Using High-Level Synthesis to model SystemVerilog procedural timing controls

Luca Ezio Pozzoni
Politecnico di Milano
Milan, Italy
lucaezio.pozzoni@polimi.it

Fabrizio Ferrandi
Politecnico di Milano
Milan, Italy
fabrizio.ferrandi@polimi.it

Loris Mendola
STMicroelectronics
Catania, Italy
loris.mendola@st.com

Alfio Antonino Palazzo
STMicroelectronics
Catania, Italy
alfio.palazzo@st.com

Francesco Pappalardo
STMicroelectronics
Catania, Italy
francesco.pappalardo@st.com

Abstract—**In modern SoC designs, digital components' development and verification processes often depend on the component's interactions with other digital and analog modules on the same die. While designers can rely on a wide range of tools and practices for validating fully-digital models, porting the same workflow to mixed models' development requires significant efforts from the designers. A common practice is to use Real Number Modeling techniques to generate HDL-based behavioral models of analog components to efficiently simulate mixed models using only event-based simulations rather than Analog Mixed Signals (AMS) simulations. However, some of these models' language features are not synthesizable with existing synthesis tools, requiring additional efforts from the designers to generate post-tapeout prototypes. This paper presents a methodology for transforming some non-synthesizable SystemVerilog language features related to timing controls into functionally-equivalent synthesizable Verilog constructs. The resulting synthesizable models replicate their respective RNMs' behavior while explicitly managing delay controls and event expressions. The RNMs are first transformed using the MLIR framework and then synthesized with open-source HLS tools to obtain FPGA-synthesizable Verilog models.**

I. Introduction

Mixed-signal circuits integrate both analog and digital circuitry on the same semiconductor die. Although increasingly common in modern electronics, their design and verification flow still pose challenges. The typical design and verification flow starts from the definition of specifications and high-level requirements for the application. Then, while the analog components of the chip are developed, digital designers implement the digital portions of the circuit using hardware description languages (HDLs) like Verilog or VHDL.

While being developed, digital models are continuously tested using standard methodologies like the universal verification methodology (UVM) [1] to validate the correctness of their behavior. To test the correctness of digital components while analog devices are still in development, it is common practice to create Real Number Models (RNMs) of analog devices using the SystemVerilog language. These models' compatibility with event-based simulators provides access to the well-formalized verification metrics and approaches used in digital models' design. Furthermore, they significantly accelerate simulation times when compared to SPICE-based mixed-signal simulations.

After the design has been fully verified and its netlist generated, the time before the availability of the first production chips is used to prepare boards, instruments, and tests for testing the chips upon their arrival. Additionally, it may be necessary to analyze the design's performance beyond what is possible in simulation.

It is common practice to generate hardware prototypes of the design to fulfill these tasks. Mapping the digital portion into a hardware emulation board [2] or an FPGA is straightforward. Problems arise when mapping analog parts of the design into hardware. Sometimes it is possible to reuse analog devices produced for other products, but in general, developers need to hand-build a synthesizable model replicating RNMs behavior.

Doing so can become a burden since developers have to create and re-validate synthesizable models that replicate non-synthesizable SystemVerilog language features commonly used in RNMs.

This paper presents a methodology for implementing procedural timing controls using only synthesizable language features. The presented patterns can be applied to RNMs without forcing designers to use specific coding structures in their models. They can serve as a guideline for manual RNM rewriting or be generated automatically by translation tools. Furthermore, they can operate in emulation environments using fixed or variable time steps.

The presented modeling techniques' implementation uses MLIR [3], a compiler infrastructure for defining, manipulating, and mixing different intermediate representations, to generate the patterns modeling timing controls. Models are then translated into LLVM-IR, becoming compatible with the front end of High-Level Synthesis (HLS) tools like Bambu [4], an open-source HLS framework.

The main contributions of this paper are the definition of synthesizable models for representing SystemVerilog timing controls and their applications in an HLS-based toolchain for supporting fast prototyping of mixed-signal models.

The paper proceeds as follows: Section II introduces the related work; Section III discusses the details of the proposed time controls' implementations. We present results over a case study in Section IV, and Section V draws conclusions and outlines future research directions.

II. State of the Art

RNMs have proven to be a good tool for accelerating analog-mixed signals models' co-simulations thanks to their compatibility with event-based simulators. However, their definition is often a complex task. To minimize designers' efforts and potential for human mistakes, multiple works [5]–[7] defined methodologies and parametric models' templates to generate RNMs. Others [8] built a template library of HDL-defined analog components covering different circuits' classes to model complex designs' behavior. These approaches help designers to model analog circuits' continuous-time behaviors using the discrete-time model provided by HDLs. Furthermore, they often rely heavily on non-synthesizable language features to model analog circuits' time-dependent effects when present.

As for fully-digital designs' verification, emulation platforms can offer substantial speedups to the verification flow of mixed-signal models. The work presented in [9] takes advantage of this speedup by implementing a pragma-based conversion tool that generates synthesizable models. Following designers' annotations, it converts analog real-typed signals to a fixed-point representation. Like the results obtained from our toolchain, the resulting models are generated starting from validated RNMs and rely on oversampling input signals to replicate the analog models' behavior. However, this approach can only model linear circuits because of the timing requirements needed to preserve compatibility with digital components in emulation.

