelined processor [19]. Matthews et al. have developed the Hawk language to create executable specifications of processors [15].

Paper Organization: This section introduced the concept and advantages of operation-centric hardware description. The next section presents an example. Section 3 explains the synthesis of operation-centrically described hardware, while Section 4 explains the concurrent scheduling of *conflict-free* rules. Section 5 presents a comparison of designs synthesized from operation-centric descriptions vs. hand-coded RTL descriptions. Section 6 summarizes the key contributions of this paper.

2 An Operation-Centric Example

2.1 Description of a Pipelined Processor

We describe a two-stage pipelined processor where a pipeline buffer is inserted between the fetch stage and the execute stage. We use a bounded FIFO of unspecified size to model the pipeline buffer. The FIFO provides the isolation to allow the operations in the two stages to be described independently. Although the description reflects an asynchronous and elastic pipeline, our synthesis can infer a legal implementation that is fully-synchronous and has stages separated by simple registers.

Our operation-centric description framework borrows the notation of Term Rewriting Systems (TRS) [2]. A two-stage pipelined processor can be specified as a TRS whose terms have the signature Proc(pc,rf,bf,imem,dmem). The five fields of the processor term are pc the program counter, rf the register file (an array of integer values)[2], bf the pipeline buffer (a FIFO of fetched instructions), imem the instruction memory (an array of instructions), and dmem the data memory (an array of integer values).

Instruction fetching in the fetch stage can be described by the rule:

Fetch Rule:
Proc(pc,rf,bf,imem,dmem)
→ Proc(pc+1,rf,*enq*(bf,imem[pc]),imem,dmem)

The execution of the different instructions in the execute stage can be described by separate rules. First consider the *Add* instruction:

Add Exec Rule:
Proc(pc,rf,bf,imem,dmem)
where Add(rd,r1,r2)=*first*(bf)
→ Proc(pc,rf[rd:=v],*deq*(bf),imem,dmem)
where v=rf[r1]+rf[r2]

The *Fetch* rule fetches instructions from consecutive instruction memory locations and enqueues them into bf. The *Fetch* rule is not concerned with what happens if a branch is taken, or if the pipeline encounters an exception. The *Add Exec* rule, on the other hand, processes the next pending instruction in bf as long as it is an *Add* instruction. Next, consider the two possible executions of a *Bz* (branch if zero) instruction:

Bz-Taken Exec Rule:
Proc(pc, rf, bf, imem, dmem)
if (rf[rc]=0) where (Bz(rc,ra)=*first*(bf))
→ Proc(rf[ra], rf, *clear*(bf), imem, dmem)

Bz-Not-Taken Exec Rule:
Proc(pc, rf, bf, imem, dmem)
if (rf[rc]≠0) where (Bz(rc,ra)=*first*(bf))
→ Proc(pc, rf, *deq*(bf), imem, dmem)

The *Fetch* rule performs a weak form of branch speculation by always incrementing pc. Consequently, in the execute stage, if a branch is resolved to be taken, besides setting pc to the branch target, all speculatively fetched instructions in bf need to be discarded.

In this pipeline description, the *Fetch* rule and an execute rule can be ready to fire simultaneously. Even though conceptually only one rule should be fired in each step, an implementation of this processor description must carry out the effect of both rules in the same clock cycle. Without concurrent execution, the implementation does not behave like a pipeline. However, the implementation must also ensure that a concurrent execution of multiple rules produces the same result as a sequential execution. In particular, consider the concurrent firing of the *Fetch* rule and the *Bz-Taken Exec* rule. Both rules affect pc and bf. In such a case, the implementation has to guarantee that these rules fire in some sequential order. 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.

2.2 State-Transformer View

In a TRS, the state of the system is represented by a collection of values, and a rule rewrites values to values. Given a collective state value s, a TRS rule computes a new value s' such that

$s' = \mathit{if}\ \pi(s)\ \mathit{then}\ \delta(s)\ \mathit{else}\ s$

where the π function captures the firing condition and the δ function captures the effect of a rule. It is also possible to view a rule as a state-transformer in a state-based system. In this paper, we are going to concentrate on the synthesis of state-based systems with three types of state elements: registers (R), arrays (A) and FIFOs (F). The state elements are depicted in Figure 1. 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 ***get***() query and updated to *v* using the ***set***(*v*) action. The entry of an array can be referenced using the side-effect-free ***a-get***(*idx*) query and updated to *v* using the ***a-set***(*idx*,*v*) action. The oldest value in a FIFO can be referenced using the side-effect-free ***first***() query, and can be removed by the ***deq***() action. A new value *v* can be added to a FIFO using the ***enq***(*v*) action. In addition, the contents of a FIFO can be cleared using the ***clear***() action. The status of a FIFO can be queried using the side-effect-free ***notfull***() and ***notempty***()

[2] In an expression, rf[r] gives the value stored in location r of rf, and rf[r:=v] gives the new value of the array after location r has been updated by the value v.

