AN OPEN-SOURCE FRAMEWORK FOR FPGA EMULATION OF ANALOG/MIXED-SIGNAL INTEGRATED CIRCUIT DESIGNS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Steven Herbst
June 2021

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Mark Horowitz, Primary Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Amin Arbabian

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Pat Hanrahan

Approved for the Stanford University Committee on Graduate Studies.

Stacey F. Bent, Vice Provost for Graduate Education

This signature page was generated electronically upon submission of this dissertation in electronic format. An original signed hard copy of the signature page is on file in University Archives.

Abstract

FPGA emulation is widely used to speed up simulations of digital chip designs, but the technique is difficult to extend to analog/mixed-signal (AMS) designs because they contain blocks that cannot be directly mapped to an FPGA's programmable logic. This necessitates the creation of synthesizable models for AMS blocks, a process that is time-consuming, error-prone, and expensive, as it requires a rare combination of expertise: analog modeling and FPGA implementation. Furthermore, fast AMS simulation techniques do not always work well on an FPGA, and can limit the speedup that is possible through emulation.

This thesis is about my efforts to unlock the potential of FPGA emulation for AMS designs by developing new high-performance AMS emulation techniques, along with an open-source framework that makes it easy to apply those techniques. The underlying premise is that AMS emulators should take large, but precise, timesteps corresponding to digital events, leveraging compile-time precomputation to make the FPGA implementation more efficient. "Analog-only" timesteps should be avoided, since their effect on emulator performance is more severe than in a CPU-based simulation.

The open-source emulation framework has been applied to six commercial designs, as well as one academic design. I focus on three of those applications in this thesis: an ADC-based high-speed link design, a firmware-controlled flyback converter, and an NFC-powered chip. Across those applications, speedups as compared to existing CPU-based simulations ranged from 2-3 orders of magnitude, creating new opportunities for pre-silicon firmware development and RTL-level verification of slow feedback loops. My hope is that this freely available, high-performance framework will help to break through the verification bottleneck in modern AMS design.

Acknowledgments

First off, I'd like the thank the organizations whose funding made this research possible: the Hertz Foundation, the Standard Graduate Fellowship (SGF), Stanford SystemX, and DARPA POSH. The freedom provided by Hertz and SGF was amazing, empowering me to explore "side quests" far outside the scope of this thesis. In addition, I am grateful to have been able to count on the support of the Hertz community, both in providing feedback on ideas and, near the end of the journey, in brainstorming ideas about what to do next. On a similar note, I am glad to have had the chance to be part of the open source community around the DARPA IDEA/POSH programs.

Of course, I wouldn't be at this point if it weren't for my advisor, Prof. Mark Horowitz, who invited me to come to Stanford in 2016 to work on open-source tools. I particularly appreciate how supportive he was in my experiments to bring software methods into our analog design flows, even when it was not smooth sailing at first.

I would like to thank the entire VLSI Research Group for being a fun, thoughtful group, and for teaching me all sorts of new things about software and hardware design. More specifically, thanks to the circuits subgroup for being such a great team through the ButterPHY and DragonPHY tapeouts. I think we ultimately figured out a really effective workflow, so thank you for putting in the effort to make that happen.

I'd also like Dr. Byong Chan Lim, who was a researcher in Mark's group near the beginning of my time at Stanford. Byong helped me get started on the analog/mixed-signal emulation path, and provided really insightful feedback on my early work.

More broadly in the EE Department, thank you to John DeSilva and Joe Little for helping to set up and maintain servers for regression testing on physical FPGA boards. This was a unique capability that proved useful for several open-source projects. Although no one could have anticipated it when those servers were set up, they proved invaluable for FPGA testing during the COVID-19 pandemic, when we didn't have physical access to research facilities.

Outside of Stanford, I'd like to thank Gabriel Rutsch at Infineon for being a great collaborator throughout the development of the emulator framework. He not only led the development of one of the framework's tools, but championed the project within Infineon, finding many interesting use cases. I would particularly like to thank him for setting up a three-month visit for me to work on the

framework at Infineon's global headquarters in Munich in 2019, which was an amazing experience.