Other works [10] implement support for non-linear models following a different approach: synthesizable models' generation starts from a library-based block representation of analog circuits. Each block has a synthesizable implementation with linear or quadratic piecewise descriptions. This property guarantees that chains of blocks will compute a result within a given time step that determines the maximum sampling rate of the model. Further improvements in emulation performance have been explored by [11] using variable time steps to overcome the tradeoff between the emulator's throughput and the represented time resolution. The toolchain used for that approach uses designer-provided mathematical models of analog circuits to synthesize them with piecewise linear approximations. It relies on a dedicated controller to manage time steps and minimize analog models' evaluations. Like the previously presented one, this approach uses newly defined representations of analog circuits rather than reusing validated RNMs. Therefore, their integration into existing development flows requires additional efforts from the designers and potentially introduces another entry point for human-generated errors.

Instead, our approach replicates RNMs' behaviors according to the SystemVerilog Language Reference Manual (LRM) [12] specification and relies on HLS tools to generate synthesizable models that are backward-compatible with existing simulation testbenches, enabling the verification of functional equivalence between the toolchain results and their RNM counterparts.

III. Methodology

While low-level models describe analog components' behavior in continuous time, RNMs target event-based discrete-time simulations. Event-based simulators allow the designer to set a time granularity for the simulation through the `timeunit` and `timeprecision` parameters. These determine the frequency at which digital components are driven and the minimum time step modeled in the simulation.

RNMs describe analog behaviors using many non-synthesizable HDLs' features like real-valued data types (LRM 6.12), system tasks and functions (LRM 20), and procedural timing controls (LRM 9.4) among others. Commercial simulators implement these features according to the LRM specification, while synthesis tools do not since there is no direct mapping of their functionality in digital devices.

We developed an automatic toolchain for modeling combinatorial RNMs' behavior with an LLVM-IR representation. This representation was chosen because it can be used as a specification from HLS tools like [4] and [13] to generate synthesizable Verilog models reproducing it.

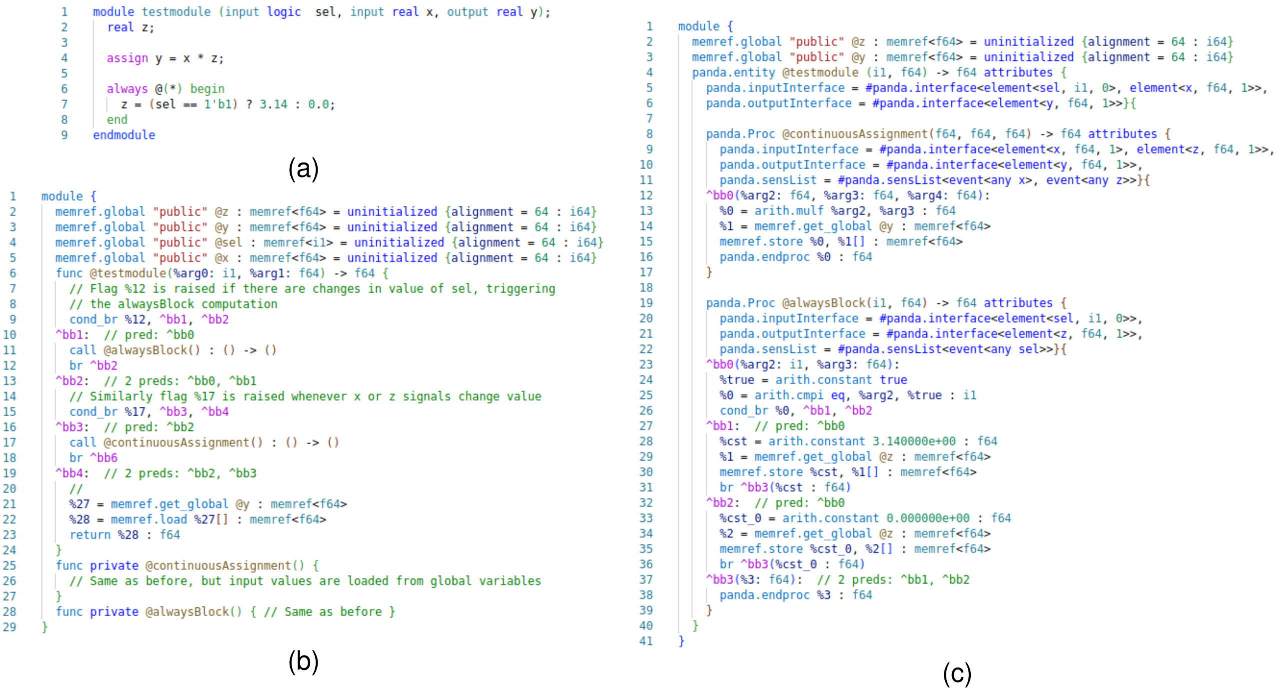
The translation process from SystemVerilog-based RNMs to their LLVM-IR-based representation implementation is based on MLIR. This tool's language consists of operations grouped under "dialects" that describe computations using different levels of abstraction. MLIR also provides a pass infrastructure for translating operations belonging to high-level dialects into equivalent operations or patterns of operations using lower-level dialects. Furthermore, MLIR-based models described using low-level dialects can be used to produce LLVM-IR models using a provided translation tool.