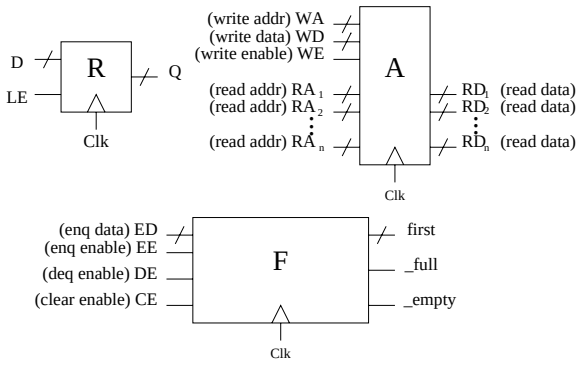

Figure 1: Synchronous state elements

queries. A rule is restricted to perform at most one action on each state element per rewrite.

In the state-transformer view, the applicability of a rule is determined by computing the π function on the current state. However, the next-state logic consists of a set of actions that alter the contents of the state elements to match $\delta(s)$. The processor rules in Section 2.1 can be restated in terms of actions:

$$
\begin{aligned}
\pi_{Fetch} &= \textbf{\textit{notfull}}(\mathsf{bf}) \\
\mathbf{a}_{Fetch,\mathsf{pc}} &= \textbf{\textit{set}}(\mathsf{pc+1}) \\
\mathbf{a}_{Fetch,\mathsf{bf}} &= \textbf{\textit{enq}}(\mathsf{imem[pc]}) \\
\pi_{Add} &= (\textbf{\textit{first}}(\mathsf{bf})=\mathsf{Add(rd,r1,r2)}) \\
&\qquad \wedge \textbf{\textit{notempty}}(\mathsf{bf}) \\
\mathbf{a}_{Add,\mathsf{rf}} &= \textbf{\textit{a-set}}(\mathsf{rd,rf[r1]+rf[r2]}) \\
\mathbf{a}_{Add,\mathsf{bf}} &= \textbf{\textit{deq}}() \\
\pi_{BzTaken} &= (\textbf{\textit{first}}(\mathsf{bf})=\mathsf{Bz(rc,ra)}) \wedge (\mathsf{rf[rc]=0}) \\
&\qquad \wedge \textbf{\textit{notempty}}(\mathsf{bf}) \\
\mathbf{a}_{BzTaken,\mathsf{pc}} &= \textbf{\textit{set}}(\mathsf{rf[ra]}) \\
\mathbf{a}_{BzTaken,\mathsf{bf}} &= \textbf{\textit{clear}}() \\
\pi_{BzNotTaken} &= (\textbf{\textit{first}}(\mathsf{bf})=\mathsf{Bz(rc,ra)}) \wedge (\mathsf{rf[rc]} \neq 0) \\
&\qquad \wedge \textbf{\textit{notempty}}(\mathsf{bf}) \\
\mathbf{a}_{BzNotTaken,\mathsf{bf}} &= \textbf{\textit{deq}}()
\end{aligned}
$$

Null actions, represented as ε, on a state element are omitted from the action list above. The complete list of actions implied by the *Add Execute* rule is $\alpha_{Add}=\langle \mathbf{a}_{\mathsf{pc}},\mathbf{a}_{\mathsf{rf}},\mathbf{a}_{\mathsf{bf}},\mathbf{a}_{\mathsf{imem}},\mathbf{a}_{\mathsf{dmem}} \rangle$ where $\mathbf{a}_{\mathsf{pc}}$, $\mathbf{a}_{\mathsf{imem}}$ and $\mathbf{a}_{\mathsf{dmem}}$ are ε's.

3 Hardware State Machine Synthesis

Implementing an operation-centric TRS description as a finite-state machine (FSM) involves combining the actions of all rules to form the FSM's next-state logic. The actions of a rule need to be qualified by the rule's π signal. For performance reasons, an implementation should carry out multiple rules concurrently while still maintaining a behavior that is consistent with a sequential execution of the atomic operations that the rules represent. We will describe such a concurrent scheduler in the next section.

