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.

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

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

- How to support parallel and hierarchical composition of FSMs in a \textit{declarative} language?
- How to transform a complex system of FSMs into a flat FSM?

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

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

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

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

Our contributions are listed below:

- We introduce the concept of implicit state machines, and formalize the concept in a declarative calculus. Implicit state machines support parallel and hierarchical composition, and we may optimize and reason about the code by equational reasoning.
- We show that any parallel and hierarchical FSMs can be flattened into a flat implicit state machine in polynomial time and code size. As far as we know, this is the first abstraction for hierarchial and parallel FSMs that avoids exponential blowup in flattening.
- To the best of our knowledge, we are the first to theorize that any synchronous sequential circuits is equivalent to a circuit with all state elements at the boundary and a big combinational core at the center with the same area and delay.
- We create an embedded DSL in Scala based on implicit state machines, and the initial experiments show positive results when implicit state machine is used as a programming 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 $(I, S, s_0, \sigma, O)$:

- $I$ 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:

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

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 $t$:

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

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 $s$:

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

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

\[
\text{fsm} \{ s \Rightarrow t \} : O
\]

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

\[
\text{fsm} \{ v | s \Rightarrow t \} : O
\]

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:

\[
\text{fsm} \{ 0 | s \Rightarrow (d, s) \}
\]

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

\[
\begin{align*}
\text{let } q_1 &= \text{fsm} \{ 0 | s \Rightarrow (d, s) \} \text{ in} \\
\text{let } q_2 &= \text{fsm} \{ 0 | s \Rightarrow (q_1, s) \} \text{ in} \\
\text{let } q_3 &= \text{fsm} \{ 0 | s \Rightarrow (q_2, s) \} \text{ in} \\
\text{let } q_4 &= \text{fsm} \{ 0 | s \Rightarrow (q_3, s) \} \text{ in} \\
(q_1, q_2, q_3, q_4)
\end{align*}
\]

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

\[
\text{fsm} \{ (0, 0, 0, 0) | s \Rightarrow ((d, s.1, s.2, s.3), s) \}
\]

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

\[
\begin{align*}
\text{fsm} \{ 0 | q_1 \Rightarrow \\
\text{let } q_2 &= \text{fsm} \{ 0 | s \Rightarrow (q_1, s) \} \text{ in} \\
\text{let } q_3 &= \text{fsm} \{ 0 | s \Rightarrow (q_2, s) \} \text{ in} \\
\text{let } q_4 &= \text{fsm} \{ 0 | s \Rightarrow (q_3, s) \} \text{ in} \\
(d, (q_1, q_2, q_3, q_4))
\end{align*}
\]