Section III-A briefly describes the MLIR operations and transformations we developed for modeling combinatorial RNMs using MLIR and extracting an equivalent LLVM-IR representation from them. Sections III-B and III-C present the MLIR operations and transformations developed to model procedural timing controls within the same toolchain, achieving compatibility with a significantly broader array of existing RNMs.

A. Automated Synthesis of Combinatorial RNMs

Many SystemVerilog language features describe operations and data types common to most imperative programming languages and can be directly mapped to low-level MLIR operations and data types. In the case of combinatorial RNMs, this holds true for most statements contained in processes' bodies.

Other language features' (continuous assignments, always blocks, entities, etc.) semantics model HDL-specific behaviors, and the MLIR built-in dialects do not provide operation modeling them. We therefore defined dedicated operations grouped under a new dialect to model their semantics in MLIR. Operations modeling entities replicate the original entities' interface, and their body contains an unordered collection of operations modeling continuous assignments and always blocks. These



```
module testmodule (input logic  sel, input real x, output real y);
  real z;

  assign y = x * z;

  always @(*) begin
    z = (sel == 1'b1) ? 3.14 : 0.0;
  end
endmodule
```

(a)

```
module {
  memref.global "public" @z : memref<f64> = uninitialized {alignment = 64 : i64}
  memref.global "public" @y : memref<f64> = uninitialized {alignment = 64 : i64}
  memref.global "public" @sel : memref<i1> = uninitialized {alignment = 64 : i64}
  memref.global "public" @x : memref<f64> = uninitialized {alignment = 64 : i64}
  func @testmodule(%arg0: i1, %arg1: f64) -> f64 {
    // Flag %12 is raised if there are changes in value of sel, triggering
    // the alwaysBlock computation
    cond_br %12, ^bb1, ^bb2
  ^bb1:  // pred: ^bb0
    call @alwaysBlock() : () -> ()
    br ^bb2
  ^bb2:  // 2 preds: ^bb0, ^bb1
    // Similarly flag %17 is raised whenever x or z signals change value
    cond_br %17, ^bb3, ^bb4
  ^bb3:  // pred: ^bb2
    call @continuousAssignment() : () -> ()
    br ^bb6
  ^bb4:  // 2 preds: ^bb2, ^bb3
    //
    %27 = memref.get_global @y : memref<f64>
    %28 = memref.load %27[] : memref<f64>
    return %28 : f64
  }
  func private @continuousAssignment() {
    // Same as before, but input values are loaded from global variables
  }
  func private @alwaysBlock() { // Same as before }
}
```

(b)

```
module {
  memref.global "public" @z : memref<f64> = uninitialized {alignment = 64 : i64}
  memref.global "public" @y : memref<f64> = uninitialized {alignment = 64 : i64}
  panda.entity @testmodule (i1, f64) -> f64 attributes {
    panda.inputInterface = #panda.interface<element<sel, i1, 0>, element<x, f64, 1>>,
    panda.outputInterface = #panda.interface<element<y, f64, 1>>}{

    panda.Proc @continuousAssignment(f64, f64, f64) -> f64 attributes {
      panda.inputInterface = #panda.interface<element<x, f64, 1>, element<z, f64, 1>>,
      panda.outputInterface = #panda.interface<element<y, f64, 1>>,
      panda.sensList = #panda.sensList<event<any x>, event<any z>>}{
    ^bb0(%arg2: f64, %arg3: f64, %arg4: f64):
      %0 = arith.mulf %arg2, %arg3 : f64
      %1 = memref.get_global @y : memref<f64>
      memref.store %0, %1[] : memref<f64>
      panda.endproc %0 : f64
    }

    panda.Proc @alwaysBlock(i1, f64) -> f64 attributes {
      panda.inputInterface = #panda.interface<element<sel, i1, 0>>,
      panda.outputInterface = #panda.interface<element<z, f64, 1>>,
      panda.sensList = #panda.sensList<event<any sel>>}{
    ^bb0(%arg2: i1, %arg3: f64):
      %true = arith.constant true
      %0 = arith.cmpi eq, %arg2, %true : i1
      cond_br %0, ^bb1, ^bb2
    ^bb1:  // pred: ^bb0
      %cst = arith.constant 3.140000e+00 : f64
      %1 = memref.get_global @z : memref<f64>
      memref.store %cst, %1[] : memref<f64>
      br ^bb3(%cst : f64)
    ^bb2:  // pred: ^bb0
      %cst_0 = arith.constant 0.000000e+00 : f64
      %2 = memref.get_global @z : memref<f64>
      memref.store %cst_0, %2[] : memref<f64>
      br ^bb3(%cst_0 : f64)
    ^bb3(%3: f64):  // 2 preds: ^bb1, ^bb2
      panda.endproc %3 : f64
    }
  }
}
```

(c)

Fig. 1. A very simple combinatorial RNM (a) is parsed and represented within MLIR (c) using a mix of built-in operations for the processes bodies and newly-defined operations. The resulting model can be further transformed to obtain an MLIR representation (b) that can be translated into LLVM-IR.

