Generating Circuits with Generators

MAREK MATERZOK, University of Wrocław, Faculty of Mathematics and Computer Science, Poland

The most widely used languages and methods used for designing digital hardware fall into two rough categories. One of them, register transfer level (RTL), requires specifying each and every component in the designed circuit. This gives the designer full control, but burdens the designer with many trivial details. The other, the high-level synthesis (HLS) method, allows the designer to abstract the details of hardware away and focus on the problem being solved. This method however cannot be used for a class of hardware design problems because the circuit's clock is also abstracted away.

We present YieldFSM, a hardware description language that uses the generator abstraction to represent clock-level timing in a digital circuit. It represents a middle ground between the RTL and HLS approaches: the abstraction level is higher than in RTL, but thanks to explicit information about clock-level timing, it can be used in applications where RTL is traditionally used. We also present the YieldFSM compiler, which uses methods developed by the functional programming community – including continuation-passing style translation and defunctionalization – to translate YieldFSM programs to Mealy machines. It is implemented using Template Haskell and the Clash functional hardware description language. We show that this approach leads to short and conceptually simple hardware descriptions.

CCS Concepts: • Hardware → Hardware description languages and compilation; • Software and its engineering → Domain specific languages; Control structures.

Additional Key Words and Phrases: hardware description languages, circuit synthesis, generators

1 INTRODUCTION

It is recognized by experts in the field that there is a need for rapid evolution of the architecture of computing hardware [Hennessy and Patterson 2019]. For this evolution to happen, it needs to be supported by increasing the efficiency of hardware design process. This is possible thanks to several developments in different fields. One is the emergence of reconfigurable hardware – in particular, field-programmable gate arrays (FPGAs). Thanks to them, the hardware can be developed and evaluated rapidly, like software, and used as domain-specific accelerators [Bacon et al. 2013]. Another is the development of open-source hardware design and synthesis tools, such as Yosys [Shah et al. 2019], which lower the barriers to entry into hardware design, much in the same way open-source compilers and tools achieved that for software. Finally, and most interestingly for the programming language community, there is the evolution of the languages used for designing digital hardware – the hardware description languages, or HDLs [Truong and Hanrahan 2019].

The HDLs which are traditionally used in the industry are Verilog/SystemVerilog [IEEE 2018] and VHDL [IEEE 2019], both of which emerged in the 1980s. These languages can be understood as concurrent programming languages, which model (or simulate) hardware as concurrently running,
A subset of these languages, often called the register transfer level style (or RTL), can be used for hardware synthesis. Unfortunately, there exist many pitfalls which lead to simulation/synthesis mismatch, i.e. the simulation and synthesis behavior of these languages diverge [Mills and Cummings 1999]. Another issue with these languages is their limited support for abstraction, which reduces the ways the designs written in these languages can be parametrized and reused [Bacon et al. 2013; Shacham et al. 2011].

The shortcomings of designing hardware using Verilog and VHDL were the driving force for creating a new family of hardware description languages. Their common feature is that they are embedded in an established programming language, often a functional one. This allows using the abstraction features of these languages to increase the abstraction level of the design, which improves e.g. readability, maintainability and code reuse. Notable languages belonging to this family are Chisel [Bachrach et al. 2012] and SpinalHDL [Papon 2017] (on Scala), Hardcaml [Clerc and Alekseyev 2018] (on OCaml) and Clash [Baaij et al. 2010] (on Haskell).

The languages described in the previous paragraph are still, in a way, low-level HDLs. This is because the basic concepts of these languages are registers, combinatorial circuits and connections. Therefore the code written in them directly specifies a particular digital circuit. This gives the designer full control over the final shape of the circuit and the way it is synchronized to the clock, which can be desirable for maximizing performance, but increases the number of details the designer is burdened with.

The programming language community has for a long time understood the need for improving the structure of code by abstracting from the details of the underlying programming model, like jumps [Dijkstra 1968] or memory addresses [Hoare 1975]. This need was the motivation for high-level synthesis (HLS) tools, which enable “compiling” algorithmic code to hardware. In high-level synthesis, the structure of the circuit and its use of the clock is abstracted away, freeing the designer to focus on the problem domain. Examples of tools in this family are Bambu [Pilato and Ferrandi 2013] and Legup [Canis et al. 2011], which take a subset of C/C++ as their source language.

It can be argued that HLS tools are a success: for a class of problems, they can be used to design a circuit with acceptable size and performance, while keeping its description high-level, readable, maintainable and amenable to verification [Sarkar et al. 2009]. But there are problems in hardware design for which HLS is a poor fit as a paradigm. Often, hardware is used not only to calculate something, but to communicate with other hardware, e.g. using some kind of protocol. In these applications, awareness and synchronization with the circuit’s clock is required. But in the HLS approach, the clock is abstracted away.

We believe there is a need for a hardware description language which is, in a way, in the middle of the two opposite approaches presented above. Such a language would abstract away enough of the circuit structure to allow clearer, more readable and manageable solutions to hardware design problems. Yet the circuit’s clock would still be explicit, which would allow one to use such a language when control over clock-level timing is required. In other words, we would like to raise the level of abstraction without losing expressivity in terms of the circuit’s observed behavior.

In this paper, we propose the programming language feature known as generators (as introduced in CLU [Liskov et al. 1977] and popularized in Python [Schemenauer et al. 2001]) as one well suited for designing circuits which need to have precisely defined behavior in relation to the clock. The most prominent language feature used in generators is the yield statement. It can be understood as a control flow operator which suspends the execution of a generator, while simultaneously providing a new value to the code (e.g. a loop) which invoked the generator. We propose that, in a hardware description language, the yield statement could be used to synchronize with the circuit’s clock; each yielded value would become the circuit’s output in consecutive clock cycles.
For example, a circuit whose output changes between high and low states on each clock cycle could be described as follows:

```plaintext
forever:
    yield True
    yield False
```

To explore this idea, we present a proof-of-concept hardware design language using generators called YieldFSM. In its implementation, we are using techniques developed by the functional programming community, including continuation-passing style (CPS) translation [Plotkin 1975] and defunctionalization [Reynolds 1972]. The YieldFSM language is embedded in the functional HDL Clash [Baaj et al. 2010] using Template Haskell. This design decision was made for several reasons, including:

- Simplified prototyping. Because Clash already is a mature, fully functional hardware design language, the development could focus on the core concepts of YieldFSM, leaving other matters to Clash. YieldFSM also benefits from various features of Haskell, including robust type inference.
- Simplified synthesis. Clash generates HDL code in Verilog and VHDL, which allows use of YieldFSM with standard simulation and testing tools used in hardware design.
- Deep integration into Clash. YieldFSM uses Haskell expressions as its expression language, which allows functions written in Clash to be used inside YieldFSM code. Conversely, the YieldFSM compiler generates Clash code, which can be then used in a larger Clash circuit. In this way, the implementation of YieldFSM on Clash allows describing different parts of the circuit using abstractions which are most adequate for them.

We use a number of small but practical problems in digital hardware design, which include an implementation of a RISC-V core [Waterman and Asanović 2019], to evaluate our solution. Our results are promising: the designs using YieldFSM are shorter in code size and conceptually easier to understand, while having similar or only slightly reduced performance. We believe that the ideas presented, with further refinement, could find their way into a hardware designer’s toolbox.

The rest of the article is structured as follows. In Section 2, we present background information important for understanding the presented results, which includes an introduction to Mealy and Moore machines (Section 2.1), digital circuits (Section 2.2), and the Clash hardware design language (Section 2.3). In Section 3, we present the YieldFSM language and its idioms informally by walking through several code examples. In Section 4 the implementation of the YieldFSM compiler is presented, with the focus on the key algorithms, which are presented in detail. In Section 5, we evaluate YieldFSM by applying it to problems of various complexity and comparing different solutions using code size and hardware performance metrics. Finally, we conclude in Section 6 with a short summary and a discussion of possible future developments.
2.1 Mealy and Moore Machines

Synchronous circuits can be abstractly modelled as Mealy machines [Mealy 1955]. A Mealy machine can be formalized as a tuple \( \langle \Sigma, \Omega, Q, q_0, \delta, X \rangle \) where:

- \( \Sigma \) is a (non-empty) input alphabet.
- \( \Omega \) is a (non-empty) output alphabet.
- \( Q \) is a (non-empty) set of states.
- \( q_0 \in Q \) is the initial state of the machine.
- \( \delta : Q \times \Sigma \rightarrow Q \) is the transition function.
- \( X : Q \times \Sigma \rightarrow \Omega \) is the output function.

In each time step, a Mealy machine reads a single input character, \( a \in \Sigma \), which together with the current state of the machine, \( s \in Q \), determines the next state \( \delta(s, a) \) and the output character \( X(s, a) \) produced in the given step.

If the output of a Mealy machine depends only on its current state (and thus does not depend on its input), it is also a Moore machine [Moore 1956]. Moore and Mealy machines can be thought of as a generalization of deterministic finite automata (DFA): a Moore machine where \( \Omega = \{0, 1\} \) is essentially a DFA.

To illustrate the concepts of Mealy and Moore machines, and other concepts introduced in this section, two variants of a counter machine will be used as examples. They differ only on the output function, \( X \) or \( X' \), where:

\[
\begin{align*}
\Sigma &= \{0, 1\} \\
\Omega &= Q = \{0, 1, 2, 3\} \\
q_0 &= 0 \\
\delta(n, i) &= (n + i) \mod 4 \\
X(n, i) &= \delta(n, i) \quad \text{(for a Mealy machine)} \\
X'(n, i) &= n \quad \text{(for a Moore machine)}
\end{align*}
\]

In the Mealy machine, the incremented value of the counter is available in the same time cycle as the “1” input which triggered it; while in the Moore machine, there is a single cycle delay. The difference in behavior can be illustrated by using an example. Consider the input sequence \([1, 0, 1, 1, 0, 0]\). For the Mealy counter, the sequence of outputs is \([1, 1, 2, 3, 3, 3]\). Whereas the Moore counter always outputs 0 in its first time cycle, and the changes in outputs occur one cycle later: \([0, 1, 1, 2, 3, 3]\).