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

Figure 2: ATS summary

3.1 Abstract Transition Systems (ATS)

ATS is the formalization of a state-based intermediate representation of operation-centric descriptions. An ATS is defined a triple $\langle S, S^o, X \rangle$ where S is a list of state elements, S^o is a list of initial values for the elements in S, and X is a list of operation-centric transitions where each transition is represented by a pair, $\langle \pi, \alpha \rangle$. The components of an ATS is summarized in Figure 2. Besides registers, arrays and FIFOs, ATS includes register-like state elements for input and output. An input state element (I) is like a register but without the ***set***() action. A ***get***() query on an input element returns the value of an external input. An output state element (O) supports both ***set***() and ***get***(), and its content is visible from outside of the ATS.

3.2 Reference Implementation of an ATS

One straightforward implementation of an ATS is a FSM that executes one transition per clock cycle. The elements of S are the state of the FSM. The transitions in X are combined to form the next-state logic of the FSM in three steps.

Step 1: All value expressions in the ATS are mapped to combinational signals on the current state of the state elements. In particular, this step creates a set of signals, $\pi_{T_1},...,\pi_{T_M}$, that are the π signals of transitions $T_1,...,T_M$ of an M-transition ATS. The logic mapping in this step assumes all required combinational resources are available. RTL optimizations can be employed to simplify the combinational logic and to share duplicated logic.

Step 2: In the second step, a scheduler is created to generate the set of arbitrated enable signals, $\phi_{T_1},...,\phi_{T_M}$, based on $\pi_{T_1},...,\pi_{T_M}$. (The block diagram of a scheduler is shown in Figure 3.) Any valid scheduler must, at least, ensure that for any s,

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

The reference implementation scheduler asserts only one ϕ signal in each clock cycle, reflecting the selection of one applicable transition. A priority encoder is a valid scheduler for the reference implementation.

Step 3: In the final step, one conceptually creates M independent versions of the next-state logic, each corresponding to one of the M transitions in the ATS. Next, the M sets of next-state logic are merged, state-element by state-element, using the ϕ signals for arbitration. For example, a register may have N transitions that could 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 enabled in a clock cycle. Thus, the register's latch enable is the logical-*OR* of the ϕ signals of the N relevant transitions. The new value of the register is selected from the N candidate next-state values via a multiplexer controlled by the ϕ signals. Figure 4 illustrates the merge circuit for a register that can be affected by the ***set*** actions from two transitions. The scheme assumes at most one transition's action needs to be applied to a particular element in a clock cycle. Furthermore, all the actions of a selected transition should be enabled in the same clock cycle to achieve the appearance of an atomic transition.

The merge circuit for the three state element types are given next as RTL equations. For each R, the set of transitions that update R is $\{T_{x_i} \mid \mathbf{a}^R_{T_{x_i}} = \textbf{\textit{set}}(exp_{x_i})\}$ where $\mathbf{a}^R_{T_{x_i}}$ is the action by T_{x_i} on R. R's data (D) and latch enable (LE) inputs are

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

For each A, the set of transitions that write A is $\{T_{x_i} \mid \mathbf{a}^A_{T_{x_i}} = \textbf{\textit{a-set}}(idx_{x_i}, data_{x_i})\}$. A's write address (WA), data (WD) and enable (WE) inputs are

$$\mathsf{WA} = \phi_{T_{x_1}} \cdot idx_{x_1} + ... + \phi_{T_{x_n}} \cdot idx_{x_n}$$
$$\mathsf{WD} = \phi_{T_{x_1}} \cdot data_{x_1} + ... + \phi_{T_{x_n}} \cdot data_{x_n}$$
$$\mathsf{WE} = \phi_{T_{x_1}} \vee ... \vee \phi_{T_{x_n}}$$

The set of transitions that enqueues a new value into F is $\{T_{x_i} \mid (\mathbf{a}^F_{T_{x_i}} = \textbf{\textit{enq}}(exp_{x_i})) \vee (\mathbf{a}^F_{T_{x_i}} = \textbf{\textit{en-deq}}(exp_{x_i}))\}$.

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

The set of transitions that dequeues from F is $\{T_{x_i} \mid (\mathbf{a}^F_{T_{x_i}} = \textbf{\textit{deq}}()) \vee (\mathbf{a}^F_{T_{x_i}} = \textbf{\textit{en-deq}}(exp_{x_i}))\}$.

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

Similarly, the set of transitions that clears the contents of F is $\{T_{x_i} \mid \mathbf{a}^F_{T_{x_i}} = \textbf{\textit{clear}}()\}$.