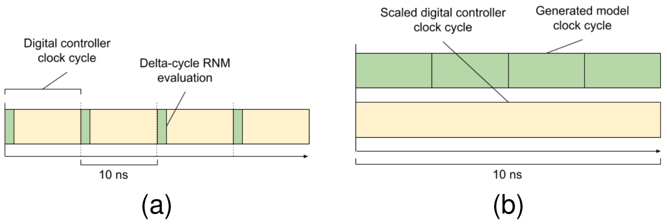

Digital controller clock cycle
Delta-cycle RNM evaluation
10 ns
(a)
Scaled digital controller clock cycle
Generated model clock cycle
10 ns
(b)

Fig. 2. Fig. (a) shows the time interactions between a combinatorial RNM, evaluated within a δ-cycle when its inputs are updated, and its digital controller in simulation. Fig. (b) shows the HLS-generated model latency and the time scaling required to hide it from the digital component. If the HLS-generated model requires 4 FPGA clock cycles to compute, the digital controller's clock frequency must be scaled by a factor 4 to let it sample the generated model's results as if they were computed instantly.

in turn describe their counterparts' interfaces, as well as their sensitivity lists, and their body contains the MLIR translation of the statements present in their counterparts.

The inputs of the toolchain are simulation-validated RNMs (Fig. 1a) parsed with the help of sv-lang [14] to extract their Abstract Syntax Trees. Then, a second parser developed for this toolchain analyzes the trees and maps each element into the most appropriate MLIR operation, building an equivalent representation (Fig. 1c) that uses a mix of MLIR operations coming from different dialects, only a portion of which is compatible with the transformations required to generate an LLVM-IR model.

The operations grouped under our custom dialect are not supported by any of the MLIR's built-in transformations, so it was necessary to define new transformations to replace them with operations supported by the translation mechanism to LLVM-IR. Operations defined to model entities are translated into MLIR functions which call procedure will be described later in this section. Within these functions, operations modeling processes and assignments are ordered according to their data dependencies. To remove each one of these operations, two functions are created: the first one evaluates the block's sensitivity list, and the second contains the original block's body. The resulting model no longer needs to rely on a simulator to determine which blocks must be executed but determines it autonomously at runtime.

The MLIR model resulting from these transformations (Fig. 1b) preserves all blocks' bodies and schedules their execution according to a static, serial order. Furthermore, being composed only of MLIR operations for which an LLVM-IR translation is provided, the model can be translated into LLVM-IR. It is then possible to use SW-oriented optimization tools like `llvm-opt` to remove dead and duplicated code from the model.

Finally, the serial model generated through these MLIR transformations is used as a basis for generating an equivalent synthesizable Verilog representation using HLS tools [4], [13] accepting LLVM-IR models as inputs. The synthesis process produces fully-digital Verilog models using a Finite State Machine (FSM) controller to drive their computation. Therefore, the synthesizable model replicating a combinatorial RNM's behavior can perform the same computation, but doing so requires it several clock cycles rather than a single δ-cycle.

Fig. 2 illustrates the time scaling that must be adopted to allow toolchain-generated models to interact with digital components after synthesis. The time scaling applied to the original digital component's clock must be long enough that the generated model's FSM will conclude its computation before its results are sampled by the digital component. To obtain a correct result in emulation, components cannot rely on the simulator's time management but need to adopt a common time representation within the emulation environment that hide the additional latency introduced by the toolchain from the digital components.

Toolchain-generated models are evaluated at fixed time intervals, which length is driven by the smallest time step that would be modeled in simulation. In the case of a combinatorial RNM which inputs are driven by a digital component, the smallest time step that must be modeled corresponds to the digital component's clock frequency. The synthesizable modules generated by the toolchain maintain the original RNMs' interfaces with additional signals for controlling the beginning and completion of the computation to be easily connected with digital components. Differently from other approaches, this toolchain does not aim at modeling analog components through a classification based on their behavior, but rather models SystemVerilog language features commonly used in RNMs and transforms them to obtain a synthesizable model that replicates its original RNM computation, regardless of the analog component that it models.

B. Modeling Procedural Timing Controls

Analog components' behaviors are typically related to time, either because they model an analog signal's fluctuations or because they model the presence of transport and inertial delays on digital signals. Consequently, RNMs of these components use SystemVerilog's procedural timing controls extensively to replicate them.

Procedural timing controls, similarly to always blocks and continuous assignments, are language features peculiar to HDLs and lack corresponding operations to which they could be mapped in MLIR. Therefore, we defined new MLIR operations under the same dialect used for always blocks. As before, the use of these new operations prevents the MLIR representation to be translated to LLVM-IR, so additional transformations for modeling the same behavior using only LLVM-IR compatible operations have to be defined as well.

Most delay and event controls (LRM 9.4.1 to 9.4.4) model similar behaviors: upon encountering them, the execution of the process is interrupted until a given time delay has passed or an event occurs. As mentioned in the previous subsection, the time model required to achieve compatibility between digital components and toolchain-generated components relies on slowing down the digital model's time notion by a factor big enough to hide the introduced latency from the digital components. Scaling the digital clock frequency to make it slower than the worst-case computation of a single δ-cycle was enough for combinatorial models, where the smallest time step to model is the clock signal driving the digital component. Timing controls require the modeling of fine-grain time steps, resulting in larger scaling factors for the digital components.

