


Digital Design with Implicit State Machines

Fengyun Liu 

EPFL, Switzerland

fengyun.liu@epfl.ch

Aleksandar Prokopec 

Oracle Labs, Switzerland

aleksandar.prokopec@gmail.com

Martin Odersky

EPFL, Switzerland

martin.odersky@epfl.ch

Abstract

Claude Shannon, in his famous thesis (1938), revolutionized circuit design by showing that *Boolean algebra* subsumes all ad-hoc methods that are used in designing switching circuits, or combinational circuits as they are commonly known today. But what is the calculus for sequential circuits? Finite-state machines (FSM) are close, but not quite, as they do not support arbitrary parallel and hierarchical composition like that of Boolean expressions. We propose an abstraction called *implicit state machine* (ISM) that supports parallel and hierarchical composition. We formalize the concept and show that any system of parallel and hierarchical ISMs can be flattened into a single flat FSM without exponential blowup. As one concrete application of implicit state machines, we show that they serve as an attractive abstraction for digital design and logical synthesis.

2012 ACM Subject Classification Replace ccsdesc macro with valid one

Keywords and phrases Finite-state machines, hierarchical FSM

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

1 Introduction

Claude Shannon [26] revolutionized circuit design by showing that *Boolean algebra* subsumes all ad-hoc methods that are used in designing switching circuits, or combinational circuits as they are commonly known today. In contrast to combinational circuits which only contain stateless gates, sequential circuits may also contain stateful elements, like registers. But what is the calculus for sequential circuits? Finite-state machines (FSM) are close, but not quite.

A good abstraction in programming should be composable. In a Boolean expression $a \vee b$, the sub-expression a and b can be arbitrary Boolean expressions. We may also put two Boolean expression side by side to achieve parallel composition. Essentially, any combinational circuit design will eventually result in a Boolean expression, regardless of whether the design language is in VHDL, Verilog, or Chisel [1]. The composability of Boolean expression ensures that any combinational circuit can be represented.

If we turn to sequential circuits, which may contain state elements and cycles, what is the calculus that all sequential circuits can compile to, like Boolean algebra for combinational circuits? Finite-state machines are close to fulfill the role, but not quite. Classic FSMs support neither hierarchical composability nor parallel composition. The milestone paper by Benveniste and Berry [2] argued that the lack of support for hierarchical design and concurrency is mentioned in as a major drawback of FSMs.

Conceptually, we may compose FSMs side by side or in a nested way, which leads to *parallel and hierarchical FSMs*. In a hierarchical FSM, the behavior of the outer FSM depends on that of the inner FSM, and the inner FSM has a privileged access to the current state of the outer FSM. Parallel FSMs run side-by-side and respond to inputs concurrently.



© Fengyun Liu, Aleksandar Prokopec and Martin Odersky;
licensed under Creative Commons License CC-BY

42nd Conference on Very Important Topics (CVIT 2016).

Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:25

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

46 If one FSM can be in state a, b , the other can be in state c, d , then their parallel composition
 47 may be in states ac, ad, bc, bd .

48 There has been proposals for programming with hierarchical and parallel FSMs [7, 8, 12, 19],
 49 but so far no proposals address the two problems below:

- 50 ■ How to support parallel and hierarchical composition of FSMs in a *declarative* language?
- 51 ■ How to transform a complex system of FSMs into a flat FSM?

52 While experts in logic verification and synthesis usually work with flat FSMs for its
 53 simplicity and expressiveness, digital designers primarily work with hierarchical FSMs to
 54 decompose the complexity of a system. It is unknown how to support hierarchical and parallel
 55 composition of FSMs in a language, and then transform it into a flat FSM to facilitate formal
 56 verification such as model checking [5], and optimizations such as state encoding [10, 30].

57 The flattening of hierarchical and parallel FSMs generally results in exponential blowup
 58 in the size of their representation, e.g. flattening of 32 parallel 2-state FSMs would result in
 59 a flat FSM with 2^{32} states. Existing programming models with FSMs require one case for
 60 each state in the code [7, 8, 12, 19], consequently, the exponential blowup cannot be avoided
 61 in such languages. This creates a gap between a complex system of parallel and hierarchical
 62 FSMs and a flat FSM. Despite its simplicity and mathematical elegance, we still do not know
 63 how to make FSMs a first-class construct for programming, optimization and verification
 64 due to the lack of efficient composability and flattening.

65 To bridge the gap, we propose a novel abstraction, called *implicit state machine* (ISM),
 66 that supports arbitrary parallel and hierarchical composition of FSMs. Implicit state machines
 67 do not mandate states to be explicitly specified in the program, which avoids the exponential
 68 blowup when flattening a complex system of FSMs. This flexible composability makes
 69 implicit state machine an elegant first-class programming construct for digital design, and
 70 the avoidance of exponential blowup in flattening makes implicit state machines an attractive
 71 intermediate language for compilation, optimization and verification.

72 From the perspective of circuit design, the flattening keeps the *area* and the *delay*, the two
 73 optimization goals of logic synthesis, unchanged. The result implies that any synchronous
 74 sequential circuits is equivalent to a circuit with all state elements at the boundary, and a big
 75 combinational core at the center. We conjecture this result will lead to more optimization
 76 opportunities. For example, now combinational techniques may be used to optimize the
 77 whole circuit, while it was previously convenient to optimize only combinational fragments
 78 using the fundamental techniques. It may also give rise to novel hardware architectures. For
 79 example, FPGAs no longer need to scatter state elements (e.g. D flip-flops) in its layout.

80 Our contributions are listed below:

- 81 ■ We introduce the concept of implicit state machines, and formalize the concept in a
 82 declarative calculus. Implicit state machines support parallel and hierarchical composition,
 83 and we may optimize and reason about the code by equational reasoning.
- 84 ■ We show that any parallel and hierarchical FSMs can be flattened into a flat implicit state
 85 machine in polynomial time and code size. As far as we know, this is the first abstraction
 86 for hierarchical and parallel FSMs that avoids exponential blowup in flattening.
- 87 ■ To the best of our knowledge, we are the first to theorize that any synchronous sequential
 88 circuits is equivalent to a circuit with all state elements at the boundary and a big
 89 combinational core at the center with the same area and delay.
- 90 ■ We create an embedded DSL in Scala based on implicit state machines, and the initial
 91 experiments show positive results when implicit state machine is used as a programming
 92 model and an intermediate representation for logic synthesis.

2 Implicit State Machines

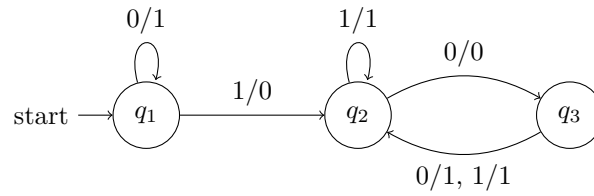
2.1 Introduction

Finite-state machines are widely used in the design and verification of reactive and real-time systems, which include critical systems that control nuclear plants, airplanes, trains, cars, etc. As a mathematical model, finite-state machines can precisely and succinctly characterize the behaviors of such systems, which forms the basis to formally verifying that the systems work reliably in accordance with the specification.

Mathematically, a finite state machine is usually represented as a quintuple (I, S, s_0, σ, O) :

- I is the set of inputs
- S is the set of states
- $s_0 \in S$ is the initial state
- $\sigma : I \times S \rightarrow S \times O$ maps the input and the current state to the next state and the output
- O is the set of outputs

FSM can also be represented graphically by *state-transition diagrams*, as the following figure shows:



In the state machine above, q_1 is the initial state, and each edge denotes a transition: the label $0/1$ on the edge means the transition happens when the input is 0, and it outputs 1 when the transition occurs.

Implicit state machines are based on a reflection on the essence of FSM: a mapping from input and state to the next state and output. The first insight towards implicit state machines is that the mapping function does not have to be represented as a set whose size correlates with the size of the state space, as it is the case in existing languages for programming with FSMs [12, 8, 7, 19]. In a declarative language, the mapping functionality can be represented by any expression. This gives us a tentative representation as follows:

$$\lambda x: I \times S. (t_1, t_2) \quad : \quad I \times S \rightarrow S \times O$$

The body (t_1, t_2) enforces that the output and next state are implemented as two functions. This imposes unnecessary constraints. If we introduce tuples in the language, we can replace (t_1, t_2) just by t :

$$\lambda x: I \times S. t \quad : \quad I \times S \rightarrow S \times O$$

The second insight is that the state is neither an input to an FSM nor an output of an FSM, but a self reference. It leads us to the following representation with the state variable s :

$$\lambda x: I. fsm \{ s \Rightarrow t \} \quad : \quad I \rightarrow O$$

In the above, the term t still has the type $S \times O$, but seen from outside, a state machine just maps input to output, which corresponds to our intuition.