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

3.3 Correctness of the Reference Implementation

The reference implementation is said to implement an ATS correctly if

1. The implementation's sequence of state transitions corresponds to some execution of the ATS.
2. The implementation maintains liveness.

A correct implementation is not necessarily equivalent to the source ATS. Unless the scheduler employs true randomization, the reference implementation is deterministic. In other words, the implementation can only embody one of the behaviors allowed by the ATS. Thus, the implementation can enter a livelock if the ATS depends on non-determinism to make progress. The reference implementation can use a round-robin priority encoder to ensure *weak-fairness*, that is, if a transition remains applicable for a sufficient number of consecutive cycles then it is guaranteed to be selected at least once.

Although the semantics of an ATS require an execution in sequential and atomic update steps, a hardware implementation can exploit the underlying parallelism and execute multiple transitions concurrently in one clock cycle. For a pipelined processor, it is necessary to execute transitions for different pipeline stages concurrently to achieve pipelined execution.

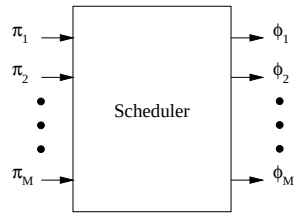

Figure 3: A monolithic scheduler for an M-transition ATS.

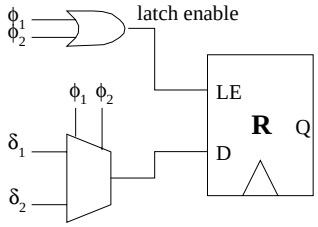

Figure 4: Circuits for combining two transitions' actions on the same state element.

4 Concurrent Scheduling of Conflict-Free Transitions

In a multiple-transitions-per-cycle implementation, the state transition in each clock cycle must correspond to a sequential execution of the ATS transitions in some order. If two transitions T_a and T_b become applicable in the same clock cycle when S is in state s, $\pi_{T_a}(\delta_{T_b}(s))$ or $\pi_{T_b}(\delta_{T_a}(s))$ must be true for an implementation to correctly select both transitions for execution. Otherwise, executing both transitions would be inconsistent with any sequential execution in two atomic update steps.

There are two approaches to execute the actions of T_a and T_b in the same clock cycle. The first approach cascades the combinational logic from the two transitions. However, arbitrary cascading does not always improve circuit performance since it may lead to 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.

This section develops a scheduling algorithm based on the *conflict-free* relationship ($<>_{CF}$). $<>_{CF}$ is a symmetrical relationship that imposes a stronger requirement than necessary for executing two transitions concurrently. However, the symmetry of $<>_{CF}$ permits a straightforward implementation that concurrently executes multiple transitions if they are pairwise $<>_{CF}$. An analysis based on the Sequential Composibility ($<_{SC}$) relationship can further increase hardware concurrency [10]. The intuition behind $<_{SC}$, an asymmetrical relationship, is that concurrent execution does not need to produce the same result as all possible sequential executions, just one.

4.1 Conflict-Free Transitions

The conflict-free relationship and the parallel composition function ***PC*** are defined in Definition 1 and Definition 2.

Definition 1 (Conflict-Free Relationship)

Two transitions T_a and T_b are said to be conflict-free ($T_a <>_{CF} T_b$) 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 $\boldsymbol{PC}(\alpha_{T_a}, \alpha_{T_b})$. □

The function ***PC*** computes a new α by composing two α's that do not contain conflicting actions on the same state element.

Definition 2 (Parallel Composition)

$$\boldsymbol{PC}(\alpha_a,\alpha_b) = \langle \boldsymbol{pc}_R(\mathbf{a}^{R_1},\mathbf{b}^{R_1}),..., \boldsymbol{pc}_A(\mathbf{a}^{A_1},\mathbf{b}^{A_1}),..., \boldsymbol{pc}_F(\mathbf{a}^{F_1},\mathbf{b}^{F_1}),...\rangle$$

where $\alpha_a = \langle \mathbf{a}^{R_1},...\mathbf{a}^{A_1},...\mathbf{a}^{F_1},...\rangle$, and $\alpha_b = \langle \mathbf{b}^{R_1},...\mathbf{b}^{A_1},...\mathbf{b}^{F_1},...\rangle$

$\boldsymbol{pc}_R(\mathbf{a}, \mathbf{b}) = $ *case* $\mathbf{a}, \mathbf{b}$ *of* $\mathbf{a}, \varepsilon \Rightarrow \mathbf{a}$; $\varepsilon, \mathbf{b} \Rightarrow \mathbf{b}$; $... \Rightarrow$ *undefined*