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

### 2.2 Syntax

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

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

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

### 2.3 Semantics

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

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

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

- **E-VALUE**. If it is already a value, do nothing. There are no nested state machines, thus the mapping for the next state is the empty set.
- **E-INPUT**. Look up the external variable $a$ from the environment $\rho$.
- **E-LET**. First evaluate $t_1$ to the value $v_1$, then evaluate $t_2$ with $x$ replaced by $v_1$.
- **E-TUPLE**. Evaluate each component in parallel to a value, and accumulate the mapping for the next state.
- **E-PROJECT**. First evaluate the term to a tuple value, then return the corresponding component.
- **E-AND**. Evaluate the two components in parallel to Boolean values, then call the helper method $\text{and}$ to compute the resulting Boolean value $\beta$. As each component may contain implicit state machines, accumulate the mapping for the next state.
\[
\begin{align*}
  v & \xrightarrow{\sigma,\rho} v | \emptyset & \text{(E-Value)} \\
  v &= \rho(a) & \text{(E-Input)} \\
  a & \xrightarrow{\sigma,\rho} v | \emptyset \\
  t_1 & \xrightarrow{\sigma,\rho} v_1 | \sigma' \quad [x \mapsto v_1] t_2 & \xrightarrow{\sigma,\rho} v_2 | \sigma'' & \text{(E-Let)} \\
  & \text{let } x = t_1 \text{ in } t_2 & \xrightarrow{\sigma,\rho} v | \sigma' \cup \sigma'' \\
  t_1 & \xrightarrow{\sigma,\rho} v_1 | \sigma_1 \quad \ldots \quad t_n & \xrightarrow{\sigma,\rho} v_n | \sigma_n & \text{(E-Tuple)} \\
  (t_1, \ldots, t_n) & \xrightarrow{\sigma,\rho} (v_1, \ldots, v_n) | \sigma_1 \cup \ldots \cup \sigma_n \\
  t & \xrightarrow{\sigma,\rho} (v_1, \ldots, v_n) | \sigma' & \text{(E-Project)} \\
  t_1 & \xrightarrow{\sigma,\rho} \beta_1 | \sigma' \quad t_2 & \xrightarrow{\sigma,\rho} \beta_2 | \sigma'' & \beta = \text{and}(\beta_1, \beta_2) & \text{(E-And)} \\
  t_1 & \xrightarrow{\sigma,\rho} \beta_1 | \sigma' \quad t_2 & \xrightarrow{\sigma,\rho} \beta_2 | \sigma'' & \beta = \text{or}(\beta_1, \beta_2) & \text{(E-Or)} \\
  t_1 & \xrightarrow{\sigma,\rho} \beta | \sigma' \quad t_2 & \xrightarrow{\sigma,\rho} \beta | \sigma'' & \beta' = \text{not}(\beta) & \text{(E-Not)} \\
  v &= \sigma(s) \quad [s \mapsto v] t & \xrightarrow{\sigma,\rho} (v_1, v_2) | \sigma' & \text{(E-Fsm)} \\
  \text{fsm} \{ v \mid s \Rightarrow t \} & \xrightarrow{\sigma,\rho} v_2 | \{ s \mapsto v_1 \} \cup \sigma' 
\end{align*}
\]

**Figure 1** Big-step operational semantics

- **E-Or.** Similar as above, but use the helper function \textit{or} to compute the resulting value.
- **E-Not.** Similar as above, but use the helper function \textit{not} to compute the resulting value.
- **E-Fsm.** First look up the value for the current state from the state map \( \sigma \). Then evaluate the body of the state machine to a pair value \((v_1, v_2)\). The output is \( v_2 \), and the next state is \( v_1 \).

The reduction relation only defines one-step semantics. The semantics of a system is defined by the \textit{trace} of a given input series \( \rho_0, \rho_1, \ldots \). We define it formally below:

\textbf{Definition 1} (Trace). The \textit{trace} of a system \( t \) with respect to an input sequence \( \rho_0, \rho_1, \cdots \) is the sequence \( o_0, o_1, \cdots \) such that

\[
\begin{align*}
  t & \xrightarrow{\sigma_0,\rho_0} o_0 | \sigma_1 \\
  & \cdots \\
  t & \xrightarrow{\sigma_t,\rho_t} o_t | \sigma_{t+1} \\
  & \cdots \\
\end{align*}
\]

In the above, \( \sigma_0 \) 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: \textit{Bool} for Boolean values and \((T_1, \ldots, T_n)\) for tuples. We explain the typing rules below:

\[
T ::= \text{Bool} \mid (T_1, \ldots, T_n)
\]

\[
\begin{align*}
\Gamma \vdash \beta : \text{Bool} & \quad \text{(T-BOOL)} \\
\Gamma \downarrow t : (T_1, \ldots, T_n) & \quad \text{(T-PROJECT)} \\
\Gamma \vdash t.i : T_i & \\
\end{align*}
\]

\[
\begin{align*}
a : T \in \Gamma & \quad \text{(T-INPUT)} \\
\Gamma \vdash a : T & \\
\end{align*}
\]