23:4 Digital Design with Implicit State Machines

The last insight is that the inputs do not need to be represented explicitly, they can be *captured* from the lexical scope:

$$fsm \{ s \Rightarrow t \} \quad : \quad O$$

We still miss the initial state, so we use the value v to denote the initial state of the FSM:

$$fsm \{ v \mid s \Rightarrow t \} \quad : \quad O$$

Voila! Suppose we are working in the domain of digital circuits, a one-bit D flip-flop with an input signal d can be represented as follows:

$$fsm \{ 0 \mid s \Rightarrow (d, s) \}$$

114 It takes the value d as the next state, and outputs the last state on every clock. We may
115 compose several such flip-flops to implement a shift register for a given input d :

```
116 let q1 = fsm { 0 | s => (d, s) } in
117 let q2 = fsm { 0 | s => (q1, s) } in
118 let q3 = fsm { 0 | s => (q2, s) } in
119 let q4 = fsm { 0 | s => (q3, s) } in
120 (q1, q2, q3, q4)
```

121 An equivalent flat FSM that implements the 4-bit shift register is shown below:

```
122 fsm { (0, 0, 0, 0) | s => ((d, s.1, s.2, s.3), s) }
```

123 Implicit state machines are just expressions, thus they may appear anywhere that an ex-
124 pression is allowed. In particular, we may nest them to get another equivalent implementation
125 of the shift register:

```
126 fsm { 0 | q1 =>
127   let q2 = fsm { 0 | s => (q1, s) } in
128   let q3 = fsm { 0 | s => (q2, s) } in
129   let q4 = fsm { 0 | s => (q3, s) } in
130   (d, (q1, q2, q3, q4))
131 }
```

132 In the following, we formalize implicit state machines in a calculus.

133 2.2 Syntax

134 The syntax of the language is presented below:

t	$::=$	<i>terms</i>
	a, b, c	<i>external input</i>
	x, y, z, s	<i>variables</i>
	$\text{let } x = t \text{ in } t$	<i>let binding</i>
	β	<i>Boolean value</i>
	$t * t$	<i>1 bit and</i>
	$t + t$	<i>1 bit or</i>
135	$!t$	<i>1 bit not</i>
	(t, \dots, t)	<i>tuple</i>
	$t.i$	<i>projection</i>
	$\text{fsm } \{ v \mid s \Rightarrow t \}$	<i>implicit state machine</i>
β	$::=$	$0 \mid 1$ <i>Boolean values</i>
v	$::=$	$\beta \mid (v, \dots, v)$ <i>values</i>
i	$::=$	$0, 1, 2, \dots$ <i>indexes</i>

136 Beyond the basic elements of Boolean algebra, we also introduce *let*-bindings, which is a
 137 basic abstraction and reuse mechanism. Tuples and projections are introduced for parallel
 138 composition and decomposition. In a projection $t.i$, the index i must be a statically known
 139 number. For implicit state machines, we require that the initial state is a value.

140 A circuit usually has external inputs, which is represented by variables a, b, c . By
 141 convention, we use x, y, z for *let*-bindings, and s for the binding in implicit state machines.

142 We choose Boolean algebra as the domain theory, but it can also be other mathematical
 143 structures, like groups or abelian groups. Our transform does not assume properties of
 144 mathematical structures as long as we may *substitute equals for equals* [29].

145 2.3 Semantics

The semantics of the language is defined with the help of a state map σ and an environment ρ . The state σ maps a state variable to a state value, the environment variable ρ maps an external signal to a value. The big-step operational semantics is defined with the following reduction relation:

$$t \xrightarrow{\sigma, \rho} v \mid \sigma'$$

146 It means that given the current state σ and environment ρ , the term t evaluates to the
 147 value v with the next state σ' . The semantics follows the *synchronous hypothesis* [2], which
 148 assumes that the computation of the response to an input takes no time. For synchronous
 149 digital circuits, it means that the system produces an output at each clock tick. The reduction
 150 rules are defined in Figure 1. We explain the rules below:

- 151 ■ E-VALUE. If it is already a value, do nothing. There are no nested state machines, thus
 152 the mapping for the next state is the empty set.
- 153 ■ E-INPUT. Look up the external variable a from the environment ρ .
- 154 ■ E-LET. First evaluate t_1 to the value v_1 , then evaluate t_2 with x replaced by v_1 .
- 155 ■ E-TUPLE. Evaluate each component in parallel to a value, and accumulate the mapping
 156 for the next state.
- 157 ■ E-PROJECT. First evaluate the term to a tuple value, then return the corresponding
 158 component.
- 159 ■ E-AND. Evaluate the two components in parallel to Boolean values, then call the helper
 160 method *and* to compute the resulting Boolean value β . As each component may contain
 161 implicit state machines, accumulate the mapping for the next state.

$v \xrightarrow{\sigma, \rho} v \mid \emptyset$	(E-VALUE)
$\frac{v = \rho(a)}{a \xrightarrow{\sigma, \rho} v \mid \emptyset}$	(E-INPUT)
$\frac{t_1 \xrightarrow{\sigma, \rho} v_1 \mid \sigma' \quad [x \mapsto v_1] t_2 \xrightarrow{\sigma, \rho} v_2 \mid \sigma''}{let\ x = t_1\ in\ t_2 \xrightarrow{\sigma, \rho} v \mid \sigma' \cup \sigma''}$	(E-LET)
$\frac{t_1 \xrightarrow{\sigma, \rho} v_1 \mid \sigma_1 \quad \dots \quad t_n \xrightarrow{\sigma, \rho} v_n \mid \sigma_n}{(t_1, \dots, t_n) \xrightarrow{\sigma, \rho} (v_1, \dots, v_n) \mid \sigma_1 \cup \dots \cup \sigma_n}$	(E-TUPLE)
$\frac{t \xrightarrow{\sigma, \rho} (v_1, \dots, v_i, \dots, v_n) \mid \sigma'}{t.i \xrightarrow{\sigma, \rho} v_i \mid \sigma'}$	(E-PROJECT)
$\frac{t_1 \xrightarrow{\sigma, \rho} \beta_1 \mid \sigma' \quad t_2 \xrightarrow{\sigma, \rho} \beta_2 \mid \sigma'' \quad \beta = and(\beta_1, \beta_2)}{t_1 * t_2 \xrightarrow{\sigma, \rho} \beta \mid \sigma' \cup \sigma''}$	(E-AND)
$\frac{t_1 \xrightarrow{\sigma, \rho} \beta_1 \mid \sigma' \quad t_2 \xrightarrow{\sigma, \rho} \beta_2 \mid \sigma'' \quad \beta = or(\beta_1, \beta_2)}{t_1 + t_2 \xrightarrow{\sigma, \rho} \beta \mid \sigma' \cup \sigma''}$	(E-OR)
$\frac{t \xrightarrow{\sigma, \rho} \beta \mid \sigma' \quad \beta' = not(\beta)}{!t \xrightarrow{\sigma, \rho} \beta' \mid \sigma'}$	(E-NOT)
$\frac{v = \sigma(s) \quad [s \mapsto v] t \xrightarrow{\sigma, \rho} (v_1, v_2) \mid \sigma'}{fsm\ \{ v \mid s \Rightarrow t \} \xrightarrow{\sigma, \rho} v_2 \mid \{ s \mapsto v_1 \} \cup \sigma'}$	(E-FSM)

■ **Figure 1** Big-step operational semantics

- 162 ■ E-OR. Similar as above, but use the helper function *or* to compute the resulting value.
- 163 ■ E-NOT. Similar as above, but use the helper function *not* to compute the resulting value.
- 164 ■ E-FSM. First look up the value for the current state from the state map σ . Then evaluate
- 165 the body of the state machine to a pair value (v_1, v_2) . The output is v_2 , and the next
- 166 state is v_1 .

167 The reduction relation only defines one-step semantics. The semantics of a system is

168 defined by the *trace* of a given input series ρ_0, ρ_1, \dots . We define it formally below:

169 ► **Definition 1** (Trace). *The trace of a system t with respect to an input sequence ρ_0, ρ_1, \dots*

170 *is the sequence o_0, o_1, \dots such that*

- 171 ■ $t \xrightarrow{\sigma_0, \rho_0} o_0 \mid \sigma_1$
- 172 ■ \dots
- 173 ■ $t \xrightarrow{\sigma_i, \rho_i} o_i \mid \sigma_{i+1}$
- 174 ■ \dots

175 *In the above, σ_0 is the initial state of FSMs as specified in t .*

2.4 Type System

We introduce a simple type system to ensure that the system is sound, i.e. it never gets stuck. The type system is presented in Figure 2. In the system, there are only two types: *Bool* for Boolean values and (T_1, \dots, T_n) for tuples. We explain the typing rules below:

$T ::= Bool \mid (T, \dots, T)$		
$\Gamma \vdash \beta : Bool$	(T-BOOL)	$\frac{\Gamma \vdash t : (T_1, \dots, T_i, \dots, T_n)}{\Gamma \vdash t.i : T_i}$ (T-PROJECT)
$\frac{a : T \in \Gamma}{\Gamma \vdash a : T}$	(T-INPUT)	$\frac{\Gamma \vdash t_1 : Bool \quad \Gamma \vdash t_2 : Bool}{\Gamma \vdash t_1 * t_2 : Bool}$ (T-AND)
$\frac{x:T \in \Gamma}{\Gamma \vdash x : T}$	(T-VAR)	$\frac{\Gamma \vdash t_1 : Bool \quad \Gamma \vdash t_2 : Bool}{\Gamma \vdash t_1 + t_2 : Bool}$ (T-OR)
$\frac{\Gamma \vdash t : Bool}{\Gamma \vdash !t : Bool}$	(T-NOT)	$\frac{\Gamma \vdash t_1 : T_1 \quad \Gamma, x:T_1 \vdash t_2 : T_2}{\Gamma \vdash let\ x = t_1\ in\ t_2 : T_2}$ (T-LET)
$\frac{\Gamma \vdash t_1 : T_1 \quad \dots \quad \Gamma \vdash t_n : T_n}{\Gamma \vdash (t_1, \dots, t_n) : (T_1, \dots, T_n)}$	(T-TUPLE)	$\frac{\Gamma \vdash v : T_1 \quad \Gamma, s:T_1 \vdash t : (T_1, T_2)}{\Gamma \vdash fsm\ \{ v \mid s \Rightarrow t \} : T_2}$ (T-FSM)