$\boldsymbol{pc}_A(\mathbf{a}, \mathbf{b}) = $ *case* $\mathbf{a}, \mathbf{b}$ *of* $\mathbf{a}, \varepsilon \Rightarrow \mathbf{a}$; $\varepsilon, \mathbf{b} \Rightarrow \mathbf{b}$; $... \Rightarrow$ *undefined*

$\boldsymbol{pc}_F(\mathbf{a}, \mathbf{b}) = $ *case* $\mathbf{a}, \mathbf{b}$ *of* $\mathbf{a}, \varepsilon \Rightarrow \mathbf{a}$; $\varepsilon, \mathbf{b} \Rightarrow \mathbf{b}$; $\boldsymbol{enq}(exp), \boldsymbol{deq}() \Rightarrow \boldsymbol{en\text{-}deq}(exp)$; $\boldsymbol{deq}(), \boldsymbol{enq}(exp) \Rightarrow \boldsymbol{en\text{-}deq}(exp)$; $... \Rightarrow$ *undefined*

□

Suppose T_a and T_b become applicable in the same state s. $T_a <>_{CF} T_b$ implies that the two transitions can be applied in either order in two successive steps to produce the same final state s'. $T_a <>_{CF} T_b$ further implies 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. Theorem 1 extends this result to multiple pairwise $<>_{CF}$ transitions.

Theorem 1 (Composition of $<>_{CF}$ Transitions)

Given a collection of n transitions applicable in state s, if all n transitions are pairwise conflict-free, then the following holds for any ordering $T_{x_1},...,T_{x_n}$:

$$\pi_{T_{x_2}}(\delta_{T_{x_1}}(s)) \wedge ... \wedge \pi_{T_{x_n}}(\delta_{T_{x_{n-1}}}(...\delta_{T_{x_3}}(\delta_{T_{x_2}}(\delta_{T_{x_1}}(s)))...)) \wedge (\delta_{T_{x_n}}(\delta_{T_{x_{n-1}}}(...\delta_{T_{x_3}}(\delta_{T_{x_2}}(\delta_{T_{x_1}}(s)))...)) == \delta_{PC}(s))$$

where δ_{PC} is the functional equivalent of the parallel compositions of $\alpha_{T_{x_1}},...,\alpha_{T_{x_n}}$, in any order. A proof for Theorem 1 can be found in [10].

□

4.2 Static Deduction of $<>_{CF}$

The scheduling algorithm given in this section can work with a conservative test for $<>_{CF}$, that is, if the test fails to identify a pair of transitions as $<>_{CF}$, the algorithm might generate a less optimal, but still correct implementation.

A static determination of $<>_{CF}$ can be made by comparing the domains and ranges of the transitions. The domain of a transition is the set of state elements in S "read" by the expressions in either π or α. The domain of a transition can be further sub-classified as π-domain and α-domain depending on whether the state element is read by the π-expression or an expression in α. The range of a transition is the set of state elements in 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 $\boldsymbol{D}(T)$ and $\boldsymbol{R}(T)$, a sufficient condition that ensures two transitions are $<>_{CF}$ is given by the following theorem.

Theorem 2 (Sufficient Condition for $<>_{CF}$)

Given T_a and T_b,

$$(\boldsymbol{D}(\pi_{T_a}) \cup \boldsymbol{D}(\alpha_{T_a})) \not\cap \boldsymbol{R}(\alpha_{T_b})) \wedge (\boldsymbol{D}(\pi_{T_b}) \cup \boldsymbol{D}(\alpha_{T_b})) \not\cap \boldsymbol{R}(\alpha_{T_a})) \wedge (\boldsymbol{R}(\alpha_{T_a}) \not\cap \boldsymbol{R}(\alpha_{T_b})) \Rightarrow (T_a <>_{CF} T_b)$$

□

If the domain and range of two transitions do not overlap, then the two transitions do not have any data dependences. Since their ranges do not overlap, a valid parallel composition of α_{T_a} and α_{T_b} must exist.

Definition 3 (Mutually Exclusive Relationship)

If two transitions never become applicable on the same state, then they are said to be mutually exclusive, i.e.,

$$T_a <>_{ME} T_b \text{ if } \forall\, s.\ \neg(\pi_{T_a}(s) \wedge \pi_{T_b}(s))$$

□

Two transitions that are $<>_{ME}$ satisfy the definition of $<>_{CF}$ trivially. An exact test for $<>_{ME}$ requires determining the satisfiability of the expression $(\pi_{T_a}(s) \wedge \pi_{T_b}(s))$. Fortunately, the π expression is usually a conjunction of relational constraints on the current values of state elements. A conservative test that scans two π expressions for contradicting constraints on any one state element works well in practice.