\[
\begin{align*}
x : T \in \Gamma & \quad \text{(T-VAR)} \\
\Gamma \vdash x : T & \\
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash t_1 : \text{Bool} & \quad \Gamma \vdash t_2 : \text{Bool} \\
\Gamma \vdash t_1 + t_2 : \text{Bool} & \quad \text{(T-OR)} \\
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash t_1 : T_1 & \quad \Gamma \vdash t_2 : T_2 \\
\Gamma \downarrow \text{let } x = t_1 \text{ in } t_2 : T_2 & \quad \text{(T-LET)} \\
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash t_1 : T_1 & \quad \Gamma, x : T_1 \vdash t_2 : T_2 \\
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash v : T_1 & \quad \Gamma, s : T_1 \vdash t : (T_1, T_2) \\
\Gamma \downarrow \text{fsm} \{ \{ v \mid s \Rightarrow t \} : T_2 & \quad \text{(T-FSM)} \\
\end{align*}
\]

\textbf{Figure 2} Type System

- **T-BOOL.** The type for Boolean values is always \textit{Bool}.
- **T-INPUT.** For inputs, their types are predefined in the environment.
- **T-VAR.** For variables, their types also appear in the environment.
- **T-Not.** The term \(t\) must be \textit{Bool}.
- **T-TUPLE.** If each component has a type, and then the type of the tuple has a corresponding tuple type.
- **T-PROJECT.** If the term \(t\) has a tuple type, then the projection has the type of the corresponding component.
- **T-AND.** If each component has the type \textit{Bool}, the result also has the type \textit{Bool}.
- **T-Or.** The same as above.
- **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 \(\Gamma\) 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.
- **T-FSM.** If the initial value has the type \(T_1\), and the body has the type \((T_1, T_2)\) under the environment \(\Gamma\) extended with the binding \(s : T_1\), then the FSM has the type \(T_2\).

We need an auxiliary definition of \textit{value map typing}:
\( \Gamma \vdash \emptyset \) \hspace{1cm} \( \Gamma \vdash \xi \) \hspace{1cm} \( \Gamma \vdash v : T \)

\[ \Gamma, \alpha : T \vdash \xi \cup \{ \alpha \mapsto v \} \]

In the above, \( \alpha \) ranges over inputs \( a \) and state variables \( s \), and \( \xi \) ranges over input map \( \rho \) and state map \( \sigma \).

\textbf{Theorem 2 (Soundness).} If \( \Gamma \vdash t : T \), and if for each \( \rho_i \) in the input sequence \( \rho_0, \rho_1, \ldots \) 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:

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

\textbf{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:

\textbf{Lifting.} This step lifts FSMs to top-level.

\textbf{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 \( t_1 \sim_L t_2 \) 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 \( J \cdot K \), which transforms the source program to the expected form. We explain the concrete transform rules below:

\( \text{fsm} \{ \ v \mid s \Rightarrow e_1 \} * t_2 \). The FSM absorbs \( t_2 \) into its body. The symmetric case, and the cases for AND and OR are similar.

\( \text{let } x = \text{fsm} \{ \ v \mid s \Rightarrow e_1 \} \text{ in } t_2 \). It pulls the let-binding into the body. The case in which FSM is in the body of let-binding is similar.

\( \text{fsm} \{ \ v \mid s \Rightarrow e \} \cdot i \). It pulls the projection into the body of FSM.

\( \bar{e}, \text{fsm} \{ \ v \mid s \Rightarrow e \}, i \). 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 \( t_1 \sim_F t_2 \) 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 \( t_1 \sim t_2 \) to mean that \( t_1 \) takes either a lifting step \( (\sim_L) \) or a flattening step \( (\sim_F) \) to \( t_2 \). We write \( t_1 \sim^* t_2 \) to mean 0 or multiple such transform steps. For simplicity of presentation, we omit the formal definitions.
numbers of state machines in the code. Thus, the worse case is $t \sim^* \text{fsm} \{ v \mid s \Rightarrow e \}$ in $O(m \ast n)$ steps where $m$ is the size of the term $t$, and $n$ is the number of state machines in the code.

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

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

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

Figure 3 Flatting of nested FSMs. We write let $x, y = t_1 \in t_2$ as a syntactic sugar for let $z = t_1$ in let $x = z, 1 \in t_2$ in let $y = z, 2 \in t_2$.