■ **Figure 2** Type System

- 180 ■ T-BOOL. The type for Boolean values is always *Bool*.
- 181 ■ T-INPUT. For inputs, their types are predefined in the environment.
- 182 ■ T-VAR. For variables, their types also appear in the environment.
- 183 ■ T-NOT. The term t must be *Bool*.
- 184 ■ T-TUPLE. If each component has a type, and then the type of the tuple has a corresponding tuple type.
- 185 ■ T-PROJECT. If the term t has a tuple type, then the projection has the type of the corresponding component.
- 186 ■ T-AND. If each component has the type *Bool*, the result also has the type *Bool*.
- 187 ■ T-OR. The same as above.
- 188 ■ T-LET. If the bound term has the type of T_1 , and the body of the let-binding has the type T_2 under the environment Γ extended with the binding $x:T_1$, then the let-binding has the type T_2 . Note that this rule forbids the usage of x_1 in t_1 , which prevents undesired circles.
- 189 ■ T-FSM. If the initial value has the type T_1 , and the body has the type (T_1, T_2) under the environment Γ extended with the binding $s:T_1$, then the FSM has the type T_2 .
- 190
- 191
- 192
- 193
- 194
- 195
- 196 We need an auxiliary definition of *value map typing*:

$$\frac{\Gamma \vdash \emptyset \quad \Gamma \vdash \xi \quad \Gamma \vdash v : T}{\Gamma, \alpha : T \vdash \xi \cup \{ \alpha \mapsto v \}}$$

In the above, α ranges over inputs a and state variables s , and ξ ranges over input map ρ and state map σ .

► **Theorem 2 (Soundness).** *If $\Gamma \vdash t : T$, and if for each ρ_i in the input sequence ρ_0, ρ_1, \dots we have $\Gamma \vdash \rho_i$, then there exists a trace corresponding to the input sequence.*

The proof follows from the following lemma by induction on the length of the input sequence:

► **Lemma 3.** *Given $\Gamma \vdash t : T$, $\Gamma \vdash \rho$, $\Gamma \vdash \sigma$, $\Gamma \vdash \sigma_0$, where σ_0 is the initial state map as specified in t , then $t \xrightarrow{\sigma, \rho} v \mid \sigma'$, $\Gamma \vdash v : T$ and $\Gamma \vdash \sigma'$.*

Sketch. By induction on the typing judgment $\Gamma \vdash t : T$. ◀

2.5 Flattening

In this section, we present a transform that translates any system of parallel and hierarchical implicit state machines into a flat implicit state machine. The transformation is defined in Figure 3. It consists of two major steps:

- **Lifting.** This step lifts FSMs to top-level.
- **Flattening.** This step merges FSMs to a single FSM.

For the purposes of the transformation, we first define the FSM-free fragment of the language, which is represented by e . Lifting will result in *lifted normal form* (N), where all FSMs are at the nested at the top of the program, with an FSM-free fragment in the middle.

The relation $t_1 \rightsquigarrow_L t_2$ says that the term t_1 takes a lifting step to t_2 . Lifting is defined with the help of the lifting context L . The lifting context specifies that the transform follows the order left-right and top-down. The actual lifting happens with the function $\llbracket \cdot \rrbracket$, which transforms the source program to the expected form. We explain the concrete transform rules below:

- $fsm \{ v \mid s \Rightarrow e_1 \} * t_2$. The FSM absorbs t_2 into its body. The symmetric case, and the cases for AND and OR are similar.
- $let x = fsm \{ v \mid s \Rightarrow e_1 \} in t_2$. It pulls the let-binding into the body. The case in which FSM is in the body of let-binding is similar.
- $fsm \{ v \mid s \Rightarrow e \}.i$. It pulls the projection into the body of FSM.
- $(\bar{e}, fsm \{ v \mid s \Rightarrow e \}, \bar{t})$. It pulls the tuple into the body of FSM.

Once all FSMs are nested at the top-level after lifting, flattening takes place. The relation $t_1 \rightsquigarrow_F t_2$ says that the term t_1 takes a flattening step to t_2 . Flattening is defined with the help of the flattening context F . The flattening context specifies that the flattening happens from inside towards outside. The actual merging step is quite straightforward: it just combines the initial states v_1 and v_2 , as well as merges s_1 and s_2 into s .

We use the notation $t_1 \rightsquigarrow t_2$ to mean that t_1 takes either a lifting step (\rightsquigarrow_L) or a flattening step (\rightsquigarrow_F) to t_2 . We write $t_1 \rightsquigarrow^* t_2$ to mean 0 or multiple such transform steps. For simplicity of presentation, we omit the formal definitions.

FSM-free Fragment

$$e ::= v \mid e * e \mid e + e \mid !e \mid (e, \dots, e) \mid e.i \mid \text{let } x = e \text{ in } e \mid x \mid s \mid a$$

Lifted Normal Form

$$N ::= e \mid fsm \{ v \mid s \Rightarrow N \}$$

Lifting

$$L ::= [\cdot] \mid L * t \mid e * L \mid L + t \mid e + L \mid !L \mid L.i \mid (e_1, \dots, L, \dots, t_n) \mid fsm \{ v \mid s \Rightarrow L \} \mid \text{let } x = L \text{ in } t \mid \text{let } x = e \text{ in } L$$

$$\frac{\llbracket t \rrbracket = fsm \{ v \mid s \Rightarrow t' \}}{L[t] \rightsquigarrow_L L[fsm \{ v \mid s \Rightarrow t' \}]}$$

$$\begin{aligned} \llbracket fsm \{ v \mid s \Rightarrow e_1 \} * t_2 \rrbracket &= fsm \{ v \mid s \Rightarrow \text{let } x = e_1 \text{ in } (x.1, x.2 * t_2) \} \\ \llbracket e_2 * fsm \{ v \mid s \Rightarrow e_1 \} \rrbracket &= fsm \{ v \mid s \Rightarrow \text{let } x = e_1 \text{ in } (x.1, e_2 * x.2) \} \\ \llbracket fsm \{ v \mid s \Rightarrow e_1 \} + t_2 \rrbracket &= fsm \{ v \mid s \Rightarrow \text{let } x = e_1 \text{ in } (x.1, x.2 + t_2) \} \\ \llbracket e_2 + fsm \{ v \mid s \Rightarrow e_1 \} \rrbracket &= fsm \{ v \mid s \Rightarrow \text{let } x = e_1 \text{ in } (x.1, e_2 + x.2) \} \\ \llbracket ! fsm \{ v \mid s \Rightarrow e \} \rrbracket &= fsm \{ v \mid s \Rightarrow \text{let } x = e \text{ in } (x.1, !x.2) \} \\ \llbracket \text{let } x = fsm \{ v \mid s \Rightarrow e_1 \} \text{ in } t_2 \rrbracket &= fsm \{ v \mid s \Rightarrow \text{let } s_1, x = e_1 \text{ in } (s_1, t_2) \} \\ \llbracket \text{let } x = e_1 \text{ in } fsm \{ v \mid s \Rightarrow e_2 \} \rrbracket &= fsm \{ v \mid s \Rightarrow \text{let } x = e_1 \text{ in } e_2 \} \\ \llbracket fsm \{ v \mid s \Rightarrow e \}.i \rrbracket &= fsm \{ v \mid s \Rightarrow \text{let } x = e \text{ in } (x.1, x.2.i) \} \\ \llbracket (\bar{e}, fsm \{ v \mid s \Rightarrow e \}, \bar{t}) \rrbracket &= fsm \{ v \mid s \Rightarrow \text{let } x = e \text{ in } (x.1, (\bar{e}, x.2, \bar{t})) \} \end{aligned}$$

Flattening

$$F ::= [\cdot] \mid fsm \{ v \mid s \Rightarrow F \}$$

$$\frac{\llbracket N \rrbracket = fsm \{ v \mid s \Rightarrow e \}}{F[N] \rightsquigarrow_F F[fsm \{ v \mid s \Rightarrow e \}]}$$

$$\llbracket fsm \{ v_1 \mid s_1 \Rightarrow fsm \{ v_2 \mid s_2 \Rightarrow e_2 \} \} \rrbracket = fsm \{ (v_1, v_2) \mid s \Rightarrow \text{let } s_1, s_2 = s \text{ in } \text{let } x = e_2 \text{ in } ((x.2.1, x.1), x.2.2) \}$$

■ **Figure 3** Flattening of nested FSMs. We write $\text{let } x, y = t_1 \text{ in } t_2$ as a syntactic sugar for $\text{let } z = t_1 \text{ in } \text{let } x = z.1 \text{ in } \text{let } y = z.2 \text{ in } t_2$.