2.2 Digital Circuits

Digital circuits are usually represented in graphical form using circuit diagrams (or schematics). A circuit diagram consists of a number of symbols (representing components), which are connected with lines (representing wires). Wires can only be connected to components in distinguished points called ports. The ports have different functions; in particular, they can be inputs or outputs.

Most components in digital logic are combinatorial: their outputs depend only on the values of their inputs. Examples include logic gates (AND, OR etc.) and complex components built using gates (e.g. multiplexers, adders). Some components are stateful: their outputs can retain some knowledge about the previous values of inputs. The most commonly used component of this kind is a register. Registers have a data input and a clock input. When the clock rises from 0 to 1, the output of the register changes to the current value of the data input. This value is retained until the next rising
clock edge. Registers also often have a reset input, which allows the value stored in the register to be set to some initial, known value.\footnote{In FPGAs, resets can often be omitted, because registers can be initialized during programming. In the rest of the paper, the issues related to reset signals will be ignored.}

In synchronous circuits, every register in the circuit has its clock input connected to the same clock signal. Alternatively, one can model synchronous circuits as having only one register, which contains the entire state of the circuit. Therefore, an entire synchronous circuit can be modelled as a single Mealy machine.

Figure 1 presents the circuit diagrams of the example counter machines. Their behavior can be presented using a timing diagram (Table 2). In it, the delays of the circuit elements occurring in real circuits are ignored. Each rising edge of the clock signal triggers loading a new value to the register, and corresponds to the beginning of the next time step of the machine.

2.3 Clash HDL

Clash \cite{Baaij:2010} is a hardware description language built upon Haskell. Its distinguishing feature is that it is not an embedded domain specific language (EDSL or DSEL \cite{Hudak:1996}) in a conventional sense. Instead, Clash defines a subset of Haskell that is synthesizable to hardware. Synthesizable Clash code has two possible semantics: it can either be executed as a Haskell program, or synthesized by the Clash compiler. The compiler operates on GHC Core expressions, which are transformed using a term rewriting system to a form that can be used to derive circuits from it \cite{Baaij:2014}.

In Clash there exists a subset of types that are representable in hardware, i.e. the values of these types can be encoded as bit strings of constant length. Non-recursive expressions of these types correspond to combinatorial circuits. Among these types are:

- \texttt{Bool}, \texttt{Int} and \texttt{Word} types.
- \texttt{Tuples}, \texttt{Maybe a} and \texttt{Either a b} (if their type arguments are representable).

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{example_circuit_diagram}
\caption{Circuit diagrams of the example Moore and Mealy counters.}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{example_timing_diagram}
\caption{Example timing diagram for the example Moore and Mealy counters.}
\end{figure}
• **Unsigned n, Signed n** and **BitVector n**, which all represent \(n\)-bit bit vectors, but have different instances of e.g. **Num** and **Show**.

• Any programmer-defined ADTs that are not recursive and only contain values of other representable types.

For the purpose of representing synchronous circuits, Clash introduces a type of time-varying signals, **Signal dom a**\(^2\), which is an instance of **Applicative**. Values of this type can be constructed using the **register** function:

\[
\text{register} :: \ a \rightarrow \text{Signal dom a} \rightarrow \text{Signal dom a}
\]

In the executable semantics, **Signal** denotes infinite streams, where **register** is its constructor. Pattern matching on this constructor is forbidden, as it would enable looking forwards in time and thus violating temporal causality. During synthesis, the **Signal** type gives rise to signals generated by potentially stateful, synchronous circuits. The **register** function corresponds to an actual register in the circuit: its first argument denotes the initial value of the register, while the second is the register’s input signal, which is captured on the rising edge of the (implicit) clock.

To define a Mealy machine in Clash, the following function can be used:

\[
\text{mealy} :: ([s \rightarrow i] \rightarrow ([s, o]) \rightarrow s \rightarrow \text{Signal dom i} \rightarrow \text{Signal dom o}
\]

\[
\text{mealy f iS i} = \text{where}
(s', o) = \text{unbundle $f <\$> s <\$> i}
\]

\[
\text{s} = \text{register iS s'}
\]

The first argument to **mealy** represents both the transition function and the output function of the automaton. The function **unbundle** transforms a signal of tuples to a tuple of signals.

Using this function, one can define the two example counter automata as follows:

\[
\text{count4\_mealy, count4\_moore} :: \text{Signal dom (Unsigned 1)}
\]

\[
\rightarrow \text{Signal dom (Unsigned 2)}
\]

\[
\text{count4\_mealy} = \text{mealy (\n i \rightarrow \text{let} s' = n + \text{extend i} \text{in} (s', s')) 0}
\]

\[
\text{count4\_moore} = \text{mealy (\n i \rightarrow (n + \text{extend i}, n)) 0}
\]

Alternatively, the Moore counter can be defined using the **moore** function, which takes the transition and output functions as separate parameters:

\[
\text{count4\_moore} = \text{moore (\n i \rightarrow n + \text{extend i}) id 0}
\]

The behavior of the automata defined above can be tested e.g. using the **simulateN** function, with the following (simplified) type:

\[
\text{simulateN} :: \text{Int} \rightarrow ([\text{Signal dom a} \rightarrow \text{Signal dom b}) \rightarrow [a] \rightarrow [b]
\]

The first argument denotes the number of cycles to be simulated. We can now validate that the defined automata behave as expected:

\[
> \text{simulateN @System 6 count4\_mealy [1,0,1,1,0,0] [1,1,2,3,3,3]}
\]

\[
> \text{simulateN @System 6 count4\_moore [1,0,1,1,0,0] [0,1,1,2,3,3]}
\]

In the example above, **@System** denotes a type application, which applies the type **System** – a built-in, default clock domain – to the type argument **dom**.

---

\(^2\)The type argument **dom** is used for representing clock domains, which is useful for dividing the circuit into multiple parts which use clocks of different frequency. This functionality is not used in this paper. For clarity, various constraints on this type argument occurring in actual Clash code will often be omitted.
3 A YIELDFSM TUTORIAL

We are now ready to discuss YieldFSM as a proof-of-concept language for designing finite state machines. This section will present it in an informal way, using a series of examples illustrating various features of the language. In the following, we will sometimes refer to YieldFSM automaton descriptions as “programs” – and even though they can be executed on a computer, the YieldFSM language is designed to be compiled to Mealy automata and it should be analyzed with this in mind.

3.1 Example Counters using YieldFSM

We’ll start the tutorial by considering the two counters introduced in the previous section. The Mealy counter can be written as follows:

```
fsm count4_mealy :: Signal dom (Unsigned 1) -> Signal dom (Unsigned 2)

input i
var n = 0
forever:
  n = n + extend i
  yield n

```

An automaton description in YieldFSM begins with a header. The first line names the automaton and gives its type. This type declaration is important for the purpose of type inference in the implementation. Additional type annotations are usually not required. The `[fsm| ... |]` brackets are for the purpose of Template Haskell quasiquotation [Mainland 2007], and will be omitted in the rest of the paper.

Next comes the `input` declaration. It defines a pattern the inputs are matched to. Every pattern variable in the `input` declaration names a variable, which is in scope in the entire code that follows. The value of this variable corresponds to a part of the automaton’s input in the current time step.

After the header follows the body, which is a YieldFSM statement. YieldFSM uses an indentation-sensitive syntax, where statements are separated by newlines.

The `yield` statement does two things. First, it decides the value of the automaton’s output for the current time step. Second, it advances the time: the statements coming after it will read input values for the next time step.

There are two possible ways to implement the Moore variant of the example counter. Both ways involve remembering the previous value of some signal using a `let` statement. One can store the previous value of the counter and yield it, instead of the incremented value:

```
fsm count4_moore :: Signal dom (Unsigned 1) -> Signal dom (Unsigned 2)

input i
var n = 0
forever:
  let prev_n = n
  n = n + extend i
  yield prev_n

```

Alternatively, one can store the previous value of `i`, and increment the counter after the `yield`:

```
fsm count4_moore :: Signal dom (Unsigned 1) -> Signal dom (Unsigned 2)

input i
var n = 0
forever:
  n = n + extend i
  yield n
```

---

3 This requirement simplifies parsing embedded Haskell expressions, which are handled by a separate parser. In this paper, we sometimes use a semicolon character instead of a newline for presentation purposes.
This is such a useful coding pattern in YieldFSM that syntactic sugar for it was added. A prime occurring after a variable name in YieldFSM means the value of this variable from a previous time cycle. Using this syntax, the `forever` loop for the Moore counter can be written as follows:

```yield
forever:
    let prev_i = i
    yield n
    n = n + extend prev_i
```

This syntactic sugar will be referred to as “magic primes”.

### 3.2 Conditional Statements

YieldFSM includes two kinds of conditional statements: the `if` and `case` statements. They have the usual semantics. The `if` statement selects one of two branches depending on the value of an expression of type `Bool`. The `case` statement has a number of clauses, which are selected by pattern matching. When several patterns match, the first one has priority. The patterns in `case` clauses are Haskell patterns.