Fig. 3 shows the pattern applied by transformations to model delays without using custom-defined operations. The function (a previously-transformed process operation) is replaced by two new functions 3b: one with the operations preceding the timing control operation and one with those following the timing control. To determine which one should execute in a time step, an additional auxiliary function modeling the required delay or event detection is generated and scheduled.

The results of operations performed before the timing control should be visible to other processes even if they are executed in the same time step in which the timing control interrupts its process execution. To replicate this behavior, the function containing operations preceding the timing control also stores all the computed values in global variables, where they can be read and used by other processes within the same time step. In the same way, when a timing event has passed and an interrupted function can resume its execution, it must be made aware of changes in values caused by other functions while it was interrupted. Therefore, rather than fetching the signals' values sampled before the execution was interrupted, it loads signals' values from global variables containing updated values before executing the remaining operations.

When applied to blocking assignments, intra-assignment timing controls (LRM 9.4.5) present a similar behavior to other procedural timing controls. The only significant difference is the following: the evaluation of the assignment's expression happens before the function's execution is interrupted, while the assignment is performed after the function's execution resumes using the previously computed value. Modeling this behavior is simply a matter of defining another global variable to pass the assignment's value between the two functions.

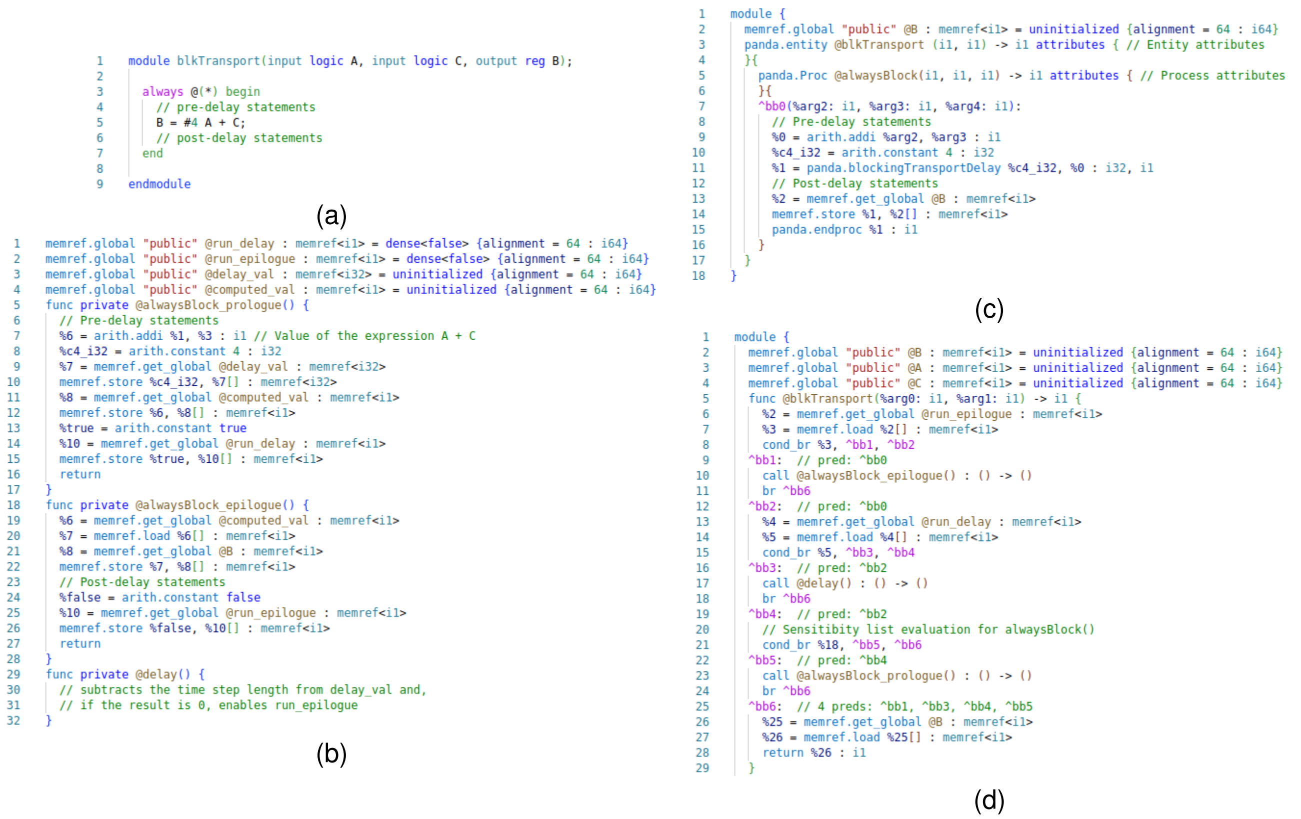
C. Intra-Assignment Events Controls

Differently from the timing controls described in the previous section, intra-assignment timing controls applied to non-blocking assignments do not interrupt the process execution. Therefore, their implementation must rely on a dedicated pattern (Fig. 4) different from the already presented one.