234 ► **Theorem 4** (Complexity). *If the term t contains FSMs, then there exists e such that*
 235 *$t \rightsquigarrow^* fsm \{ v \mid s \Rightarrow e \}$ in $O(m * n)$ steps where m is the size of the term t , and n is the*
 236 *number of state machines in the code.*

237 **Sketch.** During lifting, each step moves some code that pre-exists in t inside another FSM.
 238 Thus, the worse case is $O(m * n)$. During flattening, each step reduces one FSM, thus it
 239 takes n steps for flattening. Therefore, the complexity is $O(m * n)$. ◀

240 A tighter bound is $O(d * n)$, where d is the max depth of FSM from the root (if we see a
 241 term t as an abstract syntax tree), n is the number of FSMs. However, as lifting introduces

242 *let*-bindings which changes the height of the tree, technically it is more complex to establish
 243 the bound, we thus leave it to future work.

244 Meanwhile, the complexity also establishes the bound for the resulting code size after
 245 flattening: for each lifting and flattening step, the code size increase by a small constant
 246 (usually an additional *let*-binding and tuple), thus code size increase is also bound by $O(m*n)$.

247 ► **Corollary 5 (Code Size).** *If the term t contains FSMs, and there exists e such that*
 248 *$t \rightsquigarrow^* fsm \{ v \mid s \Rightarrow e \}$, then the code size increase of e compared to e is bounded by*
 249 *$O(m*n)$, where m is the size of the term t , and n is the number of state machines in the*
 250 *code.*

251 ► **Theorem 6 (Semantic Preserving).** *If $t \rightsquigarrow t'$, then they have the same trace for any given*
 252 *input sequence ρ_0, ρ_1, \dots .*

253 It follows from the following lemmas by induction on the length of the trace:

254 ► **Lemma 7.** *If $t \rightsquigarrow_L t'$, $t \xrightarrow{\sigma, \rho} v \mid \sigma_1$, then $t' \xrightarrow{\sigma, \rho} v \mid \sigma_1$.*

255 **Sketch.** First perform induction on the lifting contexts, then perform case analysis on the
 256 concrete transform rules. ◀

257 ► **Lemma 8.** *If $N \rightsquigarrow_F N'$, i.e. in the flattening $\llbracket fsm \{ v_1 \mid s_1 \Rightarrow fsm \{ v_2 \mid s_2 \Rightarrow e_2 \} \} \rrbracket$*
 258 *of two state machines, let $f = \lambda\sigma. \{ s \mapsto (\sigma(s_1), \sigma(s_2)) \} \cup (\sigma \setminus \{ s_1, s_2 \})$, and $f(\sigma) = \sigma'$,*
 259 *$N \xrightarrow{\sigma, \rho} v \mid \sigma_1$, then $N' \xrightarrow{\sigma', \rho} v \mid \sigma'_1$ and $f(\sigma_1) = \sigma'_1$.*

260 **Sketch.** Perform induction on the flattening contexts. Note that for the initial states σ_0 and
 261 σ'_0 specified in N and N' respectively, $f(\sigma_0) = \sigma'_0$ holds trivially.

262 ◀

263 2.6 Discussion: Are Implicit State Machines FSMs?

264 The mathematical definition of FSM requires the transition function to be a *pure function*, i.e.
 265 a function that always return the same result given the same input. However, it is generally
 266 not the case for implicit state machines, as an implicit state machine may contain a nested
 267 implicit state machine, which makes the transition function stateful or impure. Consequently,
 268 if an implicit state machine does not contain any nested ISM, then its body is a pure Boolean
 269 function, which make the ISM an FSM in the mathematical sense.

270 From this perspective, flattening plays another important role: it transforms a possibly
 271 non-FSM implicit state machine to an FSM. This also reflects a natural design choice of
 272 implicit state machines: in order to support hierarchical state machines, we need to give up
 273 the requirement that the transition function is pure.

274 Also note that implicit state machines just do not mandate states to be explicitly
 275 represented in the program, however, they do not forbid that. This means that programmers
 276 can continue to program with explicit states when necessary. This is can be done with a
 277 switch on the state of the FSM (in pseudocode):

```
278 fsm { 0 | s =>
279     when (s == 0) t1
280     when (s == 1) t2
281     when (s == 2) t3
282     otherwise      t4
283 }
```

In the above, we use the `when` construct to define one transition for each state. We implement `when` as a syntactic sugar in our DSL and use it to decode controller instructions (Section 4).

Note that outside the setting of formal verification and theory of computation, the term finite-state machine is sometimes used in programming to loosely mean any machine that has a finite set of states. In the rest of the paper, when there is no danger of misunderstanding, we use the term FSM in the loose sense.

3 Programming Model for Digital Design

The hardware design community is yearning for a better programming language [16, 20, 21]. We believe introducing implicit state machines as a programming model will improve the situation.

3.1 Declarative Programming

It is well-known in the programming language community that a declarative language enjoys many advantages over an imperative language. The mainstream languages for digital design, such as VHDL and Verilog, are in imperative style.

A *declarative* language is easier to work with than an imperative one. Declarative programs are easier to compose and reason about, as we may *substitute equals for equals* [29]: given an equation $x = t$ in the program, we may safely substitute the variable x with the code t without changing semantics of the program. In contrast, such substitution is generally problematic in imperative programs. Consequently, it is much easier to perform semantic-preserving transformations and optimizations of declarative programs than of imperative programs.

Imperative programming with states faces the problem of double assignment. In the Verilog code example below, the variable a is assigned twice when c is true:

```

1  always @ (posedge clk )
2      if (enable) begin
3          a <= c & d;  b <= c | d;
4          if (c) a <= b;    // double assignment of a if c is true
5          end else a <= d;    // b not assigned in else branch
6      end

```

Most languages take the last assignment as effective in the case of double assignment. The fact that such code is supported is a little counter-intuitive as all registers are refreshed exactly once on each clock tick in synchronous digital circuits. What is worse is that double assignment could be mistakes made by the programmer, for which the compiler is helpless to address.

Such problems are inherent in imperative programming with states. However, a stateful computation does not need to be in imperative style. The synchronous dataflow model in Lustre [6] and Signal [4] is one evidence for this. Yet it is unknown how to make programming with FSMs declarative, as they are stateful computation by nature, and past proposals on programming with FSMs are all in imperative style [19, 8]. With implicit state machines, we show how to program with FSMs in declarative style.

It is reported that dataflow programming is a good fit for dataflow-dominated applications, while FSM-based imperative programming is a more suitable for control-dominated applications [3, 8]. The FSM extension to Lustre [8] comes from the need to support both styles in the same language, in which FSMs desugar to a core dataflow calculus. Our calculus of implicit state machines can be seen as another synergy of dataflow programming and imperative programming. The expression-oriented nature of the calculus makes dataflow

programming easy. Meanwhile, an implicit state machine with an explicit case for each state is a good fit for control-dominated applications.

3.2 Scalable Abstraction

It is well-known that abstraction is the way to control complexity and build complex systems. Boolean algebra saves digital designers from transistors and resistors. It is a pity that Chisel [1, 18, 15], the latest hardware construction language that gains traction, still promotes programming with wires and connections. If we examine Chisel, VHDL, and Verilog closely, it is not clear what is the core calculus which plays the role of lambda calculus for functional programming.

With implicit state machines, we eliminate wires, connections, registers and flip-flops from hardware design. We cannot imagine what else can be removed further, as mathematicians would have discovered the simpler formalism and replaced FSMs with it.

Implicit state machine is a scalable abstraction. It may succinctly describe the most basic building blocks of digital design, such as D flip-flops, as well as complex systems via hierarchical and parallel composition. Any synchronous digital system that may be characterized by an FSM can be programmed with implicit state machines, because the transition function of implicit state machines can be both stateful and stateless, that latter corresponds to the transition function of FSMs.

Explicit state machines, i.e. state machines with one separate case for each state are implicit state machines by definition. It means programmers can freely choose to program with explicit states or implicit states. Some circuits are simpler to program implicitly, such as that of D flip-flop. The D flip-flop representation with implicit state machines only takes one line:

```
fsm { 0 | s => (d, s) }
```

However, explicit representation in a truth table would take several lines:

s	d	s'	Q
0	0	0	0
0	1	1	0
1	0	0	1
1	1	1	1

The D flip-flop is so simple that digital designers seldom think them as an FSM in programming. Programming with FSM in Verilog and VHDL is just a design methodology, with implicit state machines, it becomes a reality.

3.3 Acyclic by Construction

It is common to compose FSMs in digital design, as hierarchical decomposition is a widely used method to break down a complex system. In Verilog and VHDL, FSM is not a primitive programming construct. They are usually encoded with registers in separate modules, and then the modules are composed. Such composition, however, is dangerous, as combinational cycles may arise from the composition of FSMs [25, 2]. The combinational cycles resulted from FSM composition is illustrated in Figure 4.