As an example, the `forever` loop in the example Mealy counter can be alternatively written using an `if` statement as follows:

```yield
forever:
    if i == 1:
        n = n + 1
    yield n
```

### 3.3 Function Calls

The `forever` loop is not a primitive of YieldFSM. It is interpreted as syntactic sugar for a recursive function definition. Other kinds of loops that are supported in YieldFSM as syntactic sugar are `while`, `do-while`, and their counterparts with negated conditions: `until` and `do-until`.

A desugared version of the Mealy example is as follows (header was omitted).

```yield
var n = 0
fun loop ():
    n = n + extend i
    yield n
ret call loop ()
```

Functions in YieldFSM are written using `fun` blocks. The functions are first order and take a single argument, which is matched to a Haskell pattern. Function calls end with a `ret` (return) statement. It has two variants: normal `ret` with a value, or a tail call (`ret call`). The `call` keyword is used to distinguish calling a YieldFSM function from a Clash function. Examples of Clash functions in the above code are `(+)` and `extend`.

YieldFSM functions defined next to each other can be (mutually) recursive. Recursion in YieldFSM has the following limitations:

- Only tail recursion is allowed. This is due to the fact that non-tail recursion could require a potentially infinite stack, which cannot be represented in a finite state machine.
• In every cycle in the call graph, there needs to exist at least a single `yield`. This requirement, related to the notion of productivity in coinduction, ensures that the amount of computation that happens between the clock cycles of the generated automaton is bounded by a constant, and therefore can be represented by a combinatorial circuit.

For example, the following YieldFSM program is disallowed:

```plaintext
var n = 0
forever:
    if n < 3:
        n = n + 1
    else:
        yield n
```

Another place where YieldFSM function calls can occur, other than tail calls, are `let` definitions. Such calls (`let call`) cannot be recursive, so that the call stack size is bounded by a constant – and therefore the call stack can be entirely represented in the state of the automaton. There are no limitations on using `yield` in non-tail calls.

As an example use of `let call`, see the following program. On each cycle the input `i` is low, the described automaton outputs the number of consecutive cycles the input was previously high. On cycles with `i` high, the output is 0.

```plaintext
input i
fun count():
    var n = 0
    while i:
        n = n + 1
        yield 0
    ret n
forever:
    let n = call count()
    yield n
```

3.4 Named Outputs

In digital design practice it is often the case that the automaton in question has multiple outputs, each of which is actively used only in some subset of its states. For example, an automaton controlling a CPU might have separate outputs meant for controlling, e.g., an arithmetic logic unit, or a memory bus (see Section 5.3). Another example might be a device controller, which has one output connected to a common bus, while another output is connected to the device being controlled (see Section 5.2). These outputs typically have an “idle”, default value, which is overridden only in some states. To support this use case, YieldFSM has a feature called named outputs.

When using this feature, each output has a definition in the header, which names the output and defines its default value. The default value can be omitted – it is then assumed to be undefined (a “don’t care” in digital design terms). Then, on each `yield`, one names the outputs to be overridden inside angle brackets, and specifies their values using a tuple. Omitted outputs have the default value. For example, the automaton described below yields `[(1, 0), (1, 1), (0, 0)]` in a cycle:

```plaintext
output a = 0
output b = 0
forever:
    yield<a> 1
    yield<a, b> (1, 1)
yield
```
YieldFSM also introduces the `output` statement. Its role is to override the value of an output in the current cycle without yielding immediately. This feature exists for the purpose of separation of concerns: using `output`, different outputs can be managed in separate parts of the program. When appended to the previous example, the following two lines yield \((0, 1)\):

```haskell
output <b> 1
yield
```

### 3.5 Integration with Clash

Integration of YieldFSM with Clash goes both ways. In previous sections, we saw that the expression language of YieldFSM is Clash expressions: every Clash feature related to combinatorial logic can be used, including functions and data types defined in Clash. Here, we present how automata defined using YieldFSM can be used in Clash code.

In Section 3.1, we used YieldFSM to define a counter automaton, which outputs the number of ones in the input (modulo 4). It has the following type:

```haskell
count4_moore :: Signal dom (Unsigned 1) -> Signal dom (Unsigned 2)
```

Suppose we wish to count both the number of ones and zeros in the input. This problem could be solved by modifying the YieldFSM code from Section 3.1. But there is another, cleaner way to solve this problem: one can notice that the number of ones and the number of zeros can be counted by two separate, and almost identical, counting automata. We can use Clash to instantiate `count4_moore` twice as follows:

```haskell
count4_ones_zeros :: Signal dom (Unsigned 1) -> Signal dom (Unsigned 2, Unsigned 2)
count4_ones_zeros i = bundle (count4_moore i, count4_moore ((1 -) <$> i))
```

The integration with Clash allows to seamlessly use both languages in the same source file. This allows the designer to choose the appropriate abstractions for the problem being solved.\(^4\)

### 4 COMPILING YIELDFSM

As we recall from Section 3, YieldFSM is embedded in Clash using Template Haskell (TH), and uses Template Haskell expressions, patterns etc. in its syntax. The syntactic categories borrowed from Template Haskell are presented in Figure 3.

The goal of the YieldFSM compiler is to transform the automaton description to a form that can be easily encoded as a Mealy machine. In this section, we focus our attention on the core YieldFSM language, as described in Figure 4. The features of YieldFSM not included in the core language, i.e., loops, named outputs, references to previous input values, can be treated as syntactic sugar. Desugaring them is briefly described in Section 4.1.

In the following descriptions, the issues related to variable capture will be ignored for the clarity of presentation. It will be implicitly assumed that substitutions are capture-avoiding, variables are

\(^4\)In particular, the problem of clock domain crossing (connecting circuits using different clocks, represented in Clash by different `dom` type parameters) is solved using Clash, as each YieldFSM automaton uses a single clock domain.
Fig. 4. The YieldFSM core language.

not shadowed, and they are freshened when needed. The expression \( FV(e) \) is defined to be the set of free variables in a given expression (or statement, etc.). Similarly, \( BV(p) \) means the set of variables bound by a pattern.

### 4.1 Desugaring

The input program needs to be transformed into the core language (Figure 4) before the start of compilation. In this phase, features such as loops, named outputs and magic primes need to be translated away. In the following paragraphs, we briefly describe the desugaring method for these features features and other kinds of syntactic sugar used in YieldFSM.

First, notice that in the core language, function calls are allowed only in \( \text{let} \) and \( \text{ret} \) statements. For other statement types, a helper \( \text{let} \) statement is added during desugaring. For example, the statement \( \text{yield call } f() \) is desugared to \( \text{let } x = \text{call } f(); \text{yield } x \). Also, a \( \text{call } f() \) statement (which throws away the return value) is desugared to \( \text{let } x = \text{call } f(); \text{skip} \). As mentioned in Section 3.1, loops are translated to recursive functions in the usual way. Statements \( \text{break} \) and \( \text{continue} \) are translated to \( \text{ret} \) and \( \text{ret call} \) statements, respectively.

Named outputs are translated away by creating a global mutable variable for each of the named outputs. They are initialized to default values at definition time and after each \( \text{yield} \) statement. The \( \text{output} \) statement is translated to an assignment to a corresponding variable. At \( \text{yield} \), a tuple is yielded that consists of every named output in the order of definition. The example presented in Section 3.4 is desugared to the code in Figure 5a.

The underlying implementation mechanism for the “magic primes” feature – the ability to access an input value from the previous time step – is also mutable variables. For each input variable that occurs with a prime, a mutable variable is created, which holds the previous value of that input. It is assigned before each \( \text{yield} \). The Moore counter example presented at the end of Section 3.1 is desugared to the code presented in Figure 5b.

### 4.2 Lambda Lifting

The first step in the process of compiling the YieldFSM core language to a Mealy machine is lifting every function to the top level – lambda lifting [Johnsson 1985]. Unfortunately, this
transformation cannot be directly applied to YieldFSM automaton definitions because of the presence of mutable variables. They need to be handled differently than immutable variables introduced by let statements, because the changes of their values due to assignment statements need to be preserved across function calls. Example code is presented in Figure 6a.

Because functions in YieldFSM are first order, and non-tail recursion is not allowed, one possible solution would be to lift every mutable variable to the top level, making them global. These global variables would then need to be handled in some later phase of compilation. Instead, we opted for a different method. For every mutable variable defined outside of a function definition, following steps are performed:

- A local mutable variable definition is added at the beginning of the function’s body.

Fig. 5. Examples illustrating the desugaring pass.

Fig. 6. Handling mutable variables in lambda lifting.
\[ c ::= \begin{cases} p; s \\ e \mid \text{call } f \ e \end{cases} \]
\[ r ::= \begin{cases} \text{ret } r \\ \text{yield } e; s \\ \text{let } x = e; s \\ \text{let } x = \text{call } f \ e; \text{ret } r \\ \text{if } e; s \text{ else: } s \\ \text{case } e \ c \end{cases} \]
\[ s ::= \begin{cases} \text{ret call } f \ e \\ \text{yield } e; s \\ \text{let } x = e; s \\ \text{if } e; s \text{ else: } s \\ \text{case } e \ c \\ \text{fun } f \ p; s \\ \text{data } d \ a^* = (n \tau^*)^* \\ \text{fsm } x :: \tau \\ \text{input } p; F^* \\ \text{ret call } f \ e \\ \end{cases} \]

(a) The normalized language. (b) The lowered language.

Fig. 7. The YieldFSM normalized and lowered languages.

- The function definition is extended with an additional argument, which is used to initialize the local copy of the mutable variable.
- Each call of the considered function (including recursive calls) is modified so that the current value of the mutable variable is passed using the additional argument.
- Every \text{ret} statement inside the function’s body is modified to return a tuple of the original return value and the new value of the local mutable variable.
- After each non-tail call (\text{let call}) of the considered function, the caller’s mutable variable is updated with the new value.