On a personal note, I would like to thank my parents, Pat and Barbara, for being so warmly supportive during the Ph.D. program, and long before, laying the groundwork that made it possible to complete such an endeavor. Thanks as well to my sister Wendy and her husband Nick for being great compatriots through the grad student experience, as they worked on Ph.D. programs at UCLA.

Finally, I would like to thank my girlfriend, Mahati. We ended up taking a serial, not parallel, approach to grad school: she completed a Ph.D. program at Berkeley around the time that I started at Stanford. That wasn't the plan from the beginning, but she was nonetheless supportive of my decision to leave industry to start a Ph.D. program. Having recently been in my shoes, she has had a bunch of great advice. But above all, I have appreciated her joyful outlook on life, which buoyed me through the tough parts of the Ph.D. journey.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

We are in what has been called a "new golden age" for innovation in chip design [27], driven by the convergence of several factors. On one hand, increasingly complex machine learning applications are being deployed in mobile devices and on the cloud, but they are constrained by power consumption, out of both battery life and cost considerations. At the same time, with Dennard scaling having faded some time ago, we cannot rely on silicon process innovations to deliver the performance needed for these new applications.

These effects, among others, have started moving the spotlight back to innovation in chip design in recent years. Partially motivated by that trend, and partially reinforcing it, there has been a resurgence of interest in open-source tools for chip designs. In fact, I would count myself as part of that trend: back in 2016, having spent five years working as both a vendor and customer of chip designs, and having observed how slow, expensive, and risky the process can be, I decided to take a break from industry to help make chip design more accessible through open-source tools.

Over the past five years, there has been an explosion in both the quantity and quality of open-source tools for building chips, culminating in the 2020 release of a completely open-source flow, called OpenLANE [62], that provided an RTL-to-GDS flow for a real silicon process, SKY130 [20]. This is an incredible accomplishment for the open-source community, and represents the tireless efforts of dozens of organizations over many years.

As these tools advance, they will drive down the time and cost of *constructing* chip designs. However, *manufacturing* the designs themselves remains exceptionally expensive and time-consuming, and as a result, verifying chip designs prior to tapeout remains as important as ever.

Computer simulation is the workhorse for verifying blocks within chip designs, but it tends to run out of gas at the system level, where the entire chip design must be tested in conjunction with firmware that runs on it and software that interacts with it. For example, an IBM study [6] showed that it would take several years to simulate Linux booting on one of their new processor designs.

As a result, hardware emulation is often used instead of computer simulation for the highest-level

verification tasks. This entails mapping the chip design to a programmable digital platform, such as an off-the-shelf FPGA board or a commercial emulator (e.g., Cadence Palladium [13], Synopsys ZeBu [65], and Mentor Veloce [23]). There even now an open-source tool, FireSim [35], for emulating processors in the cloud on Amazon F1 instances.

On all of those platforms, emulation of digital systems can result in an orders-of-magnitude speedup as compared to computer simulation. In fact, the speedup is often so significant that the emulator can be used for pre-silicon firmware and software development, shaving months off project schedules. Hence, emulation is a powerful tool with broad applicability: it can be used not only for traditional pre-silicon verification, but also for bringup preparation, test vector development, and by customers as an early evaluation and development platform.

The problem is that many chip designs contain analog/mixed-signal (AMS) blocks that have no direct mapping to a digital emulator. Such blocks often provide the interface between the physical world off-chip and the digital world on-chip, serving in key roles such as power management and high-speed I/O [5]. These blocks cannot typically be ignored in system-level verification, because they often interact with digital blocks through complex feedback loops. Hence, building an emulator for many chip designs requires AMS blocks to be replaced by synthesizable digital approximations.

The key challenge in making that approximation is that analog signals vary continuously in time, with continuous values, whereas digital logic operates in discrete time, with discrete values. Continuous values can be represented using either fixed-point or floating-point formats, but neither is a clear-cut winner for emulation: floating-point is slow and resource-intensive, while fixed-point suffers from range and resolution issues. For time discretization, a common approach, called oversampling, is to assign a fixed timestep to each "tick" of an emulator clock. Unfortunately, that approach introduces a direct tradeoff between time resolution and emulator throughput, which can decimate the performance of emulators containing AMS blocks requiring fine time resolution.