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

\[
[N] = \text{fsm} \{ v \mid s \Rightarrow e \}
\]

\[
F[N] \sim_F F[\text{fsm} \{ v \mid s \Rightarrow e \}]
\]

\[
[\text{fsm} \{ v_1 \mid s_1 \Rightarrow \text{fsm} \{ v_2 \mid s_2 \Rightarrow e_2 \} \}] = \text{fsm} \{ (v_1, v_2) \mid s \Rightarrow let s_1, s_2 = s in let x = e_2 in ((x, 2, 1, x, 1, x, 2)) \}
\]

**FSM-free Fragment**

\[
e ::= v \mid e \ast e \mid e + e \mid !v \mid (e, \ldots, e) \mid e \cdot i \mid let x = e in e \mid x \mid s \mid a
\]

**Lifted Normal Form**

\[
N ::= e \mid \text{fsm} \{ v \mid s \Rightarrow N \}
\]

**Lifting**

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

\[
[[t]] = \text{fsm} \{ v \mid s \Rightarrow t' \}
\]

\[
L[t] \sim_L L[\text{fsm} \{ v \mid s \Rightarrow t' \}]
\]

\[
[\text{fsm} \{ v \mid s \Rightarrow e_1 \} \ast t_2] = \text{fsm} \{ v \mid s \Rightarrow let x = e_1 \text{ in } (x, 1, x, 2 \ast t_2) \}
\]

\[
[e_2 \ast \text{fsm} \{ v \mid s \Rightarrow e_1 \} \ast t_2] = \text{fsm} \{ v \mid s \Rightarrow let x = e_1 \text{ in } (x, 1, x, 2 \ast e_2) \}
\]

\[
[\text{fsm} \{ v \mid s \Rightarrow e_1 \} + t_2] = \text{fsm} \{ v \mid s \Rightarrow let x = e_1 \text{ in } (x, 1, x, 2 + e_2) \}
\]

\[
[e_2 + \text{fsm} \{ v \mid s \Rightarrow e_1 \} \ast t_2] = \text{fsm} \{ v \mid s \Rightarrow let x = e_1 \text{ in } (x, 1, x, 2 + e_2) \}
\]

\[
[\text{let } x = \text{fsm} \{ v \mid s \Rightarrow e_1 \} \text{ in } t_2] = \text{fsm} \{ v \mid s \Rightarrow let s_1, x = e_1 \text{ in } (s_1, t_2) \}
\]

\[
[\text{let } x = e_1 \text{ in } \text{fsm} \{ v \mid s \Rightarrow e_2 \}] = \text{fsm} \{ v \mid s \Rightarrow let x = e_1 \text{ in } e_2 \}
\]

\[
[\text{fsm} \{ v \mid s \Rightarrow e \} \cdot i] = \text{fsm} \{ v \mid s \Rightarrow let x = e \text{ in } (x, 1, x, 2, i) \}
\]

\[
[(\bar{e}, \text{fsm} \{ v \mid s \Rightarrow e \}) \cdot i] = \text{fsm} \{ v \mid s \Rightarrow let x = e \text{ in } (x, 1, \bar{e}, x, 2, i) \}
\]
let-bindings which changes the height of the tree, technically it is more complex to establish
the bound, we thus leave it to future work.

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

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

**Theorem 6 (Semantic Preserving).** If \( t \leadsto t' \), then they have the same trace for any given
input sequence \( \rho_0, \rho_1, \cdots \).

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

**Lemma 7.** If \( t \leadsto_L t' \), \( t \xrightarrow{\sigma,\rho} v \mid \sigma_1 \), then \( t' \xrightarrow{\sigma',\rho} v \mid \sigma'_1 \).