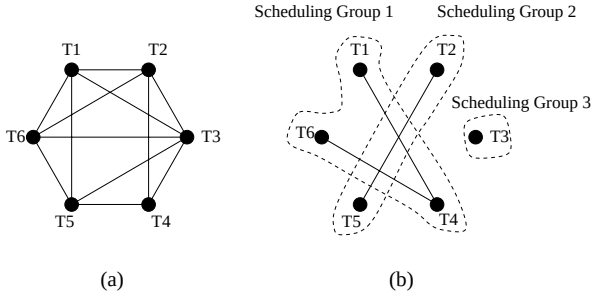

Figure 5: Scheduling Conflict-free Rules: (a) Conflict-free graph (b) Corresponding conflict graph and its connected components

4.3 Scheduling of $<>_{CF}$ Transitions

Using Theorem 1, instead of selecting a single transition per clock cycle, a scheduler can select a number of applicable transitions that are pairwise conflict-free. In other words, in each clock cycle, the ϕ signals should satisfy the condition:

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

where ϕ_T is the arbitrated transition enable signal for transition T. Given a set of applicable transitions in a clock cycle, many different subsets of pairwise conflict-free transitions could exist. Selecting the optimum subset requires evaluating the relative importance of the transitions. Alternatively, an objective metric simply optimizes the number of transitions executed in each clock cycle.

Partitioned Scheduler: In a partitioned scheduler, transitions in X are first partitioned into as many disjoint scheduling groups, P_1,...,P_k, as possible such that

$$(T_a \in P_a) \wedge (T_b \in P_b) \Rightarrow T_a <>_{CF} T_b$$

Transitions in different scheduling groups are conflict-free, and hence each scheduling group can be scheduled independently of other groups. For a given scheduling group containing T_{x_1},...,T_{x_n}, $\phi_{T_{x_1}}$,...,$\phi_{T_{x_n}}$ can be generated from $\pi_{T_{x_1}}(s)$,...,$\pi_{T_{x_n}}(s)$ using a priority encoder. In the best case, a transition T is conflict-free with every other transition in X. Hence, T is in a scheduling group by itself, and ϕ_T=π_T without arbitration.

X can be partitioned into scheduling groups by finding the connected components of an undirected graph whose nodes are transitions T_1,...,T_M and whose edges are $\{(T_i, T_j) \mid \neg(T_i <>_{CF} T_j)\}$. Each connected component is a scheduling group. For example, the undirected graph (a) in Figure 5 depicts the $<>_{CF}$ relationships in an ATS with six transitions. Graph (b) in Figure 5 gives the corresponding conflict graph where two nodes are connected if they are not $<>_{CF}$, i.e. two unconnected nodes T_i and T_j imply $T_i <>_{CF} T_j$. The conflict graph has three connected components, corresponding to the three $<>_{CF}$ scheduling groups. The ϕ signals corresponding to T_1, T_4 and T_6 can be generated using a priority encoding of their corresponding π's. Scheduling group 2 also requires a scheduler to ensure ϕ_2 and ϕ_5 are not asserted in the same clock cycle. However, ϕ_{T_3}=π_{T_3} without any arbitration.

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

Figure 6: Enumerated encoder table.

Enumerated Scheduler: Scheduling group 1 in Figure 5 contains three transitions $\{T_1, T_4, T_6\}$ such that $T_1 <>_{CF} T_6$ but neither T_1 nor T_6 is $<>_{CF}$ with T_4. Although the three transitions cannot be scheduled 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. This selection is valid because T_1 and T_6 are $<>_{CF}$ between themselves and every transition selected by the other groups. In general, the scheduler for each group can independently select multiple transitions that are pairwise $<>_{CF}$ within the scheduling group.