Consequently, it is tricky to architect high-performance AMS emulation models, a problem that is compounded by the fact that there is no complete, publicly-available tool for implementing and running such models within the context of a full-chip emulation. Instead, AMS emulation is often done in an ad-hoc fashion, using tools designed for other purposes. The result is that building synthesizable AMS models is typically a time-consuming, error-prone process, requiring an unusual combination of expertise: analog modeling and FPGA design. That barrier to entry, along with the difficulty of achieving good performance, means that the full potential of emulation is not often realized for mixed-signal chips.

This thesis is about my efforts to change that. It starts with a review of related work (Chapter 2), which demonstrates that there is no complete, publicly-available framework for constructing full-chip AMS emulators. In addition, I show that the traditional approach to discretization for synthesizable analog modeling is a method called "oversampling." Chapter 3 dives deeper into oversampling, highlighting a performance bottleneck that I uncovered: namely, that a single analog block requiring

fine time resolution can decimate the performance of a full-chip emulator. The following chapter describes my solution, which is to decouple the emulator speed and its analog accuracy by using variable timesteps, with analog waveforms represented using "feature vectors" of spline points.

Chapter 5 changes gears to describe the first complete, publicly-available framework for mixed-signal emulation, which I developed in collaboration with Infineon. The framework consists of three open-source tools: (1) a Python-based synthesizable model generator for mixed-signal blocks (**msdsl**), (2) a fixed- and floating-point synthesizable SystemVerilog library for representing real numbers (**svreal**), and (3) a Python-based tool that generates emulator control infrastructure and automates the FPGA build process (**anasymod**). The framework includes features for efficiently modeling analog dynamics, nonlinearity, and noise, often making use of compile-time precomputation to reduce the required computational resources of the FPGA.

Finally, Chapter 6 illustrates the generality of the framework through three real-world applications: (1) a firmware-controlled flyback converter, (2) an NFC-powered chip, and (3) an open-source high-speed link receiver, DragonPHY. In all cases, the framework provided a speedup of 2-3 orders of magnitude as compared to existing computer simulations.

Chapter 2

Related Work

This chapter provides an overview of previous work that has been done in the AMS emulation space. The takeaway will be that, while there have been tools and techniques proposed to solve specific problems in AMS emulation, there isn't a framework that ties those ideas together, making it straightforward to run AMS emulation at the full-chip scale. It will also become clear that when it comes to analog modeling techniques for emulation, oversampling (i.e., fixed-timestep, piecewise-constant waveforms) is almost always the approach used. That is important because, as I will show in the next chapter, oversampling is limited by a fundamental tradeoff between time resolution and emulator throughput; that limitation was what motivated me to develop the spline-based methods described in this thesis.

2.1 Hardware-in-the-Loop Emulation

Analog emulation spans a range of virtualization levels, from hardware-in-the-loop (HIL) to fully virtual. HIL emulators implement digital parts of a design on an FPGA, but analog blocks are implemented as physical components on a PCB, with analog-to-digital converters (ADCs) and digital-to-analog converters (DACs) serving as interfaces between the two domains. Fully virtual emulators, on the other hand, implement the entire system on an FPGA, including analog parts as well as digital parts.

An example of HIL emulation is work by R. Sanchez et al. in LASCAS 2016 [61], which described a HIL emulator for a high-speed link transceiver. In that case, the authors had a design for a time-interleaved ADC (TI-ADC) that they wanted to incorporate into the chip design for a complete transceiver. Since they had taped out the TI-ADC by itself, they built a HIL emulator around it, using a physical communication channel, driven by a high-speed DAC. The rest of the transceiver design, which was all digital, was implemented on an FPGA. The result was a 1 Gb/s high-speed emulator that could be used to study TI-ADC calibration algorithms.

When physical analog components are available that closely match the behavior of analog blocks in a chip design, the HIL approach can provide a very fast and realistic emulator. However, that will not generally be the case, as new analog circuits are typically designed alongside the digital circuits that will interface with them. In addition, there is a limit to the speed and density of analog signals that can be implemented at the PCB level, which may not be sufficient to match the IC design being emulated.