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

**Lemma 8.** If \( N \leadsto_T N' \), i.e. in the flattening \[ \text{fsm } \{ v_1 \mid s_1 \Rightarrow \text{fsm } \{ v_2 \mid s_2 \Rightarrow e_2 \} \} \]
of two state machines, let \( f = \lambda \sigma. \{ s \mapsto (\sigma(s_1),\sigma(s_2)) \} \cup (\sigma\{ s_1,s_2 \}) \), and \( f(\sigma) = \sigma' \).
\( N \xrightarrow{\sigma,\rho} v \mid \sigma_1 \), then \( N' \xrightarrow{\sigma',\rho} v \mid \sigma'_1 \) and \( f(\sigma_1) = \sigma'_1 \).

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

2.6 Discussion: Are Implicit State Machines FSMs?

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

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

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

```pseudocode
fsm { 0 | s =>
    when (s == 0) t1
    when (s == 1) t2
    when (s == 2) t3
    otherwise t4 }
```
In the above, we 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:

```verilog
always @ (posedge clk )
  if (enable) begin
    a <= c & d; b <= c | d;
    if (c) a <= b; // double assignment of a if c is true
  end else a <= d; // b not assigned in else branch
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:

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

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

```
<table>
<thead>
<tr>
<th>s</th>
<th>d</th>
<th>s' Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
```

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:

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

In the code above, another FSM is created with the state name \( s_3 \), 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:

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

In the above, the next state and output of the inner FSM, i.e. \( t_2 \), may depend on \( o_1 \). Meanwhile, the next state of the outer FSM, i.e. \( t_3 \), may depend on \( o_2 \). 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.
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:

```
let x = t1 in
let y = fsm { (0, 0) | s =>
    (x, s₁ & s₂)
```

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:

```
let x = t₁ in
let y = fsm { (0, 0) | s =>
    (x, s₁ & s₂)
```

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 t₁, and the circuit C₂ by the term t₂, then the
circuit before the retiming optimization can be expressed as follows:
In the above, \( x \) represents the two output signals of the circuit \( C_1 \), and the input signal to the circuit \( C_2 \) is represented by the variable \( y \). The circuit after the retiming optimization can be expressed as follows:

\[
\text{let } x = t_1 \text{ in } \text{let } y = \text{fsm} \{ 0 \mid s \Rightarrow (x.1 \& x.2, s) \} \text{ in } t_2
\]

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

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

\[
\begin{align*}
\text{let } x = t_1 \text{ in } t_2 & = [x \mapsto t_1][t_2] & \text{inlining} \\
\text{fsm} \{ v \mid s \Rightarrow (v, t) \} & = \text{let } s = v \text{ in } t & \text{stable state} \\
\text{fsm} \{ v \mid s \Rightarrow (s, t) \} & = \text{let } s = v \text{ in } t & \text{const state} \\
\text{fsm} \{ v \mid s \Rightarrow (t_1, v_2) \} & = v_2 & \text{const output} \\
\text{fsm} \{ v \mid s \Rightarrow (t_1, t_2) \} & = t_2 \text{ if } s \text{ is not free in } t_2 & \text{fake state} \\
\text{fsm} \{ v \mid s \Rightarrow (x, t_2) \} & = \text{let } x = t_1 \text{ in } \text{fsm} \{ v \mid s \Rightarrow (x, t_2) \} & \text{simple state} \\
& \quad \text{if } s \text{ is not free in } t_1 & \\
\text{fsm} \{ v \mid s \Rightarrow (x, t_2) \} & = \text{fsm} \{ v' \mid s \Rightarrow ([s \mapsto x][t_2, s) \} & \text{retiming} \\
& \quad \text{if } t_2 \sigma_0 \not\rightarrow v' & \\
\end{align*}
\]

The essence of retiming is succinctly expressed by the last rule, except the subtlety about the initial state: it requires that \( t_2 \) should evaluate to a value \( v' \) given the initial states for all FSMs in the program \( \sigma_0 \). The empty environment enforces that \( t_2 \) may not depend on external inputs. Otherwise, we do not see how to preserve semantics in the transform.

### 4 Implicit State Machine in Scala

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

#### 4.1 A Quick Glance

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