For a scheduling group with transitions T_{x_1},...,T_{x_n}, $\phi_{T_{x_1}}$,...,$\phi_{T_{x_n}}$ can be computed by a $2^n \times n$ lookup table indexed by $\pi_{T_{x_1}}(s)$,...,$\pi_{T_{x_n}}(s)$. The data value d_1,...,d_n at the table entry with index b_1,...,b_n can be determined by finding a clique in an undirected graph whose nodes N and edges E are defined as follows:

$$\begin{aligned} N &= \{T_{x_i} \mid b_i \text{ is asserted}\} \\ E &= \{(T_{x_i}, T_{x_j}) \mid (T_{x_i} \in N) \wedge (T_{x_j} \in N) \wedge \\ &\qquad (T_{x_i} <>_{CF} T_{x_j})\} \end{aligned}$$

For each T_{x_i} that is in the clique, assert d_i. For example, scheduling group 1 from Figure 5 can be scheduled by an enumerated encoder (Figure 6) that allows T_1 and T_6 to execute concurrently. The construction of an enumerated encoder is not necessarily unique. For example, in this example, row "011" in Figure 6 could also contain the data value "001".

4.4 Performance Gain

When X can be partitioned into scheduling groups, the partitioned scheduler is smaller and faster than the monolithic encoder used in the reference implementation. The partitioned scheduler also reduces wiring cost and delay since π's and ϕ's of unrelated transitions are not brought together for arbitration.

The property of the parallel composition function ensures that transitions are $<>_{CF}$ only if their actions on state elements do not conflict. Hence, the state update logic from the reference implementation can be used with a $<>_{CF}$ scheduler without any modification, and consequently, combinational delay of the next-state logic is not increased by this optimization. All in all, the $<>_{CF}$-scheduled implementation achieves better performance than the reference implementation by allowing more transitions to execute in a clock cycle without increasing the cycle time.

5 Synthesis Results

The synthesis procedures in the previous section have been implemented in the Term Rewriting Architectural Compiler (TRAC). TRAC accepts TRSPEC descriptions and outputs synthesizable structural descriptions in the Verilog Hardware Description Language [18]. The TRSPEC language is an adaptation of TRS for operation-centric hardware description [11]. This section discusses the synthesis of a five-stage pipelined implementation of the MIPS R2000 ISA (as described in [12]). The TRSPEC description implements all of the MIPS R2000 integer ISA except: multiple/divide; partial-word or non-aligned load/stores; coprocessor interfaces; privileged and exception modes. The delay semantics of the memory load and branch/jump instructions have also been removed. The TRSPEC description can be compiled by TRAC into a synthesizable Verilog RTL description in less than 15 seconds on a 266 MHz Pentium II processor. The TRAC-generated Verilog description can then be compiled by Synopsys Design Compiler to target both Synopsys CBA and LSI Logic 10K Series technology libraries.

5.1 Input and Output

The example from Section 2.1 described a simple processor whose instruction memory and data memory are storage arrays internal to the system. The description can be synthesized, as is, to a processor with an internal instruction ROM and an internal data RAM. However, as a realistic design for synthesis, the MIPS processor accesses external memory through input and output ports. TRSPEC allows I/O semantics to be assigned to terms as part of the type definition for a term.

5.2 Synchronous Pipeline Synthesis

As in the processor from Section 2.1, the MIPS processor is described as an asynchronous and elastic pipeline. The description of the processor does not depend on the exact depth of the pipeline FIFOs. This allows TRAC to instantiate one-deep FIFOs, i.e. a single register, as pipeline buffers. Flow control logic is added to ensure a FIFO is not overflowed or underflowed by enqueue and dequeue actions. In a naive construction, the one-deep 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 a steady-state execution. For example, if pipeline buffer K and $K+1$ are occupied and buffer $K+2$ is empty in some clock cycle, 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 clock cycle. The operation in stage K is not enabled until the next clock cycle when buffer $K+1$ has been emptied.

TRAC creates a flow control logic that includes a combinational multi-stage feedback path that propagates from the last pipeline stage to the first pipeline stage. The cascaded feedback scheme shown in Figure 7 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 cycle. A stall in an intermediate pipeline stage causes all up-stream stages to stall at once. If a pipeline stage never stalls, i.e., always dequeues, its feedback can be removed by combinational logic optimization.

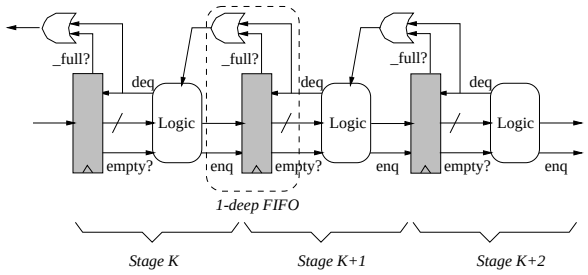

Figure 7: Synchronous pipeline with combinational multi-stage feedback flow control.