The result of the described transformation applied on the example code is shown in Figure 6b. After this transformation, standard lambda lifting is applied (Figure 6c). The top level code is moved to a new function, which becomes an “entry point” of the resulting automaton definition.

4.3 Normalization

Next, the code is transformed to a form that is related to A-normal form [Flanagan et al. 1993]. In this form, statements can no longer be combined without restrictions using the semicolon operator. Instead they form decision trees, terminated with \text{ret} statements. Control flow follows the structure of abstract syntax trees of statements; the only branching forms are \text{if} and \text{case}. Mutable variables are no longer present; the transform presented in this section replaces them with immutable ones. Also, every non-tail call (\text{let call}) of the considered function is immediately followed by a \text{ret} statement. Additional constraints, which are not expressed by the grammar in Figure 7a:

\begin{itemize}
  \item \textbf{Constraint 1}. There is at most one \text{yield} on every branch of the decision tree. I.e. for every occurrence of the form \text{yield } e; s, there is no another occurrence of this form in s.
  \item \textbf{Constraint 2}. The input variables are not referenced after a \text{yield}. I.e. for every occurrence of the form \text{yield } e; s, input variables do not occur free in s.
\end{itemize}

These constraints are important for the transform presented in Section 4.5.

\footnote{In ANF, only expressions which directly produce values (e.g. variables or literals) are allowed as function arguments in let definitions. In YieldFSM normal form, this requirement is relaxed – entire Template Haskell expressions can be used. This does not introduce difficulties, as YieldFSM statements do not occur in TH expressions. The YieldFSM compiler treats TH expressions opaquely – they are not subject to processing (with the exception of some optimizations).}
Algorithm 1: The normalization algorithm.

Data: \texttt{fsm} \(x \:: \tau; \ input \ p_i; \ \tilde{F} \); \ ret call \( f_I \) \( e_I \) (lambda-lifted YieldFSM)
Result: \( \texttt{fsm} x \:: \tau; \ input \ p_i; \ \tilde{F}_N \); \ ret call \( f_I \) \( e_I \) (normalized YieldFSM)

function \( \texttt{simple}(s) \)
\( s = \text{ret} (x_1, \ldots, x_n) \) or \( s = \text{ret call} \ f (x_1, \ldots, x_n) \) for some \( f, x_1, \ldots, x_n \)

\( \tilde{F}_N := \emptyset \)

function \( C[s] \)
\( s = FV(s) \setminus FV(\tilde{F}) \)
let \( f = \text{fresh} \)
\( \tilde{F}_N := \tilde{F}_N \cup (\text{fun } f (x_1, \ldots, x_n): s) \)
call \( f (x_1, \ldots, x_n) \)

function \( N[s_0]_{s_k} \)
// \( s_k \) is in normalized form
match \( s_0 \)
\( | (\text{skip}) \rightarrow s_k \)
\( | (\text{ret } r) \rightarrow \text{ret } r \)
\( | (s_1; s_2) \rightarrow N[s_1]_{s_k} (N[s_2]_{s_k}) \)
\( | (\text{yield } e) \text{ when } FV(s_k) \cap BV(p_I) = \emptyset \text{ and } s_k \text{ does not contain } \text{yield} \rightarrow \)
\( \text{yield } e; s_k \)
\( | (\text{if } e: s_1 \text{ else: } s_2) \text{ when } \text{simple}(s_k) \rightarrow \)
\( \text{if } e: N[s_1]_{s_k} \text{ else: } N[s_2]_{s_k} \)
\( | (\text{case } e \mid p_1: s_1 \ldots) \text{ when } \text{simple}(s_k) \rightarrow \)
\( \text{case } e \mid p_1: N[s_1]_{s_k} \ldots \)
\( | (\text{let } x = e; s) \rightarrow \)
let \( x' = \text{fresh} \)
let \( x' = e; N[s[x := x']]_{s_k} \)
\( \text{match } s_0 \)
\( | (\text{let } x = \text{call } f e; s) \rightarrow \)
let \( x' = \text{fresh} \)
let \( x' = \text{call } f e; \text{ ret } C[N[s[x := x']]_{s_k}] \)
\( \text{match } s_0 \)
\( | (\text{var } x = e; s) \rightarrow \)
let \( x' = \text{fresh} \)
let \( x' = e; N[s[x := x']]_{s_k} \)
\( | (x = e) \rightarrow \)
let \( x' = \text{fresh} \)
let \( x' = e; s_k [x := x'] \)
\( \text{otherwise} \rightarrow N[s]_{s_k} (\text{ret } C[s_k]) \)

for (fun \( f p: s := \tilde{F} \) do
\( \tilde{F}_N := \tilde{F}_N \cup (\text{fun } f p: \ N[s]_{s_k} (\text{ret } ())) \)

The method that transforms YieldFSM programs to the normalized form is presented using pseudocode as Algorithm 1. The function \( N[s]_{s_k} \) is applied to the body of every function in the automaton description. As a side effect, this algorithm may create new function definitions with fresh names, by the use of the \( C[s] \) function. This is done to satisfy the constraints of the normal form (cases for \text{yield} and \text{let call}), or to avoid code duplication (cases for \text{if} and \text{case}). The roles of the constraints will be described in the following sections.
Generating Circuits with Generators

```
input i

data Cont_inc a = C a
fun inc ((j, n), c):
  if j:
    ret call ap_inc (c, n+1)
  else:
    ret call ap_inc (c, n)
    fun ap_inc (c, n_2):
      case c | C m:
        let n_2 = n + 1
        yield m
        ret call loop n_2
      else:
        let m = n
        case c | C m:
          let n_2 = n
          yield m
          ret call loop n_2
    ret call loop n
  ret call loop n_2

ret call loop θ
```

(a) Normalized example code.

```
input i

let j = not i
let m = n
let n_2 = call inc (j, n)
ret call loop_2 (m, n_2)

yield m
ret call loop n_2
ret call loop θ
```

(b) Lowered example code.

```
input i

let j = not i
let m = n
let n_2 = n
ret call inc ((j, n), C m)
ret call loop n
ret call loop_2 (m, n_2)
ret call loop_2 (m, n_2)
ret call loop θ
```

(c) No non-emitting transitions.

Fig. 8. Normalization and stack reification.

Mutable variables are translated away by `let`-defining a new “version” of the mutable variable every time an assignment statement is encountered by the algorithm. The approach works thanks to the previous transformation, as mutable variables in the lambda lifted form are local to function definitions, and the control flow in a single function definition forms a directed acyclic graph.

Figure 8a presents the program from Figure 6c transformed using the normalization algorithm. Some simplifications were applied for clarity of presentation:

- Some `let` definitions were removed by using substitution. Only definitions which renamed a variable (e.g. `let n = n_init`) or which were used only once were subject to this simplification.
- The `toplevel` function was removed, and the initial call was changed from `ret call toplevel ()` to `ret call loop θ` – as the body of `toplevel` consisted only of this call.

Both of these simplifications are automatically performed by the YieldFSM optimizer, which is described in Section 4.6.

4.4 Reifying the Call Stack

The next step in the compilation takes care of non-tail function calls. In this step, the implicit call stack is made explicit in the form of data types, which can then be represented as part of the state of the final automaton. The approach used in the YieldFSM compiler can be explained as a combined CPS translation [Plotkin 1975] and defunctionalization [Reynolds 1972] pass. The normalized YieldFSM form makes this task simpler, as for each non-tail call, the only possible continuations are an immediate return (applying current continuation) or a tail call to another function.

However, the usual approach of creating a single continuation ADT for each type of continuation [Bell et al. 1997] cannot be used because the type information required is not available in Template Haskell. Even if it could, it would sometimes lead to suboptimal results. Because ADTs in Clash are represented in hardware as tagged unions, the constructor with the largest representation...
dictates the size of the representation of the entire ADT. Therefore, if the program contains several groups of functions that never call each other, it is beneficial to create a separate ADT for the continuations that can occur in each of the groups.

To solve these issues, the transform described in this section uses control flow information in the form of a call graph. Let $F$ be the set of functions in the YieldFSM program – the elements of this set are vertices of the graphs defined below. Let $G_T$ be the tail call graph: an edge $⟨f, f'⟩$ is in $G_T$ if and only if the function $f$ tail calls $f'$. Similarly, $G_C$ is a graph of non-tail calls (let call).

Let $F_R ⊆ F$ be the set of returning functions, defined as the smallest set such that:

- If $f$ returns directly (using ret $e$), then $f ∈ F_R$.
- If $⟨f, f'⟩ ∈ G_T$ and $f' ∈ F_R$, then $f ∈ F_R$.

Each returning function is modified to accept an additional continuation parameter $c$.

Let us now partition the set of functions $F$ into strongly connected components (SCCs) of the tail call graph $G_T$; let us call this partition $S$, and the corresponding condensation graph$^6$ $G^S_T$. For each SCC $F_i ∈ S$ consisting of returning functions (i.e., $F_i ⊆ F_R$), a continuation ADT data $Cont_i$ and a corresponding “apply” function fun $apply_i(c, ν)$ is created. Then for each pair of SCCs $F_i, F_j$ such that $F_j ⊆ F_R$ and connected by an edge $⟨F_i, F_j⟩ ∈ G^S_T$ (i.e., there exists a tail call between them), a constructor $Cinj_{i,j}$ is added to $Cont_j$, which is used to inject continuations from $Cont_i$ into $Cont_j$. The corresponding condensation graph is constructed by contracting each strongly connected component of $G$ into a single vertex. A condensation graph of $G$ is constructed by contracting each strongly connected component of $G$ into a single vertex. A condensation graph is acyclic by construction.