Hence, this thesis focuses on fully virtual AMS emulation, where the entire chip design is modeled within an FPGA. This is in some ways a more general case, because it makes it possible to model the behavior of custom analog blocks, which are not available as PCB-level components, as well as analog blocks that are too fast or too highly integrated for PCB-level implementation. A side benefit is that a fully virtual emulator is not constrained to run in real-time, which may be important if digital parts of the design run faster than is possible on an FPGA.

2.2 Oversampling with Xilinx System Generator

An example of the fully virtual approach is the DC-DC buck converter emulator developed by R. Bhattacharya et al. [9]. The authors' motivation was to create a virtual DUT that could be used in an automated test equipment (ATE) setup to verify its connectivity prior to the availability of silicon, and to do pre-silicon test vector development. Since the DUT was fully virtual, the buck topology (i.e., passive analog components and switches) needed a synthesizable model. The authors used an oversampling approach to construct that model, first selecting a fixed timestep (50 ns), and then using that timestep to hand-derive discrete-time approximations of the buck converter's dynamics. The resulting discrete-time equations were implemented in a synthesizable fashion using Xilinx System Generator [76], which provides a GUI for block diagram entry. The accuracy of the emulator compared to SPICE simulations was pretty good: just a few percent relative RMS error.

There are a few interesting takeaways from Bhattacharya's study. First, it provides a data point on how much oversampling is necessary to get reasonably accurate results: since the buck converter switching frequency was 200 kHz and the emulator timestep was 50 ns, they needed on the order of 100 timesteps per switching period. This is quite well aligned with the analysis of oversampling in the next chapter.

The second interesting point about Bhattacharya's work is that it was intended for real-time emulation, meaning that the emulator clock frequency had to be the inverse of the timestep represented by each emulator cycle. As the authors point out in their paper, this means that emulators of larger systems, which must run at lower frequencies due to longer logic propagation delays, will end up having a lower oversampling ratio. That in turn reduces emulator accuracy, meaning that a real-time oversampled emulator has a direct tradeoff between DUT scale and emulator accuracy.

The last observation about Bhattacharya's paper has to with a different aspect of scalability:

the time taken to create synthesizable models. The modeling flow involved a hand-discretization of analog dynamics, which was implemented in Xilinx System Generator, with fixed-point formats of model I/Os manually set according to signal ranges observed in Verilog-A or Simulink simulations. In other words, the process was fairly involved both in terms of the number of tools required and the amount of manual effort. For the buck converter design considered in the paper, this was not a problem, since only a few synthesizable models were needed. But it could present an issue for emulating larger systems.

2.3 Oversampling with Simscape Electrical

Another approach to analog emulation was described by B. K. Mishra et al. at ICWET 2011 [49], which suggested using Simulink HDL Coder [45] to construct synthesizable models. HDL Coder by itself is similar to Xilinx System Generator, in that it provides a GUI for drawing block diagrams of discrete-time systems. However, Mishra points out that there is a package available for HDL Coder called "Simscape Electrical" that can generate synthesizable models of analog circuits that are drawn in Simulink.

Mishra's paper only considers modeling a small circuit (a CMOS inverter, i.e. two transistors), and it does not appear that the authors attempted to run the model on an FPGA, so it is unclear from the paper whether Simscape Electrical is suitable for full-scale emulation. As a result, I assessed the tool's capabilities directly from its user guide [46]. It turns out that although Simscape Electrical has many advanced features for computer-based simulation, its HDL generation capabilities are more limited. For example, it supports linear and "switched linear" (i.e., passive components, and switch models of transistors and diodes), but not nonlinear devices. In addition, it is limited to fixed-timestep operation; it appears to solve analog dynamics at compile time using the Backward Euler method, according to a user-provided fixed timestep. Finally, Simscape Electrical does not support HDL generation for events, delays, and time-varying components.