Despite the fact that combinational cycles have been studied theoretically [27, 22], in practice they represent mistakes in the design and CAD tools for synthesis and verification

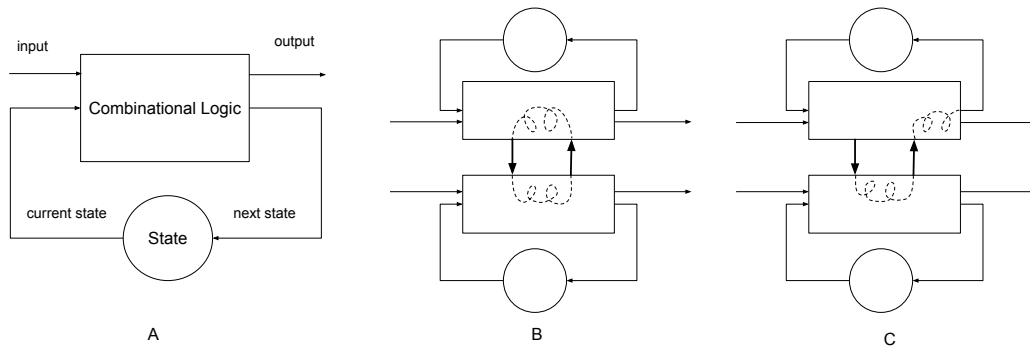


Figure 4 FSM composition. (A) An FSM in circuit, where the combinational logic is acyclic. (B) The connection of two FSMs results in combinational cycles. (C) The connection does not result in combinational cycles, as the feedback to the upper FSM only goes to the state element, which breaks the loop.

require circuits without combinational cycles as input. In our calculus, there are no combinational cycles by construction. To compose two FSMs as in Figure 4B, a digital designer has to write the following code:

```

372   fsm { v3 | s3 =>
373       let o1 = fsm { v1 | s1 => t1 } in
374       let o2 = fsm { v2 | s2 => t2 } in
375       (t3, (o1, o2))
376   }
```

In the code above, another FSM is created with the state name $s3$, which is the shared state that decouples the combinational loop.

In the case where the connection in Figure 4C does not result in combinational cycles, i.e. one feedback only goes into the state elements but not output, there is no need to create an additional FSM:

```

382   fsm { v1 | s1 =>
383       let o1 = t1 in
384       let o2 = fsm { v2 | s2 => t2 } in
385       (t3, (o1, o2))
386   }
```

In the above, the next state and output of the inner FSM, i.e. $t2$, may depend on $o1$. Meanwhile, the next state of the outer FSM, i.e. $t3$, may depend on $o2$. The code is guaranteed to be acyclic by construction.

3.4 Logic Synthesis

De Micheli [9] mentioned that sequential synthesis is hindered by combinational boundaries: typical optimizations extract combinational logic from the register-separated circuit network and optimize the combinational fragments only. The flattening of FSMs can transform any circuit into an equivalent circuit with state elements at the boundary and a big combinational core in the center. We conjecture such a transformation will facilitate optimizations as well

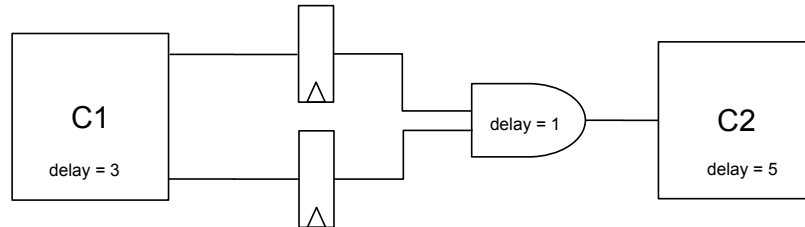
23:14 Digital Design with Implicit State Machines

as enable more optimization opportunities. We leave the conjecture to be substantiated by future research.

An expert in logic synthesis might wonder, what is the impact of flattening on *area* and *delay*, the two goals of logic optimizations? The answer is that they are unchanged. The reason is that during flattening, we only introduce let-bindings, it neither creates additional gates nor changes the number of gates on any path. With implicit state machines, experts in logic synthesis no longer need to worry about combinational boundaries any more.

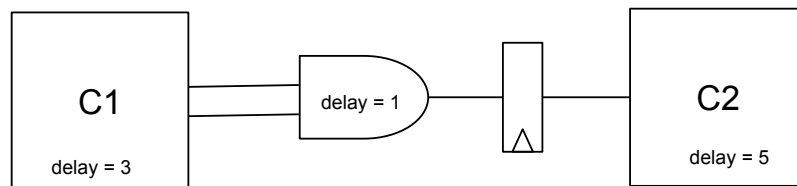
Already in 1991, Malik [23] envisioned the possibility of applying combinational techniques to optimizing sequential circuits by pushing registers to the boundary of the circuit network, and cut the loops when needed. The approach taken by Malik is based on a technique called *retiming* [17], which changes the timing behaviors of the circuit by moving registers around in the circuit network.

Our approach essentially follows the same spirit. However, we achieve the same goal without changing timing behavior of the circuit. The optimization opportunities enabled by retiming is different from ours, but it can be expressed based on top of implicit state machines. For example, given the circuit network below:



The circuit above shows that two outputs of the sub-circuit C1 go to two different registers. The output of the two registers go to an AND gate and its output in turns goes to the sub-circuit C2. The critical path of the circuit has the delay 6. The critical path is the path in the circuit that has the maximum delay between an input signal or a register read, to an output signal or a register write. The period of clock in a synchronous circuit has to be bigger than the delay of the critical path.

Using retiming, we can push the two registers after the AND gate, which results in the following network:



Now the critical path of the circuit has a delay of 5 instead of 6, and it saves one register. If we represent the circuit C1 by the term t_1 , and the circuit C2 by the term t_2 , then the circuit before the retiming optimization can be expressed as follows:

```

let x = t1 in
let y = fsm { (0, 0) | s =>
  (x, s.1 & s.2)

```

```

428     }
429     in t2

```

430 In the above, x represents the two output signals of the circuit C1, and the input signal
 431 to the circuit C2 is represented by the variable y . The circuit after the retiming optimization
 432 can be expressed as follows:

```

433     let x = t1 in
434     let y = fsm { 0 | s =>
435         (x.1 & x.2, s)
436     }
437     in t2

```

438 If the AND gate in the original circuit is a XOR gate, then we also need to change the
 439 initial state of the transformed FSM in the above.

440 If we see it from another perspective, retiming transforms are just usage of laws of implicit
 441 state machines. In addition to the transformations presented in lifting and flattening, the
 442 following transformations may also serve as laws because they are semantic-preserving:

$$\begin{array}{lll}
 \llbracket \text{let } x = t_1 \text{ in } t_2 \rrbracket & = & [x \mapsto t_1]t_2 \quad \textit{inlining} \\
 \llbracket \text{fsm } \{ v \mid s \Rightarrow (v, t) \} \rrbracket & = & \text{let } s = v \text{ in } t \quad \textit{stable state} \\
 \llbracket \text{fsm } \{ v \mid s \Rightarrow (s, t) \} \rrbracket & = & \text{let } s = v \text{ in } t \quad \textit{const state} \\
 \llbracket \text{fsm } \{ v \mid s \Rightarrow (t_1, v_2) \} \rrbracket & = & v_2 \quad \textit{const output} \\
 \llbracket \text{fsm } \{ v \mid s \Rightarrow (t_1, t_2) \} \rrbracket & = & t_2 \text{ if } s \text{ is not free in } t_2 \quad \textit{fake state} \\
 \llbracket \text{fsm } \{ v \mid s \Rightarrow (t_1, t_2) \} \rrbracket & = & \text{let } x = t_1 \text{ in fsm } \{ v \mid s \Rightarrow (x, t_2) \} \quad \textit{simple state} \\
 & & \text{if } s \text{ is not free in } t_1 \\
 \llbracket \text{fsm } \{ v \mid s \Rightarrow (x, t_2) \} \rrbracket & = & \text{fsm } \{ v' \mid s \Rightarrow ([s \mapsto x]t_2, s) \} \quad \textit{retiming} \\
 & & \text{if } t_2 \xrightarrow{\sigma_0, \emptyset} v'
 \end{array}$$

444 The essence of retiming is succinctly expressed by the last rule, except the subtlety about
 445 the initial state: it requires that t_2 should evaluate to a value v' given the initial states for
 446 all FSMs in the program σ_0 . The empty environment enforces that t_2 may not depend on
 447 external inputs. Otherwise, we do not see how to preserve semantics in the transform.

448 4 Implicit State Machine in Scala

449 To test feasibility of making implicit state machines as a programming model, we implemented
 450 an embedded DSL in Scala for hardware construction.

451 4.1 A Quick Glance

452 The following code shows how we may implement a half adder in our DSL:

```

453
454 1  def halfAdder(a: Signal[Bit], b: Signal[Bit]): Signal[Vec[2]] = {
455 2      val s = a ^ b
456 3      val c = a & b
457 4      c ++ s
458 5  }
459

```

460 In the code above, the type `Signal[Bit]` means that `a` is a signal of 1 bit. The type
 461 `Signal[Vec[2]]` means a signal of width 2. Here we take advantage of *literal types* in Scala,

23:16 Digital Design with Implicit State Machines

which supports the usage of a literal constant as a type. The type `Bit` means the same as `Vec[1]`:

```
1 type Bit = Vec[1]
```

The DSL supports common bit-wise operations like XOR (`^`), AND (`&`), OR (`|`), ADD (`+`), SUB (`-`), SHIFT (`<<` and `>>`), MUX (if/then/else). The operator `++` concatenates two bit vector to form a bigger bit vector. All these operations are supported in Verilog [28], and they follow the same semantics as in Verilog.