Let us now partition the set of functions $F$ into strongly connected components (SCCs) of the tail call graph $G_T$; let us call this partition $S$, and the corresponding condensation graph$^6$ $G^S_T$. For each SCC $F_i ∈ S$ consisting of returning functions (i.e., $F_i ⊆ F_R$), a continuation ADT data $Cont_i$ and a corresponding “apply” function fun $apply_i(c, ν)$ is created. Then for each pair of SCCs $F_i, F_j$ such that $F_j ⊆ F_R$ and connected by an edge $⟨F_i, F_j⟩ ∈ G^S_T$ (i.e., there exists a tail call between them), a constructor $Cinj_{i,j}$ is added to $Cont_j$, which is used to inject continuations from $Cont_i$ into $Cont_j$. The corresponding condensation graph is constructed by contracting each strongly connected component of $G$ into a single vertex. A condensation graph of $G$ is constructed by contracting each strongly connected component of $G$ into a single vertex. A condensation graph is acyclic by construction.

Let’s now turn our attention to the non-tail calls – the graph $G_C$. For each edge $⟨f, f'⟩ ∈ G_C$, find the SCCs $F_i, F_j$ such that $f ∈ F_i$ and $f' ∈ F_j$. For every non-tail call between $f$ and $f'$, a new constructor $C$ is added to $Cont_j$. Consider such a call, which has the form let $x = call f' e; ret r$. Let $x_1 . . . x_n$ be the free variables of $r$ which are bound in the definition of $f$ and different from $x$ – the variable holding the result of the call. There are four possible cases, depending on what follows the call to $f'$:

1. A return (i.e., $r = e'$).
2. A call to a non-returning function (i.e., $r = call f'' e'$ and $f'' /∈ F_R$).
3. A call to a returning function in the same SCC as the caller (i.e., $r = call f'' e'$ and $f'' ∈ F_i$).
4. A call to a returning function in a different SCC from the caller (i.e., $r = call f'' e'$, $f'' ∈ F_k$ and $F_i ≠ F_k$).

In each of the cases other than 2, the call is replaced with ret call $f'$ ($e_′, C c x_1 . . . x_n$). In the case 2, the current continuation $c$ is discarded, and the call has the form ret call $f'$ ($e_′, C x_1 . . . x_n$). The apply_j(c, ν) function is then extended by a new clause:

1. $C c’ x_1 . . . x_n$: let $x = ν; ret call apply_i (c’, e’)$
2. $C x_1 . . . x_n$: let $x = ν; ret call f'' e’$
3. $C c’ x_1 . . . x_n$: let $x = ν; ret call f''’ (e’, c’)$
4. $C c’ x_1 . . . x_n$: let $x = ν; ret call f''’ (e’, Cinj_i_k c’)$

The YieldFSM program resulting from the transform presented in this section now does not have any returns nor non-tail calls. It is now in the lowered form, as presented in Figure 7b.

Figure 8b presents the results of the algorithm described above on our example (Figure 8a). Again, some simplifications – which consisted of substituting let-definitions – were applied for improving readability.

$^6$A condensation graph of $G$ is constructed by contracting each strongly connected component of $G$ into a single vertex. A condensation graph is acyclic by construction.
Algorithm 2: Eliminating non-emitting transitions.

Data: fsm x :: τ; input p_I; D; F; ret call f_I e_I (lowered YieldFSM)
Result: fsm x :: τ; input p_I; D; F_O; ret call f_I e_I (lowered YieldFSM)

function inlinable(fun f p: s)
  \( f \notin V \) and FV(s) \( \cap \) BV(p_I) = \( \emptyset \) and s does not contain yield

\( V := \emptyset \)
\( F_O := \emptyset \)

function \( \llbracket f \rrbracket \)
  if \( f \notin V \) then
    Find (fun f p: s) in \( F \)
    \( V := V \cup \{ f \} \)
    \( F_O := F_O \cup \{ fun f p: \llbracket s \rrbracket_F \} \)

function \( \llbracket s_0 \rrbracket_b \)
match \( s_0 \)
| \( \text{ret call } f \ e \rightarrow \)
  Find (fun f p: s) in \( F \)
  if not b or inlinable(fun f p: s) then
    case e | p: \( \llbracket s \rrbracket_b \)
  else
    \( \llbracket f \rrbracket \)
    \( \text{ret call } f \ e \)
| yield e; s \rightarrow yield e; \( \llbracket s \rrbracket_T \)
| let x = e; s \rightarrow let x = e; \( \llbracket s \rrbracket_b \)
| if e: s_1 else: s_2 \rightarrow if e: \( \llbracket s_1 \rrbracket_b \) else: \( \llbracket s_2 \rrbracket_b \)
| case e | p_1: s_1... → case e | p_1: \( \llbracket s_1 \rrbracket_b \)... 

4.5 Eliminating Non-emitting Transitions

The YieldFSM automaton description is now in a form which can almost be interpreted as a Mealy automaton. Each function can be interpreted as a set of states, while the tail calls represent the transition function. The yield on the control flow path to the tail call describes the output function. The constraint 1 introduced by the normalization algorithm in Section 4.3 guarantees that there is at most a single yield on a given path. Therefore the one requirement which is still not satisfied is that every transition yields an output. Satisfying this constraint is the goal of the transformation presented in this section.

The algorithm has similarities to removal of \( \varepsilon \)-transitions in a DFA. It will construct a new set of functions while depth-first traversing the tail call graph. On each path where no yield occured yet, tail calls are always inlined. Otherwise, inlining occurs when none of the following is true about the called function:

- It is currently being processed or was processed before by the algorithm. In this case, instead of inlining, a recursive call is created.
- It contains a yield. This preserves constraint 1.
- It references an input. This preserves constraint 2.
If the given call is not inlined and is not a recursive call, the algorithm continues starting from the called function. Algorithm 2 presents the described method using pseudocode. When applied to the example in Figure 8b, the result is as in Figure 8c. Simplifications were applied consisting of reducing case statements as described in Section 4.6.

4.6 Optimization

Before the final Clash code is constructed, it is beneficial to perform a number of optimizations. These can help reduce the number of states of the resulting automaton, the size of the binary representation of the state, and the complexity of the circuits representing the transition and output functions. In particular, they can reduce many kinds of “administrative” steps introduced by the previous transforms. Some optimizations which were found to be beneficial are:

- Propagating constant arguments across function calls. If some function argument is always given the same constant value, it can be substituted into the function definition.
- Argument deduplication. If some pair of function arguments are always given the same value, one of them is substituted in place of the other.
- Unused argument removal. If some argument is never referenced (e.g. because of some other optimization), it can be discarded.
- Constructor expression propagation. If some let definition consists only of variables, constants and constructor calls and does not reference inputs, it can be substituted into the let body.
- Constructor expression extraction. If some let definition starts with a constructor, but does not qualify as a constructor expression, its subexpressions which are not constructor expressions are extracted to separate let definitions. For example, let x = (a, b + 1) becomes let c = b + 1; let x = (a, c).
- Case statement reduction. If some constructor expression is immediately matched by a case (e.g. case Just a:) and a matching pattern can be statically found, the case statement is replaced by let statements.
- Eliminating unused constructors and data types. If previous optimizations made some constructor or data type (created in the transform described in Section 4.4) unused, they are removed.

A remark about constructor expressions: focusing on them is important because of the performance model of circuits. In software, calling a constructor involves allocating memory and copying data; their cost is usually constant (or amortized constant), but non-zero, and thus duplicating constructors by using substitution can increase time and memory usage and reduce sharing. While in hardware, a constructor corresponds basically to grouping of wires; therefore they are essentially zero-cost and can be duplicated without restrictions.

4.7 Extracting the Automaton Definition

To finally extract a Clash automaton definition, a number of final transforms has to be performed. First, yield statements are transformed so that their argument is always a variable. If this was not the case, a let statement is added. Then, the yield statements are pushed through the statement tree, up to the ret calls. This transformation is valid thanks to the constraint 2 from Section 4.3. Recall from Section 3.1 that statements after a yield occur in the next clock cycle; if the constraint 2 was not satisfied, this transform would “mix up” inputs from different clock cycles.

After the transform described in the previous paragraph, yield and ret call statements always occur together at the leaves of the statement trees. The yield specifies the output value, while the ret call describes the next state of the automaton.
Generating Circuits with Generators

\[
\begin{align*}
c & ::= | p : s \\
s & ::= \text{yield } e; \text{ ret call } tr (n e) \\
& | \text{let } x = e; \ s \\
& | \text{if } e: s \ \text{else: } s \\
& | \text{case } e \ c^* \\
D & ::= \text{data } d \ a^* = (n \tau^*)^* \\
A & ::= \text{fsm } x :: \tau \\
& \text{input } p; \ D^* \\
& \text{fun } tr \ st: \text{ case } st \ c^* \\
& \text{ret call } tr (n e)
\end{align*}
\]

Fig. 9. The final language.

We now create a new data type `State`, which will become the type of states in the final automaton definition. For each function in the YieldFSM program, we create a constructor of this data type. The final program contains only one function, `tr`, which accepts a single parameter — a value of the `State` type. This value is immediately matched with a `case` statement, whose clauses correspond to the function bodies of the original program.

The program is now in the form presented in Figure 9. It can be now easily translated to a Clash program consisting of a single `mealy` call. The `tr` function becomes the transfer and output function of the automaton, which takes the automaton’s input and previous state as arguments, and has a single clause for each constructor of the `State` type. The automaton’s initial state is the initial argument applied to `tr` (the production rule for `A`).