2.4 Emulation of Discrete-Time Analog Circuits

Another application of fully virtual AMS emulation is the emulator built by G. Wang and Y. Chiu to study calibration algorithms for a SAR ADC [71]. In that case, the analog blocks that needed synthesizable models were the capacitor DAC and comparator that are fundamental to any SAR ADC design, along with an auxiliary capacitor DAC used in a dither-based calibration scheme. It appears that the authors manually constructed fixed-point models for these analog blocks, along with the infrastructure to move data into and out of the emulator. They were able to run the emulator at 50 MHz, which corresponded to a 3,000x speedup as compared to their existing MATLAB simulation.

The interesting thing about this example is that SAR ADCs naturally operate in discrete time,

with one cycle for each step of their binary search. As a result, oversampling is not necessary to model the basic operation of a SAR ADC. This study therefore suggests that AMS emulation can achieve a speedup of 3-4 orders of magnitude if oversampling is eliminated. Of course, that is only straightforward for a discrete-time system such as a SAR ADC, but as we'll see later, I achieved similar speedups by using techniques to reduce the need for oversampling in continuous-time systems.

2.5 Automated Floating- to Fixed-Point Conversion

In some ways, the closest thing to a general tool for analog emulation is a paper by F. Nothaft et al. in ICCAD 2014 [50], which was about the emulation of an entire cellular modem IC. This likely represents the largest design considered in this chapter, because behavioral models for analog models alone spanned 72,000 lines of code. Since the analog functionality was so substantial, and had already been modeled for computer simulation, the authors set out to convert the existing analog models to a synthesizable format for emulation. They did this by creating a library of compile-time pragmas to convert floating-point types in the existing analog models to fixed-point, given user-annotated range and resolution information.

Since the pragma library was designed to perform a one-to-one mapping of behavioral models from a non-synthesizable simulation format to a synthesizable emulation format, its focus is arithmetic operations: addition, subtraction, multiplication, and "limited exponentiation" (presumably powers of two). As a result, the tool leaves the time discretization process up to the user, although it appears that intent was to use oversampling, based on an example in the paper about using the trapezoid rule [31] to discretize an RC filter.

After applying the pragma library, the result was a synthesizable model of the chip, which the authors mapped to Cadence Palladium, a commercial emulation platform. The result was an emulator that ran at 1 MHz, providing a speedup of 120x as compared to existing RTL simulations. Considering the size and complexity of the chip design, this is an impressive feat. Presumably due to the fact that the pragma library was developed within a chip design company, the tool was not publicly released.

2.6 Library of Oversampled Analog Blocks

Another tool for mixed-signal emulation, which was also not publicly released, was described by P. Tertel and L. Hedrich [68]. Their work focused on real-time emulation of analog behaviors on an FPGA, for use in a HIL emulator. In other words, they were designing a partially virtual AMS emulator, with some analog behaviors modeled on an FPGA, and some behaviors implemented by physical components on a PCB, with the two domains interfacing through ADCs and DACs. As with the other work described so far, they used an oversampling approach, with the oversampling

rate set to a relatively low frequency, 88.2 kHz, so that they could use precision audio ADCs (16-bit) and DACs (24-bit).

Tertel and Hedrich took a higher-level approach in their framework, providing a small library of synthesizable analog functions: RC filters, addition and gain stages, rectifiers, and a sample-and-hold. In a manner somewhat like Simscape Electrical, users create block diagrams of those library components, albeit using a SPICE netlist, not a GUI. The framework then converts the user's SPICE netlist into VHDL, using a fixed-point datatype to represent analog waveforms. The fixed-point type was not auto-formatted or user-defined; it was globally defined to have ±32 V range and a few microvolts of resolution. This was sufficient for the types of systems that Tertel and Hedrich were modeling, where all real-number signals corresponded to physical voltages, rather than a more general composition including signals such as times and capacitances (which can be very small) and frequencies and resistances (which can be very large).

2.7 Oversampling at the Component Level

Another intriguing framework for partially virtual AMS emulation is a tool called WaveACE, presented by W. Wu et al. at ISCAS 2016 [73] (the tool was not publicly released). The authors took an oversampling approach to emulation, but their goal was to perform discretization at the level of individual components: resistors, capacitors, inductors, etc. Although they could have simply used an explicit integration method such as forward Euler, it would have required exceptionally small timesteps.