According to the LRM specification, non-blocking assignments' values are updated at the end of a time step evaluation. Instead, in the implemented pattern, every time the execution flow encounters a delayed non-blocking assignment the expression's value is pushed into a globally visible buffer together with its delay. This buffer contains the list of updates scheduled for execution at runtime and needs to be managed by a dedicated routine to update its content.

Each buffer (Fig. 4b) has an update routine responsible for updating all delay counters as time progresses and applying the previously-evaluated values at the correct time step. Buffer-updating routines execute at the beginning of each time step to provide each process with updated values.

Minimizing the buffer size without introducing synthesis-related errors in the emulation is crucial for optimizing the area occupation of the prototype. The SystemVerilog LRM poses no bound on the number of schedulable update events for a given non-blocking assignment. While performing static analysis of the RNM could provide upper bounds for the buffer dimensions, we chose to rely on designers' annotations to determine the buffer size. Annotating RNMs for buffer dimensioning can be considered a minimal manual intervention



```
1   module blkTransport(input logic A, input logic C, output reg B);
2
3     always @(*) begin
4       // pre-delay statements
5       B = #4 A + C;
6       // post-delay statements
7     end
8
9   endmodule
```

(a)

```
 1  memref.global "public" @run_delay : memref<i1> = dense<false> {alignment = 64 : i64}
 2  memref.global "public" @run_epilogue : memref<i1> = dense<false> {alignment = 64 : i64}
 3  memref.global "public" @delay_val : memref<i32> = uninitialized {alignment = 64 : i64}
 4  memref.global "public" @computed_val : memref<i1> = uninitialized {alignment = 64 : i64}
 5  func private @alwaysBlock_prologue() {
 6    // Pre-delay statements
 7    %6 = arith.addi %1, %3 : i1 // Value of the expression A + C
 8    %c4_i32 = arith.constant 4 : i32
 9    %7 = memref.get_global @delay_val : memref<i32>
10    memref.store %c4_i32, %7[] : memref<i32>
11    %8 = memref.get_global @computed_val : memref<i1>
12    memref.store %6, %8[] : memref<i1>
13    %true = arith.constant true
14    %10 = memref.get_global @run_delay : memref<i1>
15    memref.store %true, %10[] : memref<i1>
16    return
17  }
18  func private @alwaysBlock_epilogue() {
19    %6 = memref.get_global @computed_val : memref<i1>
20    %7 = memref.load %6[] : memref<i1>
21    %8 = memref.get_global @B : memref<i1>
22    memref.store %7, %8[] : memref<i1>
23    // Post-delay statements
24    %false = arith.constant false
25    %10 = memref.get_global @run_epilogue : memref<i1>
26    memref.store %false, %10[] : memref<i1>
27    return
28  }
29  func private @delay() {
30    // subtracts the time step length from delay_val and,
31    // if the result is 0, enables run_epilogue
32  }
```

(b)

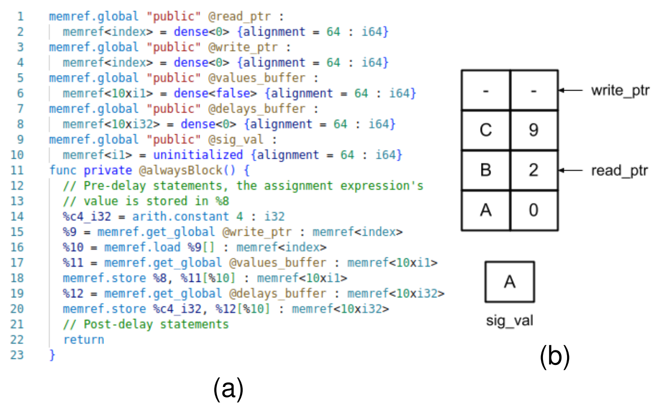
```
 1  module {
 2    memref.global "public" @B : memref<i1> = uninitialized {alignment = 64 : i64}
 3    panda.entity @blkTransport (i1, i1) -> i1 attributes { // Entity attributes
 4    }{
 5      panda.Proc @alwaysBlock(i1, i1, i1) -> i1 attributes { // Process attributes
 6      }{
 7      ^bb0(%arg2: i1, %arg3: i1, %arg4: i1):
 8        // Pre-delay statements
 9        %0 = arith.addi %arg2, %arg3 : i1
10        %c4_i32 = arith.constant 4 : i32
11        %1 = panda.blockingTransportDelay %c4_i32, %0 : i32, i1
12        // Post-delay statements
13        %2 = memref.get_global @B : memref<i1>
14        memref.store %1, %2[] : memref<i1>
15        panda.endproc %1 : i1
16      }
17    }
18  }
```