We may compose two half adders to create a full adder, which takes a carry `cin` as input:

```
1 def full(a: Signal[Bit], b: Signal[Bit], cin: Signal[Bit]): Signal[Vec[2]] = {
2   val ab = halfAdder(a, b)
3   val s = halfAdder(ab(0), cin)
4   val cout = ab(1) | s(1)
5   cout ++ s(0)
6 }
```

In the above, we make two calls to `halfAdder`. Each call will create a copy of the half adder circuit to be composed in the fuller adder. It returns the carry and the sum. We may compose them further to create a 2-bit adder:

```
1 def adder2(a: Signal[Vec[2]], b: Signal[Vec[2]]): Signal[Vec[3]] = {
2   val cs0 = full(a(0), b(0), 0)
3   val cs1 = full(a(1), b(1), cs0(1))
4   cs1(1) ++ cs1(0) ++ cs0(0)
5 }
```

To actually generate a representation of the circuit, we need to specify the input signals:

```
1 val a = variable[Vec[2]]("a")
2 val b = variable[Vec[2]]("b")
3 val circuit = adder2(a, b)
```

Now we may generate Verilog code for the circuit:

```
1 circuit.toVerilog("Adder", a, b)
```

For testing purposes, we can call the interpreter to get the result for a specific input:

```
1 val add2 = circuit.eval(a, b)
2 val Value(c1, s1, s0) = add2(Value(1, 0) :: Value(0, 1) :: Nil)
3 assertEquals(c1, 0)
4 assertEquals(s1, 1)
5 assertEquals(s0, 1)
```

You might be wondering, what about a generic adder that generates circuits for a given width? This can be implemented with a recursion on the number of bits:

```
1 def adderN[N <: Num](lhs: Signal[Vec[N]], rhs: Signal[Vec[N]])
2 : Signal[Bit ~ Vec[N]] = {
3   val n: Int = lhs.size
4   def recur(index: Int, cin: Signal[Bit], acc: Signal[Vec[_]]) =
5     if (index >= n) cin ~ acc.as[Vec[N]]
6     else {
7       val cs: Signal[Vec[2]] = full(lhs(index), rhs(index), cin)
8       recur(index + 1, cs(1), (cs(0) ++ acc.as[Vec[N]]).asInstanceOf)
9     }
10
11   recur(0, lit(false), Vec().as[Vec[_]])
```



```
522 12 }
523
```

524 In the code above, the type `Signal[Bit ~ Vec[N]]` means a signal that is a pair, the left
 525 is one bit, the right is a bit vector of length `N`. To construct a signal of such a type, we just
 526 connect two signals with `~` as it is used at line 5. At line 8, we used several type cast in the
 527 code, due to the fact that Scala currently does not support arithmetic operations at type
 528 level.

529 4.2 Sequential Circuits

We show how to create sequential circuits with the example of moving average. The moving average filter we are going to implement is specified below:

$$Y_i = (X_i + 2 * X_{i-1} + X_{i-2})/4$$

530 For the input X_i , the output Y_i also depends on the previous values X_{i-1} and X_{i-2} . The
 531 FSM that delays a given signal by one clock can be implemented as follows:

```
532
533 1 def delay[T <: Type](sig: Signal[T], init: Value): Signal[T] =
534 2   fsm("delay", init) { (last: Signal[T]) =>
535 3     sig ~ last
536 4   }
537
```

538 In the code above, we declare an implicit state machine with the specified initial state
 539 `init`. The body of the FSM is a pair `sig ~ last`, where the first part becomes the next state,
 540 and the second part becomes the output. This is exactly the D flip-flop.

Now we may create the circuit for the moving average:

```
541
542
543 1 def movingAverage(in: Signal[Vec[8]]): Signal[Vec[8]] = {
544 2   let(delay(in, 0.toValue(8))) { z1 =>
545 3     let(delay(z1, 0.toValue(8))) { z2 =>
546 4       (z2 + (z1 << 1) + in) >> 2.W[2]
547 5     }
548 6   }
549 7   }
550
```

551 In the code above, we first create an instance of the delay circuit and bind it to the
 552 variable `z1`. Then we delay the signal `z1`, and bind it to `z2`. Finally, the computation is
 553 expressed on bit vectors.

Note that it is tempting to implement the same circuit without using the let-bindings:

```
554
555
556 1 def movingAverage(in: Signal[Vec[8]]): Signal[Vec[8]] = {
557 2   val z1 = delay(in, 0.toValue(8))
558 3   val z2 = delay(z1, 0.toValue(8))
559 4   (z2 + (z1 << 1) + in) >> 2.W[2]
560 5   }
561
```

562 The circuit, though functions the same, will need more gates to implement. The reason is
 563 that, in our DSL, the variable definition `z1` represents the D flip-flop circuit (not the signal),
 564 each usage of the variable `z1` will create a copy of the circuit. It is used twice, the circuit is
 565 thus duplicated twice. The way to avoid duplication is to use let-bindings, which serves the
 566 same role as that of wires: a bound variable may be used multiple times, just like a wire
 567 may forward the same signal to multiple gates.

568 The adder example in the previous section also suffers from this problem. However, to
 569 our surprise, the version without let-binding is optimized better by synthesis tools from our
 570 testing. This problem is common in meta-programming, i.e. write a program to generate

another program (possibly in another language). We believe linear type systems might be useful in such settings to ensure that method call results are used linearly, as a method usually synthesizes some piece of code, duplicate usage or no usage are usually mistakes. Meanwhile, method arguments should be non-linear, i.e., they may be used multiple times.

4.3 Optimizations

The synthesized code for the moving average example initially looks like the following (in a notation close to the calculus):

```

579 1   let x: Vec[8] = fsm { 0 | delay => a ~ delay }
580 2   in
581 3       let x1: Vec[8] = fsm { 0 | delay1 => x ~ delay1 }
582 4       in (x1 + (x << 1) + a) >> 2
583

```

After lifting of FSMs, we get the following code:

```

586 1   fsm { 0 | delay =>
587 2       fsm { 0 | delay1 =>
588 3           let x6: Vec[8] ~ Vec[8] = a ~ delay
589 4           in
590 5               let x: Vec[8] = x6.2
591 6               in
592 7                   let x8: Vec[8] ~ Vec[8] =
593 8                       let x7: Vec[8] ~ Vec[8] = x ~ delay1
594 9                       in
595 10                           let x1: Vec[8] = x7.2
596 11                           in (x7.1 ~ x1 + (x << 1) + a) >> 2
597 12                           in x8.1 ~ x6.1 ~ x8.2
598 13                   }
599 14       }
600

```

As expected, a lot of unnecessary let-bindings are introduced, and the flattening of FSMs will introduce several more let bindings. To eliminate such bindings, we first transform the code into A-normal form (ANF), then perform detupling that reduces pairs to bit vectors, and finally inline trivial let-bindings. In the end, we get the following compact code:

```

606 1   fsm { 0 | state =>
607 2       a ++ state(15..8) ++ ((state(7..0) + (state(15..8) << 1) + a) >> 2)
608 3   }
609

```

Eventually, the generated Verilog code looks like the following:

```

612 1 module Filter (CLK, a, out);
613 2 input CLK;
614 3 input [7:0] a;
615 4 output [7:0] out;
616 5 wire [7:0] out;
617 6 reg [15:0] state;
618 7
619 8 assign out = ( ( ( state[7:0] + ( state[15:8] << 1'b1 ) ) + a ) >> 2'b10 );
620 9
621 10 initial begin
622 11     state = 16'b0000000000000000;
623 12 end
624 13
625 14 always @ (posedge CLK)

```

```

626 15   state <= { a, state[15:8] };
627 16   endmodule
628

```

629 In the Verilog code above, only the following line updates the state of the FSM, other
630 lines compute the next state and output:

```

631
632 1   always @ (posedge CLK)
633 2     state <= { a, state[15:8] };
634 3   endmodule
635

```

636 This is the typical code generated by our DSL compiler, all the code is combinational
637 except one line, no matter how complex the circuit is. Is the generated Verilog efficient? For
638 curiosity, we implemented the moving average filter in Chisel:

```

639
640 1   class MovingAverage3 extends Module {
641 2     val io = IO(new Bundle {
642 3       val in = Input(UInt(8.W))
643 4       val out = Output(UInt(8.W))
644 5     })
645 6     val z1 = RegNext(io.in)
646 7     val z2 = RegNext(z1)
647 8     io.out := (io.in + (z1 << 1.U) + z2) >> 2.U
648 9   }
649

```

Chisel generates the following Verilog code after removing comments and the reset input:

```

650
651
652 1   module MovingAverage3(
653 2     input      clock,
654 3     input [7:0] io_in,
655 4     output [7:0] io_out
656 5   );
657 6     reg [7:0] z1;
658 7     reg [7:0] z2;
659 8     wire [8:0] _GEN_0;
660 9     wire [8:0] _T_12;
661 10    wire [8:0] _GEN_1;
662 11    wire [9:0] _T_13;
663 12    wire [8:0] _T_14;
664 13    wire [8:0] _GEN_2;
665 14    wire [9:0] _T_15;
666 15    wire [8:0] _T_16;
667 16    wire [8:0] _T_18;
668 17    assign _GEN_0 = {{1'd0}, z1};
669 18    assign _T_12 = _GEN_0 << 1'h1;
670 19    assign _GEN_1 = {{1'd0}, io_in};
671 20    assign _T_13 = _GEN_1 + _T_12;
672 21    assign _T_14 = _GEN_1 + _T_12;
673 22    assign _GEN_2 = {{1'd0}, z2};
674 23    assign _T_15 = _T_14 + _GEN_2;
675 24    assign _T_16 = _T_14 + _GEN_2;
676 25    assign _T_18 = _T_16 >> 2'h2;
677 26    assign io_out = _T_18[7:0];
678 27    always @(posedge clock) begin
679 28      z1 <= io_in;
680 29      z2 <= z1;
681 30    end
682 31  endmodule
683

```

Now we run the synthesis tool Yosys¹ on both files, we get the following result:

	wires	wire bits	public wires	public wire bits	cells
Chisel (original)	73	147	11	85	85
Chisel (after correction)	59	106	8	55	73
Our DSL	55	84	4	33	73

For all columns, lower is better. The most important is last column `cells`, which says the number of gates required to implement the circuit. The column `wires` means the total number of wires in the synthesized design, the column `wire bits` means the total number of wires in bits, as wires may be wider than 1 bit. The column `public wires` means the wires that exist in the original design, i.e. not created by Yosys, the column `public wire bits` is similar.