Wu instead sought to use the trapezoid rule for discretization, which is an implicit method. In general, implicit methods require the simultaneous solution of all signals, which is hard to parallelize on an FPGA. However, Wu points out that by treating analog components as transmission lines with specially-chosen port impedances, the trapezoid rule becomes an explicit integration method, and therefore a better candidate for FPGA emulation. (This approach to discretization is called a "wave digital filter," or WDF.)

WDFs are a clever technique for local simulation of analog circuits at component-level, but there are a couple of drawbacks. The first is that it is not particularly fast: the resulting emulator could only process oversampled timesteps at 512 kHz, which is 1-2 orders of magnitude slower than many of the other approaches described. The reason seems to be that there is a lot of computational overhead required in connecting together WDFs, since each connection point requires an impedance adapter. For example, a seven-transistor differential amplifier requires 38 adaptors.

The second drawback is that WDFs seem to have some difficulty handling nonlinearities. Wu points out that some extensions for nonlinearities have been proposed for use in computer simulation, but they don't map well to an FPGA. As a result, WaveACE can only handle small-signal models of nonlinear devices, such as transistors, using lookup tables for small-signal resistances, capacitances,

etc. In theory, those lookup tables could be addressed according to circuit operating points, although WaveACE did not have that capability. Instead, the lookup tables were addressed in an off-line manner, essentially setting bias points for a small-signal transient analysis.

2.8 Running Oversampled Analog Models on a Processor

An intriguing alternative to synthesizable analog models was presented by A. Fernandez-Alvarez et al. at DCIS 2016 [17]: emulating digital parts of a design directly on an FPGA's programmable logic (PL), but simulating analog parts of the design with its processor system (PS). This takes advantage of the fact that most modern FPGAs contain one or more "hard" processors that can run bare metal code at a few hundred MHz.

Fernandez-Alvarez used an oversampling approach to analog modeling, as in the other projects discussed, but with a twist: the sampling frequency of the analog models could be any multiple or submultiple of the sampling frequency of digital models in PL. Hence, if the sampling frequency of the analog models is faster than that of the digital models, then analog models will run autonomously for several cycles before "syncing" with the digital models prior to their next cycle.

Although implementing analog models in the PS provides flexibility, it unfortunately comes at the expense of emulator speed. For example, the buck converter implemented by Fernandez-Alvarez operated at a speed of 12.6 switching cycles per second, which is more than four orders of magnitude slower than Bhattacharya's fully synthesizable buck model.

2.9 Gate-Level Timing Emulation

There is one published example of an AMS emulation flow that did not use traditional oversampling: Henkel and Ossoinig's presentation at CDNLive 2013 [26].[1] They were concerned with accurate timing modeling of digital edges in analog circuits, such as delay lines, a PLL, and a time-to-digital converter (TDC).

The solution they came up with was essentially a hybrid of oversampling and an event-driven scheduler: they had emulation time tick forward in fixed increments of 1.6 ns, but attached a 14-bit time offset to timing-critical digital signals, indicating when their transitions occurred with respect to the last emulator tick. With that scheme, logic propagation delays are easily implemented as additions to the timing offset.

Henkel and Ossoinig applied their concept to a 100-million transistor chip used in ATE products. Since the design was large, and modeled at gate level in some places, they built the emulator with Cadence Palladium, ultimately achieving a speedup of 50x as compared to their fastest RTL simulation. Overall, they demonstrate that it is possible to run an AMS emulator with fine time

[1] The slides are no longer available on Cadence's website, but a summary of the talk can be found in a blog post from Steve Carlson [14].

resolution without sacrificing much performance; their speedup is on the same order as what was reported by Nothaft for a different commercial design. That said, the approach is limited to modeling the timing of digital transitions in analog circuits; it does not address how analog dynamics interact with the time-offset edges.

Chapter 3

The Trouble with Oversampling

As became clear in the previous chapter, the state-of-the-art in constructing synthesizable analog models is essentially still an old technique called "oversampling." Unfortunately, it turns out that in some cases, oversampling can introduce a direct tradeoff between speed and accuracy that decimates emulator performance. This chapter delves deeper into oversampling to describe precisely what it is, and why it can be problematic.