(c)

```
 1  module {
 2    memref.global "public" @B : memref<i1> = uninitialized {alignment = 64 : i64}
 3    memref.global "public" @A : memref<i1> = uninitialized {alignment = 64 : i64}
 4    memref.global "public" @C : memref<i1> = uninitialized {alignment = 64 : i64}
 5    func @blkTransport(%arg0: i1, %arg1: i1) -> i1 {
 6      %2 = memref.get_global @run_epilogue : memref<i1>
 7      %3 = memref.load %2[] : memref<i1>
 8      cond_br %3, ^bb1, ^bb2
 9    ^bb1:  // pred: ^bb0
10      call @alwaysBlock_epilogue() : () -> ()
11      br ^bb6
12    ^bb2:  // pred: ^bb0
13      %4 = memref.get_global @run_delay : memref<i1>
14      %5 = memref.load %4[] : memref<i1>
15      cond_br %5, ^bb3, ^bb4
16    ^bb3:  // pred: ^bb2
17      call @delay() : () -> ()
18      br ^bb6
19    ^bb4:  // pred: ^bb2
20      // Sensitibity list evaluation for alwaysBlock()
21      cond_br %18, ^bb5, ^bb6
22    ^bb5:  // pred: ^bb4
23      call @alwaysBlock_prologue() : () -> ()
24      br ^bb6
25    ^bb6:  // 4 preds: ^bb1, ^bb3, ^bb4, ^bb5
26      %25 = memref.get_global @B : memref<i1>
27      %26 = memref.load %25[] : memref<i1>
28      return %26 : i1
29    }
```

(d)

Fig. 3. A simple RNM involving a delay (a) is translated into MLIR (c) modeling the delay using a newly-defined operation (line 14). After the conversion from processes to functions, the delay operation is removed splitting the function into two (b) and adding an auxiliary function for counting the delay passing. Only one of the generated functions must execute in a time step, so the function controlling processes execution is updated accordingly (d)



```
memref.global "public" @read_ptr :
  memref<index> = dense<0> {alignment = 64 : i64}
memref.global "public" @write_ptr :
  memref<index> = dense<0> {alignment = 64 : i64}
memref.global "public" @values_buffer :
  memref<10xi1> = dense<false> {alignment = 64 : i64}
memref.global "public" @delays_buffer :
  memref<10xi32> = dense<0> {alignment = 64 : i64}
memref.global "public" @sig_val :
  memref<i1> = uninitialized {alignment = 64 : i64}
func private @alwaysBlock() {
  // Pre-delay statements, the assignment expression's
  // value is stored in %8
  %c4_i32 = arith.constant 4 : i32
  %9 = memref.get_global @write_ptr : memref<index>
  %10 = memref.load %9[] : memref<index>
  %11 = memref.get_global @values_buffer : memref<10xi1>
  memref.store %8, %11[%10] : memref<10xi1>
  %12 = memref.get_global @delays_buffer : memref<10xi32>
  memref.store %c4_i32, %12[%10] : memref<10xi32>
  // Post-delay statements
  return
}
```

(a)

-	-
C	9
B	2
A	0

(b)

Fig. 4. Non-blocking intra-assignment delays are modeled using a dedicated buffer (b) storing delayed values paired with the specified delay amount. At the beginning of each time step, the content of the sig_val register is updated by a dedicated routine if the delay pointed by the read_ptr reaches 0. The operation modeling the delay is translated into operations pushing new pairs of values in the buffer (a)

by the designers, who possess the best knowledge about the models' runtime behaviors.

The presented patterns can autonomously emulate some common SystemVerilog language features used in RNMs. Overall, they compose a minimal set of hardware controllers for modeling procedural timing controls without synthesizing a full-fledged emulation engine.

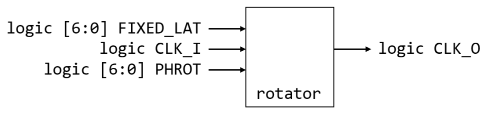


Fig. 5. The rotator's interface is composed only of digital signals. Regardless, the model's non-synthesizability comes from the use of procedural timing controls to model its programmable propagation delay.

IV. Case Study

We measure the performance obtainable by these methodologies by generating a synthesizable model starting from the RNM of a programmable phase rotator. This small, simple component is often used within larger models, for example to model analog components used to synchronize signals over the different lines of an ethernet port. Although simple, its description relies on all the language features modeled in the previous section like event controls and intra-assignment delays.

The model's interface (Fig. 5) allows the digital controller to define fixed and variable latencies to the input clock to control the output clock's phase rotation with a resolution of 1/128th of the input clock's period. Internally, the model autonomously measures the input clock's period to replicate it on the output clock.

Using [4] we generate a synthesizable representation of the MLIR-based model and re-validate it using updated testbenches

TABLE I

THE MODULE'S LATENCY DEPENDS MOSTLY ON CONTROL-FLOW RELATED COMPUTATIONS. THIS IS REFLECTED BY THE MINIMAL RESOURCES UTILIZATION REQUIRED FOR ITS SYNTHESIS.

Resource	Module Utilization	Timing-related Utilization
LUT	2354	305
FF	2259	456
DSP	10	0
BRAM	1	1

to ensure that the toolchain-generated model behavior reproduced the original RNM's one. In this example, the computation performed by the synthesizable model will complete within 30 clock cycles, 12 of which are dedicated to the computation of sensitivity lists and delay detections. The original analog circuit must manage the phase of a 1GHz clock and the smallest modeled latency is 1/128th of the clock period, so the minimum time step that must be modeled has a duration of 7.812ps. Accordingly, the RNM specifies a `timeunit` of 1ps and a `timeprecision` of 1fs. These time steps are too small to be modeled with any FPGA-based prototype, where the maximum available frequencies are in the order of hundreds of MHz. To represent smaller time steps, the time representation of the digital part of the prototype us be scaled down by a constant factor to hide the computation latency of the generated model from it.