5 EVALUATION

In this section we evaluate how YieldFSM performs in solving problems typical in digital hardware design, in comparison to traditional RTL hardware descriptions. We focus our attention on two aspects of the solutions:

- Code size and clarity. We wish to assess the suitability of YieldFSM as an abstraction for solving real problems. Code clarity is obviously subjective; we try to look at it from the perspective of a programmer accustomed with functional programming and generators.
- Performance. We want to measure if using YieldFSM incurs performance costs when compared to RTL style. Please note that the performance of digital circuits is measured using different metrics than software. Execution time is replaced with \( f_{\text{max}} \) — the maximum clock frequency which guarantees that the circuit works correctly. Instead of memory usage, the amount of physical resources used is counted — transistors in case of silicon, or configurable logic elements\(^7\) in case of an FPGA. These metrics are measured using hardware synthesis tools.

5.1 YieldFSM Unit Tests

The YieldFSM package includes a number of unit tests, intended to catch bugs during the development process and monitor the size and performance of the compiled automata. These include a number of classical examples such as:

- Oscillators — automata whose outputs alternate between two possible values.

\(^7\)Logic elements on FPGAs built by different manufacturers have different physical construction and different names, which include: logic cells (LCs) in Lattice chips, adaptive logic modules (ALMs) in Intel chips and configurable logic blocks (CLBs) in Xilinx chips.
\begin{verbatim}
fsm counter :: Signal dom (Unsigned 4)
var i = 0
forever:
do:
yield i
i = i + 1
until i == maxBound
do:
yield i
i = i - 1
until i == 0
\end{verbatim}

(a) YieldFSM version.

\begin{verbatim}
counter :: Signal dom (Unsigned 4)
counter = moore f snd (False, 0) (pure ()
where
f (False, i) () = (j == maxBound, j)
where j = i + 1
f (True, i) () = (j /= 0, j)
where j = i - 1
\end{verbatim}

(b) Pure Clash version.

Fig. 10. Up-down counter in YieldFSM and pure Clash.

- Moore and Mealy counters.
- Counters that change their value every two cycles.
- Counters that count up and, after reaching the maximum value, start counting down instead of wrapping around.
- A simple traffic light controller.

These automata are implemented in many different ways to test various features of the YieldFSM language. While writing this paper, there were 47 such tests.

For almost all of them, the compilation results are equivalent to automata hand-written in Clash, and therefore have the same performance and circuit size. For the few of them that are not, analyzing the produced code suggests that this can be fixed by further improving the YieldFSM optimizer.

Figure 10 presents one of the examples from the test suite and an equivalent implementation using pure Clash. One can immediately notice that the YieldFSM version (Figure 10a) has more lines of code and has a more imperative feeling than the Clash version (Figure 10b). Yet, the Clash version is more involved. It has a helper variable \( j \), uses pairs and constants of type \( \text{Boolean} \), all of which are merely artifacts of the implementation. Also, understanding how the state of the automaton evolves requires some analysis. Whereas in the YieldFSM version the behavior of the automaton is immediately obvious to a person accustomed to programming with generators, and the code does not involve additional variables or types which are not important for the description of the problem. Both versions generate the same circuit and therefore have identical performance.

5.2 HD44780 Display Driver

The HD44780 [Hitachi 1998] is a popular and widely cloned chip used for driving character displays. It exposes a simple communication port, which is typically used with a microcontroller to show information on the display. Before the chip starts accepting normal commands, it needs to be initialized using a sequence of commands presented in the chip’s datasheet [Hitachi 1998, Fig. 23]. The initialization sequence is time-sensitive; a certain amount of microseconds or milliseconds must pass between the commands. The commands are sent using a parallel, 8-bit (or alternatively, 4-bit\(^8\)) interface.

One might want to use a HD44780 without a microcontroller, e.g. using an FPGA. The initialization sequence would then need to be generated using a control circuit in the form of a finite state machine. Then, the automaton needs to start accepting commands from the connected application circuit and

---

\(^8\)In the 4-bit mode the lower four bits are not used and each command is sent in two communication cycles.
transmit them to the HD44780. One difficulty is that, for simplicity of communication, the control circuit should be clocked using the same frequency as the application circuit, which is often much higher than the maximum frequency for sending commands to the display. The duration of various delays, which are counted in clock cycles, need to be different for different clock frequencies, as the timing requirements of the HD44780 are specified in physical units.

An example implementation of such a control circuit can be found on the OpenCores website [op den Brouw 2012]. It is written in VHDL in a style typical for describing finite state machines. The clock frequency is specified as a constant (\texttt{simulator\_timestep}), and the bit sizes and maximum values for timing counters are derived from it. Excluding comments, empty lines and non-essential functionality, the VHDL source is around 270 lines long, where around 220 of them describe the transition and output functions of the automaton.

Using YieldFSM, such a machine can be specified using code which is curiously alike to an equivalent microcontroller program written in a system programming language like C. Its behavior should be easy to understand for an embedded systems programmer. Excluding empty lines, the code for the controller is 147 lines long, where:

- 55 are data type definitions, which capture the communication format of the chip.
- 54 are the definitions of the application bus protocol and auxiliary function definitions written in Clash.
- 40 form the actual description of the circuit’s behavior, and are written using YieldFSM.

This is a 45% reduction in code size for the entire controller, and an 80% reduction for the automaton description, compared to the VHDL implementation.

The implementation described in this section can be found in the supplementary material, in the file \texttt{yieldfsm-hd44780/src/FSM/HD44780.hs}. This file actually contains four implementations, which implement various methods of communicating with the display: with 4-bit and 8-bit data bus, and with or without polling the “busy bit” (which is used by the HD44780 to signal the ability to accept new commands). The implementation used in the comparison above (8-bit data bus, no busy bit polling) is named \texttt{controller8bit} and is presented in Figure 11.

The controller circuit is connected to the display using the output \texttt{d}, which contains the signals necessary for driving it. It accepts commands from a data bus (input \texttt{bus}). Readiness to accept commands from the bus is signalled using the output \texttt{b}.

A curious feature of the YieldFSM/Clash implementation is in handling clock domains and their frequency. In Clash, the clock domains (which were briefly introduced in Section 2.3) have their clock period (in picoseconds) statically associated with them using a Haskell type family. This is used to derive the counter bit sizes and maximum values at compile time. Essentially, thanks to this feature of Clash, the YieldFSM implementation is clock-polymorphic. The use of this feature can be seen on Figure 11, lines 13, 15 and 17. The function \texttt{unsigned\_cycles\_ns} calculates the number of clock cycles in a given number of nanoseconds, which is used as the number of iterations of the \texttt{repeat1} loop.

### 5.3 Multicycle RISC-V CPU

RISC-V [Waterman and Asanović 2019] is an instruction set architecture created at the University of California. Because of its license terms and its modern, modular design, it has found much use in the industry, and has even more promise for future use [Greengard 2020]. Thanks to the simplicity of the core instruction set, it is used for teaching computer architectures and microprocessor design [Lowe-Power and Nitta 2019; Zekany et al. 2021].

One possible method of implementing RISC-V is called a multicycle microarchitecture. In such an implementation, the execution of a single instruction can last multiple clock cycles. This allows
```plaintext
fsm controller8bit :: forall dom period.
    (HiddenClockResetEnable dom,
     DomainPeriod dom - period,
     KnownNat period, 1 <= period)
    => Signal dom Bus_Input
    -> Signal dom (Data_Input 8, Bus_Output)

input bus
output d = empty_data :: Data_Input 8
output b = busy_bus

-- set RS (register set) and D (data), cycle EN
fun sendbyte (rs, d):
    repeat1 unsigned_cycles_ns (clockPeriod @dom) d50:
        yield<d> write_data rs d False
    repeat1 unsigned_cycles_ns (clockPeriod @dom) d250:
        yield<d> write_data rs d True
    repeat1 unsigned_cycles_ns (clockPeriod @dom) d200:
        yield<d> write_data rs d False

-- wait for us microseconds
fun delay us :: Unsigned (BitsFor 40000):
    repeat1 us:
        repeat1 unsigned_cycles_us (clockPeriod @dom) d1:
            yield

-- send to display, wait for us microseconds
fun sendDelay (us :: Unsigned (BitsFor 4100), rs, d):
    call sendbyte (rs, d)
    call delay (zeroExtend us)

-- wait 40 ms
call delay 40000

-- initialize display
call sendDelay (4100, Instr, pack $ Function F8bit F1line F5x8font)
call sendDelay (100, Instr, pack $ Function F8bit F1line F5x8font)
call sendDelay (100, Instr, pack $ Function F8bit F1line F5x8font)
call sendDelay (53, Instr, pack $ Function F8bit F2lines F5x8font)
call sendDelay (53, Instr, pack $ Display DOff DNoCursor DNoBlink)
call sendDelay (3000, Instr, pack $ Clear)
call sendDelay (53, Instr, pack $ EntryMode EIncrement ENoShift)
call sendDelay (53, Instr, pack $ Display DOn DNoCursor DNoBlink)

-- handle bus requests
forever:
    do:
        yield<b> idle_bus
    until bus_valid bus'
    call sendDelay (53, bus_rs bus', bus_data bus')
```

Fig. 11. HD44780 display driver in YieldFSM (8-bit, no busy bit polling).
reusing some parts of the CPU for different purposes. For example, the same adder circuit can be used for calculating instruction and data pointer addresses, performing addition instructions, and comparing values for conditional jumps (branches). Multicycle CPUs are usually implemented by splitting the design into two parts: the data path, which contains components related to performing calculations and storing data, and the control path, which contains an automaton whose role is configuring the data path depending on the instruction being executed.