```scala
def halfAdder(a: Signal[Bit], b: Signal[Bit]): Signal[Vec[2]] = {
  val s = a ^ b
  val c = a & b
  c ++ s
}
```

In the code above, the type `Signal[Bit]` means that \( a \) is a signal of 1 bit. The type `Signal[Vec[2]]` means a signal of width 2. Here we take advantage of literal types in Scala,
which supports the usage of a literal constant as a type. The type Bit means the same as Vec[1]:

```scala
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:

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

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:

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

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

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

Now we may generate Verilog code for the circuit:

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

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

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

```scala
    val n: Int = lhs.size
    def recur(index: Int, cin: Signal[Bit], acc: Signal[Vec[_]]) =
        if (index >= n) cin ~ acc.as[Vec[N]
    else {
        val cs: Signal<Vec[2]] = full(lhs(index), rhs(index), cin)
        recur(index + 1, cs(1), (cs(0) ++ acc.as[Vec[N]])).asInstanceOf
    }
    recur(0, lit(false), Vec().as[Vec[N]])
```
In the code above, the type Signal[Bit ~ Vec[N]] means a signal that is a pair, the left is one bit, the right is a bit vector of length $N$. To construct a signal of such a type, we just connect two signals with ~ as it is used at line 5. At line 8, we used several type cast in the code, due to the fact that Scala currently does not support arithmetic operations at type level.

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:

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

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

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

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

Now we may create the circuit for the moving average:

```scala
1 def movingAverage(in: Signal[Vec[8]]): Signal[Vec[8]] = {
2 let(delay(in, 0.toValue(8))) { z1 =>
3 let(delay(z1, 0.toValue(8))) { z2 =>
4 (z2 + (z1 << 1) + in) >> 2.W[2]
5 }
6 }
7 }
```

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

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

```scala
1 def movingAverage(in: Signal[Vec[8]]): Signal[Vec[8]] = {
2 val z1 = delay(in, 0.toValue(8))
3 val z2 = delay(z1, 0.toValue(8))
4 (z2 + (z1 << 1) + in) >> 2.W[2]
5 }
```

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

The adder example in the previous section also suffers from this problem. However, to our surprise, the version without let-binding is optimized better by synthesis tools from our 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 synthesize 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):

```
let x: Vec[8] = fsm { 0 | delay => a ~ delay }
    in
        let x1: Vec[8] = fsm { 0 | delay1 => x ~ delay1 }
            in (x1 + (x << 1) + a) >> 2
```

After lifting of FSMs, we get the following code:

```
fsm { 0 | delay =>
    fsm { 0 | delay1 =>
            in
                let x: Vec[8] = x6.2
                    in
                        let x8: Vec[8] - Vec[8] =
                            let x7: Vec[8] - Vec[8] = x ~ delay1
                                in
                                    let x1: Vec[8] = x7.2
                                        in (x7.1 ~ x1 + (x << 1) + a) >> 2
                                            in x8.1 ~ x6.1 ~ x8.2
                            }
                    }
    }
```

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:

```
fsm { 0 | state =>
    a ++ state(15..8) ++ ((state(7..0) + (state(15..8) << 1) + a) >> 2)
}
```

Eventually, the generated Verilog code looks like the following:

```
module Filter (CLK, a, out);
    input CLK;
    input [7:0] a;
    output [7:0] out;
    wire [7:0] out;
    reg [15:0] state;
    assign out = ( ( ( state[7:0] + ( state[15:8] << 1'b1 ) ) + a ) >> 2'b10 );
initial begin
    state = 16'b0000000000000000;
end
always @(posedge CLK)
```
In the Verilog code above, only the following line updates the state of the FSM, other lines compute the next state and output:

```verilog
always @ (posedge CLK)
state <= { a, state[15:8] };
endmodule
```

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

```chisel
class MovingAverage3 extends Module {
  val io = IO(new Bundle {
    val in = Input(UInt(8.W))
    val out = Output(UInt(8.W))
  })
  val z1 = RegNext(io.in)
  val z2 = RegNext(z1)
  io.out := (io.in + (z1 << 1.U) + z2) >> 2.U
}
```