A conservative approach would follow the designer's specification and evaluate the model in 1fs time steps, scaling the digital components' clock by a factor of $10^7 \cdot max\_cycles = 3 \cdot 10^8$ to let the generated model execute within a time step. This would nullify the performance benefits of the emulation and result in the useless emulation of thousands of time steps with no change in signals. Choosing a time step length in line with the model behavior results in a more efficient time scaling: sampling inputs and updating outputs with a frequency double the highest one present in the system is enough. Scaling the digital clock by a factor $128 \cdot 2 \cdot 10 \cdot max\_cycles = 76800$ is enough to model the smallest time step represented in the RNM with an FPGA running at 100MHz. Generating testbenches compatible with the new time scaling is a matter of applying the same scaling to all delays defined in the testbenches rather than rewriting them from scratch for the new models.

The model was synthesized on a Virtex 7 board using Vivado [13]. Despite the use of a buffer for modeling the transport delay, the area occupation data (Table I) show that there is still room for synthesizing the phase rotator and its digital controller on a single board.

V. CONCLUSIONS

We have presented a methodology for generating synthesizable representations of RNMs with procedural timing controls. When implemented inside an automatic toolchain, it uses existing and validated RNMs rather than requiring designers to replicate the analog circuit behavior through other hand-built models. The generated synthesizable results are suitable for testing mixed-signal models with very long sequences of instructions through emulation. This approach minimizes the differences between validated RNMs and their synthesizable counterparts and does not make any autonomous decisions about numerical or time-based approximations. The throughput performance reduction required for high-resolution time modeling comes from the use of a fixed time step model. This approach results in synthesizable models that fit into existing development and validation flows with minimal manual intervention by the designers, thus minimizing the chance of introducing differences between the simulation-validated models and their prototypes. Furthermore, the generated models' architecture is compatible with existing SW-oriented optimization tools and offers a high level of flexibility for implementing further domain-oriented optimizations. For example, testbenches supporting a variable time step could significantly improve emulated models' throughput. Also, adopting lower-precision floating-point numbers representations could reduce generated models' requirements for area occupation and latency.

REFERENCES

[1] IEEE, "IEEE standard for universal verification methodology language reference manual," *IEEE Std 1800.2-2017*, pp. 1–472, 2017.

[2] Siemens, "Siemens Veloce," https://eda.sw.siemens.com/en-US/ic/veloce/strato-hardware/, accessed: 2022-05-23.

[3] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "MLIR: Scaling compiler infrastructure for domain specific computation," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2021, pp. 2–14.

[4] F. Ferrandi, V. G. Castellana, S. Curzel, P. Fezzardi, M. Fiorito, M. Lattuada, M. Minutoli, C. Pilato, and A. Tumeo, "Invited: Bambu: an open-source research framework for the high-level synthesis of complex applications," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*, 2021, pp. 1327–1330.

[5] N. Georgoulopoulos and A. Hatzopoulos, "Parameterizable real number models for mixed-signal designs using systemverilog," *Journal of Electronic Testing*, vol. 37, 12 2021.

[6] S. Balasubramanian and P. Hardee, "Solutions for mixed-signal soc verification using real number models," *Cadence Design Systems*, pp. 1–4, 2013.

[7] X. Yang, X. Niu, J. Fan, and C. Choi, "Mixed-signal system-on-a-chip (SoC) verification based on systemverilog model," in *45th Southeastern Symposium on System Theory*, 2013, pp. 17–21.

[8] B. C. Lim and M. Horowitz, "An analog model template library: Simplifying chip-level, mixed-signal design verification," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 1, pp. 193–204, 2019.

[9] F. A. Nothaft, L. Fernandez, S. Cefali, N. Shah, J. Rael, and L. Darnell, "Pragma-based floating-to-fixed point conversion for the emulation of analog behavioral models," in *2014 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2014, pp. 633–640.

[10] P. Tertel and L. Hedrich, "Real-time emulation of block-based analog circuits on an FPGA," in *2017 14th International Conference on Synthesis, Modeling, Analysis and Simulation Methods and Applications to Circuit Design (SMACD)*, 2017, pp. 1–4.

[11] S. Herbst, G. Rutsch, W. Ecker, and M. Horowitz, "An open-source framework for fpga emulation of analog/mixed-signal integrated circuit designs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 7, pp. 2223–2236, 2022.

[12] IEEE, "IEEE standard for systemverilog–unified hardware design, specification, and verification language," *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)*, pp. 1–1315, 2018.

[13] Xilinx, "Vitis HLS LLVM 2021.2," accessed: 2022-05-23. [Online]. Available: https://github.com/Xilinx/HLS

[14] M. Popoloski, "sv-lang," https://sv-lang.com/, accessed: 2022-05-23.