The difference between the first two lines comes from the fact that Chisel handles `<<` by incrementing the width of the result, it thus increases wires and gates. Our DSL follows the semantics of Verilog, i.e. to keep the result the same width as the shifted bit vector. After the correction of the semantics for `<<`, Chisel uses the same number of gates as our DSL, and our DSL still performs better on wire bits. This shows that at least for simple circuits, our DSL compiler generates efficient circuits on par with the industry-level DSL.

4.4 Case Study: Microcontroller

To further test the usability of the DSL, we implemented a 2-stage accumulator-based microcontroller. The microcontroller supports 20 instructions:

NOP, ADD, ADDI, SUB, SUBI, SHL, SHR, LD, LDI, ST, AND, ANDI, OR, ORI, XOR, XORI, BR, BRZ, BRNZ, EXIT

- NOP is the no-op. EXIT is used for testing.
- Arithmetic operations have two versions, those with immediate operands (such as ADDI and ORI) and those with indirect operands (such as ADD and OR).
- SHL and SHR always have immediate operands.
- LD loads a data from memory to the accumulator. LDI puts the immediate operand in the accumulator. ST stores the value in the accumulator to a memory address.
- BR is unconditional jump. BRZ will jump to the operand address if the accumulator is zero. BRNZ is the opposite of BRZ.

The controller interfaces with a bus, which make the requested data on bus in the next clock cycle:

```
1 type BusOut = Vec[8] ~ Bit ~ Bit ~ Vec[32] // addr ~ read ~ write ~ writedata
2 type BusIn = Vec[32] // read data
```

The signature of the microcontroller generator is as follows:

```
1 def processor(prog: Array[Int], busIn: Signal[BusIn]): Signal[BusOut ~ Debug]
```

It takes a program `prog` to store in a on-chip instruction memory, which is different from the external memory connected by the bus. Note that the output type is `BusOut ~ Debug`, where we add `Debug` for testing purposes:

¹ <https://github.com/YosysHQ/yosys>

```

724
725 1   type Debug = Vec[32] ~ Vec[_] ~ Vec[16] ~ Bit // acc ~ pc ~ instr ~ exit
726

```

Note that the width of the program counter PC is unspecified, because it depends on the size of the given program. If the program size is 62, then the width is 6.

At the high-level, the microcontroller is an FSM which contains three architectural states:

```

729
730 1   fsm("processor", pc0 ~ acc0 ~ pending0) { (state: Signal[PC ~ ACC ~ INSTR]) =>
731 2     val pc ~ acc ~ pendingInstr = state
732 3   }
733
734

```

The variable `pc` refers to the program counter, `acc` is the accumulator register, `pendingInstr` is the instruction from the last cycle waiting for data from the external memory. The type `ACC` and `INSTR` are aliases of `Vec[32]` and `Vec[16]` respectively. The type `PC` is an alias of `Vec[addrWidth.type]`, where `addrWidth` is a local variable computed from the program size.

The skeleton of the implementation is as follows:

```

739
740 1   let("pcNext", pc + 1.W[addrWidth.type]) { pcNext =>
741 2     let("instr", instrMemory(addrWidth, prog, pc)) { instr =>
742 3       let("stage2Acc", stage2(pendingInstr, acc, busIn)) { acc =>
743 4         when (opcode === ADDI.W[8]) {
744 5           val acc2 = acc + operand
745 6           next(acc = acc2)
746 7         } /* ... */ } }
747
748

```

It first increments the program counter `pc` and bind the result to `pcNext`. Then it binds the current instruction to `instr`. Next, it gets the updated value of the accumulator from the pending instruction. At the circuit-level, the three operations are executed in parallel. Finally, the instruction is decoded and executed in a series of `when` constructs. The `when` construct is a syntactic sugar created from the built-in multiplexer that supports selecting one of two n-bit inputs by a single bit control. Eventually, each branch calls the local method `next` with appropriate arguments:

```

755
756 1   def next(
757 2     pc: Signal[PC] = pcNext,
758 3     acc: Signal[ACC] = acc,
759 4     pendingInstr: Signal[INSTR] = 0.W[16],
760 5     out: Signal[BusOut] = defaultBusOut,
761 6     exit: Boolean = false
762 7   ): Signal[(PC ~ ACC ~ INSTR) ~ (BusOut ~ Debug)] = {
763 8     val debug = acc ~ (pc.as[Vec[_]]) ~ instr ~ exit
764 9     (pc ~ acc ~ pendingInstr) ~ (out ~ debug)
765 10  }
766
767

```

As can be seen from above, the method `next` defines default values for all arguments, such that each branch may only specify parameters that are different. For example, the following are the code for unconditional jump `BR` and indirect addition `ADD`:

```

770
771 1   } .when (opcode === BR.W[8]) {
772 2     next(pc = jmpAddr)
773 3   } .when (opcode === ADD.W[8]) {
774 4     next(out = loadBusOut, pendingInstr = instr)
775 5   }
776
777

```

The implementation for the method `stage2` just checks the pending instructions, and computes the updated accumulator value from the bus input. If the pending instruction is `NOP`, it simply returns the current value of the accumulator.

The on-chip instruction memory is implemented by generating nested conditional expressions. Each condition tests whether the input address is equal to a memory address, if true, the instruction at the address is returned in the same clock cycle (they are combinational circuits):

```

781
782
783
784
785
786 1  def instrMemory(addrWidth: Int, prog: Array[Int],
787 2    addr: Signal[Vec[addrWidth.type]]): Signal[Vec[16]] = {
788 3    val default: Signal[Vec[16]] = 0.W[16]
789 4    (0 until (1 << addrWidth)).foldLeft(default) { (acc, curAddr) =>
790 5      when[Vec[16]] (addr === curAddr.W[addrWidth.type]) {
791 6        if (curAddr < prog.size) prog(curAddr).W[16]
792 7        else default
793 8      } otherwise {
794 9        acc
795 10      }
796 11    }
797 12  }
798

```

We test the implementation with small assembly programs. Despite the allure of successfully running simple assembly programs, we are aware that the microcontroller is still too simple and it may not match quality standards. Our next goal is to implement RISC-V cores and compare with the state-of-the-art open source implementations by standard metrics.

5 Related Work

Statecharts [12] is a visual formalism which supports hierarchical states and orthogonal states. Its formal semantics is subtle, and was given several years later after its first introduction [14, 24, 11, 13]. Hierarchical states do not automatically give rise to hierarchical FSMs required for hierarchical module composition in circuit design. In a sense, hierarchical states and hierarchical FSMs are two orthogonal concepts, as hierarchical FSMs do not imply hierarchical states either. Implicit state machines do not support hierarchical states natively, but such an extension is conceptually possible, though what they should look like and whether they are useful in digital design is open to debate. Implicit state machines just do not mandate one separate case for each state in the program, but do not forbid them, hierarchical or not.

An extension of hierarchical FSMs [8] is experimented in Lucid Synchrone [7] and integrated in the declarative dataflow language Lustre [6]. The extension is in imperative style, and it desugars to a core dataflow calculus. Since the state machines need to define a transition for each state separately, their code representation suffers from exponential blowup after flattening.

Caisson [19] is an imperative language for digital design, which supports nested states and parameterized states. The language contains both registers and FSM as primitive constructs. In contrast, our approach is more fundamental in that it makes implicit state machines as the only primitive construct.

Malik [23] proposed the usage of combinational techniques to optimizing sequential circuits by pushing registers to the boundary of the circuit network, and cut the loops when needed. The approach is based on a technique called *retiming* [17], which changes the timing behaviors of the circuit by moving registers around in the circuit network. We achieve the same goal without changing timing behavior of the circuit. The retiming optimization can be expressed on top of implicit state machines.

6 Conclusion

It is well-known that Boolean algebra is the calculus for combinational circuits. In this paper, we propose implicit state machines as the calculus for sequential circuits. Implicit state machines do not mandate one separate case for each state in the specification of an FSM. Compared to classic FSMs, implicit state machines support arbitrary parallel and hierarchical composition, which is crucial for real-world programming.