	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

Figure 8: Summary of MIPS core synthesis results

5.3 Analysis and Discussion

The table in Figure 8 summarizes the pre-layout 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 microarchitecture. The data indicates that the TRSPEC description results in an implementation that is similar in size and speed to the result of the hand-coded Verilog description. This similarity should not be surprising because, after all, both descriptions are describing the same microarchitecture, albeit using 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 [10].

The TRSPEC and the hand-coded Verilog description are similar in length (790 vs. 930 lines of source code), but the TRSPEC description is 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 has a much weaker correlation to the ISA specification. The hand-coded RTL description also requires circuit implementation information, which the RTL designer has to improvise. This does not only create more work for the RTL designer but also creates more opportunities for error. In a TRSPEC design flow, the designer can rely on TRAC to correctly supply the implementation-related information.

6 Conclusion

The operation-centric view of hardware has existed in many forms of informal hardware specification, usually to convey high-level architectural concepts. This research improves the usefulness of an operation-centric hardware description by developing a formal description framework and by enabling automatic synthesis to an efficient circuit implementation. The result of this paper shows that an operation-centric framework offers significant reduction in design time and effort without loss in implementation quality.

7 Acknowledgements

This paper describes research done at the MIT Laboratory for Computer Science. Funding for this work is provided in part by the Defense Advanced Research Projects Agency of the Department of Defense under the Ft. Huachuca contract DABT63-95-C-0150 and by the Intel Corporation. James C. Hoe is supported in part by an Intel Foundation Graduate Fellowship during this research.

References

[1] Arvind and X. Shen. Using term rewriting systems to design and verify processors. *IEEE Micro Special Issue on Modeling and Validation of Microprocessors*, May 1999.

[2] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[3] J. Babb, M. Rinard, C. A. Moritz, W. Lee, M. Frank, R. Barua, and S. Amarasinghe. Parallelizing applications into silicon. In *Proceedings of the 7th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'99)*, Napa Valley, CA, April 1999.

[4] G. Berry. The foundations of Esterel. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 1998.

[5] D. D. Gajski, J. Zhu, R. Dömer, A. Gerslauer, and S. Zhao. *SpecC Specification Language and Methodology*. Kluwer Academic Publishers, 2000.

[6] D. Galloway. The Transmogrifier C hardware description language and compiler for FPGAs. In *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines (FCCM'95)*, Napa Valley, CA, April 1995.

[7] M. Gokhale and E. Gomersall. High level compilation for fine grained FPGAs. In *Proceedings of the IEEE Symposium on FPGA-based for Custom Computing Machines (FCCM'97)*, Napa Valley, CA, April 1997.

[8] M. Gokhale and R. Minnich. FPGA computing in a data parallel C. In *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines (FCCM'93)*, Napa Valley, CA, April 1993.

[9] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2nd edition, 1996.

[10] J. C. Hoe. *Operation-Centric Hardware Description and Synthesis*. PhD Thesis, Massachusetts Institute of Technology, June 2000.

[11] J. C. Hoe and Arvind. Hardware synthesis from term rewriting systems. In *Proceedings of X IFIP International Conference on VLSI (VLSI 99)*, Lisbon, Portugal, November 1999.

[12] G. Kane. *MIPS R2000 RISC Architecture*. Prentice Hall, 1987.

[13] L. Lavagno and E. Sentovich. ECL: A specification environment for system-level design. In *Proceedings of the 36th ACM/IEEE Design Automation Conference (DAC'99)*, New Orleans, LA, June 1999.

[14] S. Liao, S. Tjinag, and R. Gupta. An efficient implementation of reactivity for modeling hardware in the Scenic design environment. In *Proceedings of the 34th ACM/IEEE Design Automation Conference (DAC'97)*, Anaheim, CA, June 1997.

[15] J. Matthews, J. Launchbury, and B. Cook. Microprocessor specification in Hawk. In *Proceedings of the 1998 International Conference on Computer Languages*, Chicago, IL, 1998.

[16] Stanford University. *HardwareC – A Language for Hardware Design*, December 1990.

[17] D. E. Thomas, J. K. Adams, and H. Schmit. A model and methodology for hardware-software codesign. *IEEE Design and Test of Computers*, September 1993.

[18] D. E. Thomas and P. R. Moorby. *The Verilog Hardware Description Language*. Kluwer Academic Publishers, 3rd edition, 1996.

[19] P. J. Windley. Verifying pipelined microprocessors. In *Proceedings of the 1995 IFIP Conference on Hardware Description Languages and their Applications (CHDL'95)*, Tokyo, Japan, 1995.