In order to evaluate YieldFSM for the purpose of designing controllers for CPUs and other computing machines, three functionally equivalent implementations of RISC-V were implemented using Clash:

1. a “traditional”, low-level implementation, where both data and control paths are designed explicitly (at the register transfer level).
2. an implementation which uses the data path from the previous implementation, but replaces the controller with one designed in YieldFSM.
3. a high-level implementation, where both the control and data path are designed using a single YieldFSM source.

The three implementations share some code, including the instruction decoder, arithmetic logic unit, and register file, and differ only in the aspects described above.

Table 1 contains a comparison of the three cores. The following parameters were compared:

- LoC – lines of code (the shared code was not counted).
- LCs – used Logic Cells (of a Lattice ICE40 family FPGA).
- $f_{\text{max}}$ – maximum clock frequency (on a Lattice ICE40HX8K FPGA).

The last two parameters were obtained using Yosys and nextpnr [Shah et al. 2019].

The implementation (2) has similar size and performance to the traditional one. It is only a little shorter (around 6%), but arguably much more readable. The traditional controller is hard to understand from source code alone; a state diagram needs to be drawn, which normally would be included in the documentation of the design. While in the controller written in YieldFSM, the control flow can be readily deduced from the structure of the source code. However, understanding the meaning of various control signals used still requires the knowledge of the separately designed data path.

The last implementation (3) is the shortest and arguably the most readable of the three. The implementation has a feel similar to code written in C or other system programming language. Unfortunately, it performs somewhat worse than the previous two implementations: the results in Table 1 show around 27% size increase and around 15% reduction in maximum frequency, when compared with the implementation 1.

The reasons for this performance loss were investigated by manipulating the Clash output of the YieldFSM compiler. The results were unconclusive and varied with synthesis optimization options used. We were able to reduce the number of LCs used, but the reduction was modest – around 1%. We also observed an unexpected difference in performance between the code generated by the YieldFSM compiler via Template Haskell, and the same code pasted into a separate file. Therefore,
more investigation is needed to find reasons for the observed performance of the implementation 3. One of the issues discovered is discussed in the following section.

The implementations discussed above can be found in the supplementary material, in the directory yieldfsm-riscv/src/FSM/RiscV. The relevant files are:

- `ExplicitControl.hs` – the explicit control path (implementation 1).
- `ExplicitData.hs` – the explicit data path (implementations 1 and 2).
- `YieldControl.hs` – the control path written in YieldFSM (implementation 2).
- `YieldCtlData.hs` – the high-level implementation written in YieldFSM (implementation 3).

5.4 The Importance of State Representation

One of the possible reasons for reduced performance of certain circuits described in YieldFSM, in comparison to traditional implementations, is related to the way state is represented. We shall use a simple example to demonstrate the issue. It is presented in Figure 12. This YieldFSM program describes an automaton that counts either up and down and can be paused. A pulse on the first input pauses or restarts the counter, while a pulse on the second input changes the counter direction without resetting the value of the counter. Therefore the counter needs to remember four bits of information: two bits for the value, one bit for the direction, and one bit for the pause.

The following state representation is generated by the YieldFSM compiler:

```haskell
data State en1 v1 v2 en2 = SUp en1 v1 | SDn v2 en2
```

Recall that the YieldFSM compiler does not know the types of the expressions it is manipulating. Only when the Clash compiler takes over, it is inferred by the type checker that \( \text{en1} \) and \( \text{en2} \) must be `Bool`, and that \( \text{v1} \) and \( \text{v2} \) must be `Unsigned 2`.

In Clash, the definition of the datatype implies (by default) its representation in the synthesized circuit. The first few bits represent the constructor’s tag, while the arguments are packed next to
each other in the order of definition. The representation of the data type above is presented in Figure 13.

The location of the counter value is different depending on the counting direction. This is troublesome, as the synthesized circuit must therefore contain parts (e.g. multiplexers) that “move” the counter value when the counting direction changes, which increases the circuit’s size and decreases its maximum frequency. The enable bit also changes position, which requires a multiplexer to get its value. This issue can easily be fixed by changing the argument ordering of either \textit{up} or \textit{dn}. The performance figures for the unoptimal and fixed circuits are presented in Table 2.

This seemingly insignificant change yielded a factor of 2.5 improvement for the circuit size and a factor of 1.3 improvement for the maximum clock frequency. This illustrates well that in digital hardware design, data representation matters a lot.

Such an optimization could in principle be performed automatically. The representation of data types generated in the process of compilation could be optimized to minimize the size of the circuit, maximize the operating frequency, or find some compromise of the two. This would require data flow information, which describes the required connections, and the typing information, which gives the information about the size of the data that needs to be stored in the state. Unfortunately, to get access to the type information would require deeper integration with the type checker, which is not possible in Template Haskell.

### 6 CONCLUSION AND FUTURE WORK

In this paper, we set out to find a programming language abstraction which would allow descriptions of Mealy machines using high-level code. We argue that generators, characterized by their use of the \texttt{yield} statement, are well suited for this role. We make the following contributions:

- We introduce YieldFSM, the first hardware design language (HDL) in which the circuit’s behavior is described using generators. This raises the level of abstraction in comparison to traditional HDLs, while retaining the ability to synchronize to the circuit’s clock.
- We implement and describe the YieldFSM compiler, which translates programs written in the presented language to Mealy machines.
- We evaluate the YieldFSM language and compiler using examples of various complexity. We demonstrate the reduction in code size and improvement of code readability compared to other, more traditional approaches.

We believe that the YieldFSM language, even in its proof of concept stage, shows a lot of promise. We plan to improve it further, in order to make it more usable in practice. There are several avenues open for future work, which will be summarized below.

First and foremost is the issue of the implementation method, which consists of using Template Haskell to embed YieldFSM into Clash. While advantageous for development and experimentation, it leads to poor programmer experience – in particular, incomprehensible error messages. Because of the lack of type information, it also leads to missing important optimization opportunities, as discussed in Section 5.4. In a practical implementation, the automata description language should be deeply integrated into a hardware description language – which could be Clash or another HDL.

#### Table 2. Performance of the unoptimal and fixed counters (see Figure 12).

| Example  | LCs | \% | \(f_{\text{max}}\) | \% |
|----------|-----|----|---------------------|----|
| Unoptimal| 16  | 100% | 252 MHz             | 100% |
| Fixed    | 6   | 38%  | 323 MHz             | 128% |
Second, the first order nature of YieldFSM reduces abstraction opportunities. For example, loops are currently implemented as syntactic sugar, but in a more expressive language they could be implemented as higher-order functions. Variables could be implemented using a state monad, and other control effects could be similarly introduced. There are possibly more advantages to a higher order HDL with generators which are yet to be discovered. With the availability of type information, this could be implemented using defunctionalization [Bell et al. 1997] or normalization [Najd et al. 2016].

The yield instruction present in YieldFSM can be seen as a kind of computational effect [Plotkin and Power 2004]. It is worth investigating if the ideas of this paper can be fit into a language with general support for algebraic effects, e.g. in the style of [Biernacki et al. 2020]. A backend for this language that targets digital circuits, in the style of Clash, would have to be developed. YieldFSM could then be embedded in the form of a built-in effect handler. Such a language could be used as a platform for performing experiments on using computational effects to compose hardware designs.

Another possible line of work would be to give YieldFSM formal semantics, e.g. in coinductive big-step style [Nakata and Uustalu 2010]. This would allow one to formally prove correctness of the transforms presented in this paper.

The authors strongly believe that generators are an abstraction which is well suited for many problems in digital hardware design, and could help bridge the conceptual gap between designing hardware and writing software. YieldFSM is a small step in the continuing effort to bring hardware design to the masses.

DATA AVAILABILITY STATEMENT
The source codes of the YieldFSM compiler (Section 4), its unit tests (Section 5.1), the HD44780 display driver (Section 5.2), the multicycle RISC-V cores (Section 5.3), and various examples presented in the paper can be found in the reproduction package which accompanies this article [Materzok 2022].

REFERENCES
Christiaan Baaij, Matthijs Kooijman, Jan Kuper, Arjan Boeijink, and Marco Gerards. 2010. ClaSH: Structural Descriptions of Synchronous Hardware using Haskell. In Proceedings of the 13th Conference on Digital System Design (DSD 2010). IEEE, Lille, France, 714–721. https://doi.org/10.1109/DSD.2010.21

Christiaan Baaij and Jan Kuper. 2014. Using Rewriting to Synthesize Functional Languages to Digital Circuits. In Proceedings of the Fourteenth Symposium on Trends in Functional Programming (TFP 2014), Jay McCarthy (Ed.). Springer, Provo, UT, USA, 17–33. https://doi.org/10.1007/978-3-642-45340-3_2

Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: constructing hardware in a scala embedded language. In Design Automation Conference 2012 (DAC 2012). IEEE, San Francisco, CA, USA, 1212–1221. https://doi.org/10.1145/2228360.2228584

David Bacon, Rodric Rabbah, and Sunil Shukla. 2013. FPGA Programming for the Masses: The Programmability of FPGAs Must Improve If They Are to Be Part of Mainstream Computing. Queue 11, 2 (Feb. 2013), 40–52. https://doi.org/10.1145/2436696.2443836

Jeffrey M. Bell, Françoise Bellegarde, and James Hook. 1997. Type-driven defunctionalization. In Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming (SIGPLAN Notices, Vol. 32, No. 8), Mads Tofte (Ed.). ACM Press, Amsterdam, The Netherlands, 25–37. https://doi.org/10.1145/258948.258953

Dariusz Biernacki, Maciej Pioróg, Piotr Poliesiuk, and Filip Sieczkowski. 2020. Binders by Day, Labels by Night: Effect Instances via Lexically Scoped Handlers. In Proceedings of the Forty-Seventh Annual ACM Symposium on Principles of Programming Languages, Lars Birkedal (Ed.). ACM Press, Cascals, Portugal, 29 pages.