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

```verilog
module MovingAverage3(
  input clock,
  input [7:0] io_in,
  output [7:0] io_out);
  reg [7:0] z1;
  reg [7:0] z2;
  wire [8:0] _GEN_0;
  wire [8:0] _T_12;
  wire [8:0] _GEN_1;
  wire [9:0] _T_13;
  wire [8:0] _T_14;
  wire [8:0] _GEN_2;
  wire [9:0] _T_15;
  wire [8:0] _T_16;
  wire [8:0] _T_18;
  assign _GEN_0 = {{1’d0}, z1};
  assign _T_12 = _GEN_0 << 1’h1;
  assign _GEN_1 = {{1’d0}, io_in};
  assign _T_13 = _GEN_1 + _T_12;
  assign _T_14 = _GEN_1 + _T_12;
  assign _GEN_2 = {{1’d0}, z2};
  assign _T_15 = _T_14 + _GEN_2;
  assign _T_16 = _T_14 + _GEN_2;
  assign _T_18 = _T_16 >> 2’h2;
  assign io_out = _T_18[7:0];
  always @(posedge clock) begin
    z1 <= io_in;
    z2 <= z1;
  end
endmodule
```
Now we run the synthesis tool Yosys\(^1\) on both files, we get the following result:

<table>
<thead>
<tr>
<th></th>
<th>wires</th>
<th>wire bits</th>
<th>public wires</th>
<th>public wire bits</th>
<th>cells</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chisel (original)</td>
<td>73</td>
<td>147</td>
<td>11</td>
<td>85</td>
<td>85</td>
</tr>
<tr>
<td>Chisel (after correction)</td>
<td>59</td>
<td>106</td>
<td>8</td>
<td>55</td>
<td>73</td>
</tr>
<tr>
<td>Our DSL</td>
<td>55</td>
<td>84</td>
<td>4</td>
<td>33</td>
<td>73</td>
</tr>
</tbody>
</table>

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 $\ll$ 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 $\ll$, 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 date 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:

```scala
type BusIn = Vec[32] // read data
```

The signature of the microcontroller generator is as follows:

```scala
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
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:

```scala
fsm("processor", pc0 ~ acc0 ~ pending0) { (state: Signal[PC ~ ACC ~ INSTR]) =>
  val pc ~ acc ~ pendingInstr = state
}
```

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:

```scala
let("pcNext", pc + 1.W[addrWidth.type]) { pcNext =>
  let("instr", instrMemory(addrWidth, prog, pc)) { instr =>
    let("stage2Acc", stage2(pendingInstr, acc, busIn)) { acc =>
      when (opcode === ADDI.W[8]) {
        val acc2 = acc + operand
        next(acc = acc2)
      } /* ... */
    }
  }
}
```

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:

```scala
def next(
  pc: Signal[PC] = pcNext,
  acc: Signal[ACC] = acc,
  pendingInstr: Signal[INSTR] = 0.W[16],
  out: Signal[BusOut] = defaultBusOut,
  exit: Boolean = false
): Signal[(PC ~ ACC ~ INSTR) ~ (BusOut ~ Debug)] = {
  val debug = acc ~ (pc.as[Vec[_]]) ~ instr ~ exit
  (pc ~ acc ~ pendingInstr) ~ (out ~ debug)
}
```

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:

```scala
  .when (opcode === BR.W[8]) {
    next(pc = jmpAddr)
  }.
  .when (opcode === ADD.W[8]) {
    next(out = loadBusOut, pendingInstr = instr)
  }
```

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):

```java
def instrMemory(addrWidth: Int, prog: Array[Int],
    addr: Signal[Vec[addrWidth.type]]): Signal[Vec[16]] = {
  val default: Signal[Vec[16]] = 0.W[16]
  (0 until (1 << addrWidth)).foldLeft(default) { (acc, curAddr) =>
    when[Vec[16]] (addr === curAddr.W[addrWidth.type]) {
      if (curAddr < prog.size) prog(curAddr).W[16]
      else default
    } otherwise {
      acc
    }
  }
}
```

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


M. Keating. The simple art of soc design. 2011.