Compared to explicit state machines that require one separate case for each state, implicit state machines enjoy a nice property: any system of parallel and hierarchical implicit state machines may be flattened to a single implicit state machine without exponential blowup. For digital circuits, this means that any sequential circuit can be transformed into an equivalent circuit with state elements at the boundary, and a big combinational core in the center. This creates more optimization opportunities for digital circuits, and logic synthesis experts no longer need to worry about combinational boundaries anymore.

There are two directions for future work. First, implicit state machines, due to their composability, will make integrated and compositional specification in complex systems easier. Meanwhile, flattening may also flatten the specifications, which can then be fed into off-the-shelf verification tools, together with the flattened FSMs. In this sense, implicit state machines bridge the gap between complex systems and verification tools.

Second, implicit state machines may lead to new hardware architectures. For example, in FPGA architectures, currently state elements are scattered across the chip to support different kinds of sequential circuits. This architecture is still not flexible enough, and it is a waste of resource when the distribution of the state elements diverges too big from the circuit to be implemented on the FPGA chip. A possibility is to centralize all state elements, as any circuit is equivalent to a circuit with state elements at the boundary and a combinational core, of the same delay and area.

References

- 1 Jonathan Bachrach, Huy Vo, Brian C. Richards, Yunsup Lee, Andrew Waterman, Rimas Avizienis, John Wawrzynek, and Krste Asanovic. Chisel: Constructing hardware in a scala embedded language. *DAC Design Automation Conference 2012*, pages 1212–1221, 2012.
- 2 A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9), September 1991. URL: <http://ieeexplore.ieee.org/document/97297/>, doi:10.1109/5.97297.
- 3 A. Benveniste, P. Caspi, S.A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1), January 2003. URL: <http://ieeexplore.ieee.org/document/1173191/>, doi:10.1109/JPROC.2002.805826.
- 4 Albert Benveniste, Paul Le Guernic, and Christian Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Science of Computer Programming*, 16(2), September 1991. URL: <http://www.sciencedirect.com/science/article/pii/016764239190001E>, doi:10.1016/0167-6423(91)90001-E.
- 5 Jerry R Burch, Edmund M Clarke, Kenneth L McMillan, David L Dill, and Lain-Jinn Hwang. Symbolic model checking: 1020 states and beyond. *Information and computation*, 98(2), 1992.
- 6 P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE: A Declarative Language for Real-time Programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87, New York, NY, USA, 1987. ACM. event-place: Munich, West Germany. URL: <http://doi.acm.org/10.1145/41625.41641>, doi:10.1145/41625.41641.
- 7 Paul Caspi, Gregoire Hamon, Marc Pouzet, and Univ Paris-Sud. Synchronous Functional Programming: The Lucid Synchrone Experiment. 2008.

- 877 **8** Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. A conservative extension of synchronous
878 data-flow with state machines. In *Proceedings of the 5th ACM international conference on*
879 *Embedded software - EMSOFT '05*, Jersey City, NJ, USA, 2005. ACM Press. URL: <http://portal.acm.org/citation.cfm?doid=1086228.1086261>, doi:10.1145/1086228.1086261.
880
- 881 **9** G. De Micheli. Synchronous logic synthesis: algorithms for cycle-time minimization. *IEEE*
882 *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 10(1), January
883 1991. URL: <http://ieeexplore.ieee.org/document/62792/>, doi:10.1109/43.62792.
- 884 **10** Giovanni De Micheli, Robert K Brayton, and Alberto Sangiovanni-Vincentelli. Optimal
885 state assignment for finite state machines. *IEEE Transactions on Computer-Aided Design of*
886 *Integrated Circuits and Systems*, 4(3):269–285, 1985.
- 887 **11** Willem-Paul de Roever, Gerald Lüttgen, and Michael Mendler. What Is in a Step: New Per-
888 spectives on a Classical Question. In Zohar Manna and Doron A. Peled, editors, *Time for Veri-*
889 *fication*, volume 6200. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. URL: http://link.springer.com/10.1007/978-3-642-13754-9_15, doi:10.1007/978-3-642-13754-9_15.
890
- 891 **12** David Harel. Statecharts: a visual formalism for complex systems. *Science of Computer*
892 *Programming*, 8(3), June 1987. URL: <https://linkinghub.elsevier.com/retrieve/pii/0167642387900359>, doi:10.1016/0167-6423(87)90035-9.
893
- 894 **13** David Harel and Hillel Kugler. The Rhapsody Semantics of Statecharts (or, On the Executable
895 Core of the UML). In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg,
896 Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan,
897 Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Dough Tygar, Moshe Y. Vardi, Gerhard
898 Weikum, Hartmut Ehrig, Werner Damm, Jörg Desel, Martin Große-Rhode, Wolfgang Reif,
899 Eckehard Schnieder, and Engelbert Westkämper, editors, *Integration of Software Specification*
900 *Techniques for Applications in Engineering*, volume 3147. Springer Berlin Heidelberg, Berlin,
901 Heidelberg, 2004. URL: http://link.springer.com/10.1007/978-3-540-27863-4_19, doi:
902 10.1007/978-3-540-27863-4_19.
- 903 **14** David Harel and Amnon Naamad. The STATEMATE Semantics of Statecharts. *ACM Trans.*
904 *Softw. Eng. Methodol.*, 5(4), October 1996. URL: <http://doi.acm.org/10.1145/235321.235322>,
905 doi:10.1145/235321.235322.
- 906 **15** A. Izraelevitz, J. Koenig, P. Li, R. Lin, A. Wang, A. Magyar, D. Kim, C. Schmidt, C. Markley,
907 J. Lawson, and J. Bachrach. Reusability is firrtl ground: Hardware construction languages,
908 compiler frameworks, and transformations. In *2017 IEEE/ACM International Conference*
909 *on Computer-Aided Design (ICCAD)*, pages 209–216, Nov 2017. doi:10.1109/ICCAD.2017.
910 8203780.
- 911 **16** M. Keating. The simple art of soc design. 2011.
- 912 **17** Charles E. Leiserson and James B. Saxe. Retiming synchronous circuitry. *Algorithmica*,
913 6(1-6), June 1991. URL: <http://link.springer.com/10.1007/BF01759032>, doi:10.1007/
914 BF01759032.
- 915 **18** Patrick S. Li, Adam M. Izraelevitz, and Jonathan Bachrach. Specification for the firrtl language.
916 Technical Report UCB/EECS-2016-9, EECS Department, University of California, Berkeley,
917 Feb 2016. URL: <http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-9.html>.
- 918 **19** Xun Li, Mohit Tiwari, Jason K Oberg, Vineeth Kashyap, Frederic T Chong, Timothy Sherwood,
919 and Ben Hardekopf. Caisson: A Hardware Description Language for Secure Information Flow.
- 920 **20** Dan Luu. Verilog is weird. <https://danluu.com/why-hardware-development-is-hard/>.
921 Accessed: 2019-12-24.
- 922 **21** Dan Luu. Writing safe verilog. <https://danluu.com/pl-troll/>. Accessed: 2019-12-24.
- 923 **22** S. Malik. Analysis of cyclic combinational circuits. *Proceedings of 1993 International Conference*
924 *on Computer Aided Design (ICCAD)*, pages 618–625, 1993.
- 925 **23** Sharad Malik, Ellen M Sentovich, and Robert K Brayton. Retiming and Resynthesis: Optim-
926 izing Sequential Networks with Combinational Techniques.

- 927 24 A. Pnueli and M. Shalev. What is in a step: On the semantics of statecharts. In Takayasu Ito
928 and Albert R. Meyer, editors, *Theoretical Aspects of Computer Software*, Lecture Notes in
929 Computer Science, Berlin, Heidelberg, 1991. Springer. doi:10.1007/3-540-54415-1_49.
- 930 25 Daniel Sanchez. Minispec reference guide. [https://6004.mit.edu/web/_static/fall19/
931 resources/references/minispec_reference.pdf](https://6004.mit.edu/web/_static/fall19/resources/references/minispec_reference.pdf), 2019. Accessed: 2019-12-24.
- 932 26 Claude E. Shannon. A symbolic analysis of relay and switching circuits. *Transactions of the
933 American Institute of Electrical Engineers*, 57:713–723, 1938.
- 934 27 T.R. Shiple, V. Singhal, R.K. Brayton, and A.L. Sangiovanni-Vincentelli. Analysis of
935 combinational cycles in sequential circuits. In *1996 IEEE International Symposium on
936 Circuits and Systems. Circuits and Systems Connecting the World. ISCAS 96*, volume 4,
937 Atlanta, GA, USA, 1996. IEEE. URL: <http://ieeexplore.ieee.org/document/542093/>,
938 doi:10.1109/ISCAS.1996.542093.
- 939 28 IEEE Computer Society. *IEEE Standard for Verilog Hardware Description Language*. IEEE,
940 2005.
- 941 29 Harald Søndergaard and Peter Sestoft. Referential transparency, definiteness and unfoldability.
942 *Acta Informatica*, 27:505–517, 1990.
- 943 30 Lin Yuan, Gang Qu, Tiziano Villa, and Alberto Sangiovanni-Vincentelli. An fsm reengineering
944 approach to sequential circuit synthesis by state splitting. *IEEE Transactions on Computer-
945 Aided Design of Integrated Circuits and Systems*, 27(6):1159–1164, 2008.