Andrew Canis, Jongsook Choi, Mark Aldham, Victor Zhang, Ahmed Kammooona, Jason H Anderson, Stephen Brown, and Tomasz Czajkowski. 2011. LegUp: high-level synthesis for FPGA-based processor/accelerator systems. In Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA 2011). ACM, New York, NY, USA, 33–36. https://doi.org/10.1145/1950413.1950423

Xavier Clerc and Arsenly Alekseyev. 2018. Hardcaml. Jane Street. Retrieved January 30, 2022 from https://github.com/ janestreet/hardcaml
Edsger W. Dijkstra. 1968. Go To Statement Considered Harmful. Commun. ACM 11, 3 (March 1968), 147–148. https://doi.org/10.1145/362929.362947

Cormac Flanagan, Aml Sabry, Bruce F. Dub, and Matthias Felleisen. 1993. The Essence of Compiling with Continuations. In Proceedings of the ACM SIGPLAN ’93 Conference on Programming Languages Design and Implementation (SIGPLAN Notices, Vol. 28, No 6), David W. Wall (Ed.). ACM Press, Albuquerque, New Mexico, 237–247. https://doi.org/10.1145/173262.155113

Mike Gordon. 1995. The semantic challenge of Verilog HDL. In Proceedings of the Tenth Annual IEEE Symposium on Logic in Computer Science, Andre Secevod (Ed.). IEEE Computer Society, San Diego, CA, USA, 136–145. https://doi.org/10.1109/LICS.1995.523251

Samuel Greengard. 2020. Will RISC-V revolutionize computing? Commun. ACM 63, 5 (April 2020), 30–32. https://doi.org/10.1145/3386377

John L. Hennessy and David A. Patterson. 2019. A New Golden Age for Computer Architecture. Commun. ACM 62, 2 (Jan. 2019), 48–60. https://doi.org/10.1145/3282307

Hitachi 1998. HD44780U (LCD-II) Dot Matrix Liquid Crystal Display Controller/Driver. Hitachi, Tokyo, Japan.

C.A.R. Hoare. 1975. Recursive Data Structures. International Journal of Computer and Information Sciences 4, 2 (1975), 105–132. https://doi.org/10.1007/BF00976239

Paul Hudak. 1996. Building Domain-Specific Embedded Languages. Comput. Surveys 28, 4 (Dec. 1996), 196. https://doi.org/10.1145/242224.242477

IEEE 2018. IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. IEEE, New York, NY, USA. https://doi.org/10.1109/IEEESTD.2018.8299595

IEEE 2019. IEEE Standard for VHDL Language Reference Manual. IEEE. https://doi.org/10.1109/IEEESTD.2019.8938196

Thomas Johnsson. 1985. Lambda Lifting: Transforming Programs to Recursive Equations. In Functional Programming Languages and Computer Architecture (Lecture Notes in Computer Science, 201), Jean-Pierre Jouannaud (Ed.). Springer-Verlag, Nancy, France, 190–203. https://doi.org/10.1007/3-540-15975-4_37

Barbara Liskov, Alan Snyder, Russell Atkinson, and Craig Schaffert. 1977. Abstraction Mechanisms in CLU. Commun. ACM 20, 8 (Aug. 1977), 564–576. https://doi.org/10.1145/359763.359789

Jason Lowe-Power and Christopher Nitta. 2019. The Davis In-Order (DINO) CPU: A Teaching-Focused RISC-V CPU Design. In Proceedings of the Workshop on Computer Architecture Education. ACM, Phoenix, AZ, USA, 8 pages. https://doi.org/10.1145/3338698.3338892

Geoffrey Mainland. 2007. Why It’s Nice to Be Quoted: Quasiquoting for Haskell. In Proceedings of the ACM SIGPLAN Workshop on Haskell (Haskell ’07). Association for Computing Machinery, Freiburg, Germany, 73–82. https://doi.org/10.1145/1291201.1291211

Marek Materzok. 2022. Reproduction Package for Article ‘Generating Circuits with Generators’. https://doi.org/10.5281/zenodo.6874025

George H. Mealy. 1955. A method for synthesizing sequential circuits. The Bell System Technical Journal 34, 5 (1955), 1045–1079. https://doi.org/10.1023/A:10583-7035.1955.5b0t3788.x

Don Mills and Clifford E Cummings. 1999. RTL coding styles that yield simulation and synthesis mismatches. In The 9th Annual Synopsys (& EPIC) Users Group Meeting (SNUG’99). San Jose, CA, USA, 16 pages.

Edward F. Moore. 1956. Gedanken-Experiments on Sequential Machines. In Automata Studies, Claude Shannon and John McCarthy (Eds.). Vol. 34. Princeton University Press, Princeton, NJ, 129–153. https://doi.org/10.1515/9781400882618-006

Shayan Najd, Sam Lindley, Josef Svenningsson, and Philip Wadler. 2016. Everything old is new again: quoted domain-specific languages. In Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, Martin Erwig and Tiark Rompf (Eds.). ACM Press, St. Petersburg, FL, USA, 25–36.

Keiko Nakata and Tarmo Uustalu. 2010. Resumptions, Weak Bisimilarity and Big-Step Semantics for While with Interactive I/O: An Exercise in Mixed Induction-Coinduction. In Proceedings of the Seventh Workshop on Structural Operational Semantics (SOS 2010) (EPTCS, Vol. 32), Luca Aceto and Pawel Sobocinski (Eds.). Paris, France, 57–75. https://doi.org/10.4204/EPTCS.32.5

Jesse op den Brouw. 2012. Simple HD44780 Driver. OpenCores. Retrieved Feb 8, 2022 from https://opencores.org/projects/hd44780_driver

Charles Papon. 2017. SpinalHDL: An alternative hardware description language. (Feb. 2017). Retrieved Feb 8, 2022 from https://archive.fosdem.org/2017/schedule/event/spinal_hdl/ Presented on the Free and Open Source Software Developers’ European Meeting in 2017 (FOSDEM 2017).

Christian Pilato and Fabrizio Ferrari. 2013. Bambu: A modular framework for the high level synthesis of memory-intensive applications. In 2013 23rd International Conference on Field programmable Logic and Applications (FPL 2013). IEEE, Porto, Portugal, 1–4. https://doi.org/10.1109/FPL.2013.6645550

Gordon Plotkin and John Power. 2004. Computational Effects and Operations: An Overview. Electronic Notes in Theoretical Computer Science 73 (Oct. 2004), 149–163.
Gordon D. Plotkin. 1975. Call-by-name, Call-by-value and the $\lambda$-calculus. *Theoretical Computer Science* 1 (1975), 125–159. [https://doi.org/10.1016/0304-3975(75)90017-1](https://doi.org/10.1016/0304-3975(75)90017-1)

John C. Reynolds. 1972. Definitional Interpreters for Higher-Order Programming Languages. In *Proceedings of 25th ACM National Conference*. Boston, Massachusetts, 717–740. [https://doi.org/10.1145/800194.805852](https://doi.org/10.1145/800194.805852)

Soujanna Sarkar, Shashank Dabral, Praveen K Tiwari, and Raj S Mitra. 2009. Lessons and experiences with high-level synthesis. *IEEE Design & Test of Computers* 26, 4 (2009), 34–45. [https://doi.org/10.1109/MDT.2009.84](https://doi.org/10.1109/MDT.2009.84)

Neil Schemenauer, Tim Peters, and Magnus Lie Hetland. 2001. Simple Generators. PEP 255. [https://www.python.org/dev/peps/pep-0255/](https://www.python.org/dev/peps/pep-0255/)

Ofer Shacham, Omid Azizi, Megan Wachs, Wajahat Qadeer, Zain Asgar, Kyle Kelley, John Stevenson, Stephen Richardson, Mark Horowitz, Benjamin Lee, Alexandre Solomatnikov, and Amin Firoozshahian. 2011. Rethinking Digital Design: Why Design Must Change. *Micro. IEEE* 30 (2011), 9–24. [https://doi.org/10.1109/MM.2010.81](https://doi.org/10.1109/MM.2010.81)

David Shah, Eddie Hung, Clifford Wolf, Serge Bazanski, Dan Gisselquist, and Miodrag Milanovic. 2019. Yosys+nextpnr: An Open Source Framework from Verilog to Bitstream for Commercial FPGAs. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 4 pages. [https://doi.org/10.1109/FCCM.2019.00010](https://doi.org/10.1109/FCCM.2019.00010)

Lenny Truong and Pat Hanrahan. 2019. A golden age of hardware description languages: Applying programming language techniques to improve design productivity. In *3rd Summit on Advances in Programming Languages (SNAPL 2019)*, Vol. 136. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, Providence, RI, USA, 1–21. [https://doi.org/10.4230/LIPIcs.SNAPL.2019.7](https://doi.org/10.4230/LIPIcs.SNAPL.2019.7)

Andrew Waterman and Krste Asanović (Eds.). 2019. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2019121*. RISC-V Foundation.

Stephen Zekany, Jielun Tan, James Connelly, and Ronald Dreslinski. 2021. RISC-V Reward: Building Out-of-Order Processors in a Computer Architecture Design Course with an Open-Source ISA. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*. ACM, New York, NY, USA, 1096–1102. [https://doi.org/10.1145/3408877.3432472](https://doi.org/10.1145/3408877.3432472)

Proc. ACM Program. Lang., Vol. 6, No. ICFP, Article 92. Publication date: August 2022.