To illustrate how oversampling works, suppose that we wish to emulate a chip design that includes a resistor-capacitor (RC) filter (Fig. 3.1). An FPGA does not include resistor or capacitor primitives, so we will need to approximate the RC filter with the blocks that are available on the FPGA: look-up tables (LUTs), flip-flops (FFs), block RAM (BRAM), and DSP slices, which contain integer multipliers.

To do that, we start with the continuous-time dynamics of the filter:

$$\frac{dy}{dt} = \frac{x - y}{RC} \tag{3.1}$$

where the filter's input is $x\,(t)$, its output is $y\,(t)$, its resistance is R, and its capacitance is C. We will discretize the dynamics according to a fixed timestep, Δt, that is short compared to the bandwidth

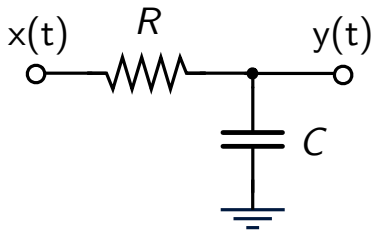


Figure 3.1: Basic RC filter with an input $x\,(t)$ and output $y\,(t)$

of the filter's input and output. Although the bandwidth of the output is bounded by the filter's cutoff frequency, the bandwidth of the input could be anything. As a result, it is not necessarily sufficient to simply set Δt to a small fraction of RC.

Once the timestep has been selected, there are many approaches for discretization, such as the Runge-Kutta methods (which include the Euler methods) [31]. However, one of the most commonly used approaches is called Zero-Order Hold (ZOH) [21], because it provides an exact solution when the system's input is a piecewise-constant waveform. The ZOH approach entails solving for the response of a system to a constant input, evaluating it at a fixed timestep Δt, and expressing the result as a discrete-time equation. As an example, the RC filter's response to a constant input x_0 is:

$$y(t) = y(0) \cdot e^{-t/(RC)} + x_0 \cdot \left(1 - e^{-t/(RC)}\right) \tag{3.2}$$

If we set $t = \Delta t$, this equation tells us how to compute the next output value of the filter, $y[k+1]$, given its previous output, $y[k]$, and the input voltage over the timestep, $x[k]$:

$$y[k+1] := \alpha \cdot y[k] + \beta \cdot x[k] \tag{3.3}$$

where $\alpha = \exp(-\Delta t/(RC))$ and $\beta = 1 - \exp(-\Delta t/(RC))$. Since the coefficients α and β are known at compile-time, this is a good fit for the resources available on an FPGA: the two multiplications each map to a DSP slice, the addition maps to LUTs (which often contain carry-chain logic), and the filter's previous output, $y[k]$ is mapped to FFs. Of course, this presumes that x and y are represented as fixed- or floating-point numbers (that implementation detail that will be discussed later, in Chapter 5).

The big problem with oversampling is that we often need to make the timestep fairly small in order to achieve reasonable accuracy. There are at least two reasons why that is the case: analog waveforms are not well-described as piecewise-constant (PWC) waveforms, and the difficulty of representing events with fine time resolution.

On the first point, consider the error of approximating a sine wave as PWC. The effective number of bits (ENOB) of fidelity of the approximation[1] is shown in Figure 3.2 as a function of the number of timesteps per period of the sine wave, revealing that many timesteps are needed even to achieve a relatively coarse level of accuracy. In addition, the accuracy improves very slowly with the number of samples; we must double the number of timesteps to gain one bit of fidelity.

On the second point, some digital events, such as clock jitter, need to be modeled with very fine time resolution even from a behavioral modeling perspective. As an example, consider the high-speed link receiver blocks described by S. Kim in VLSI 2020: a 20 GS/s time-interleaved converter

[1] ENOB was calculated as $\log_2\left[\text{peak-to-peak signal range}/\left(\text{RMS error} \cdot \sqrt{12}\right)\right]$, according to an application note from Maxim Integrated Products [48]. The RMS error, in turn, was calculated as $\sqrt{\frac{1}{2\pi}\int_0^{2\pi}\left(\sin(\theta) - \text{PWC}_n(\theta)\right)^2 d\theta}$, where PWC_n is the n-segment PWC approximation of $\sin(\theta)$.

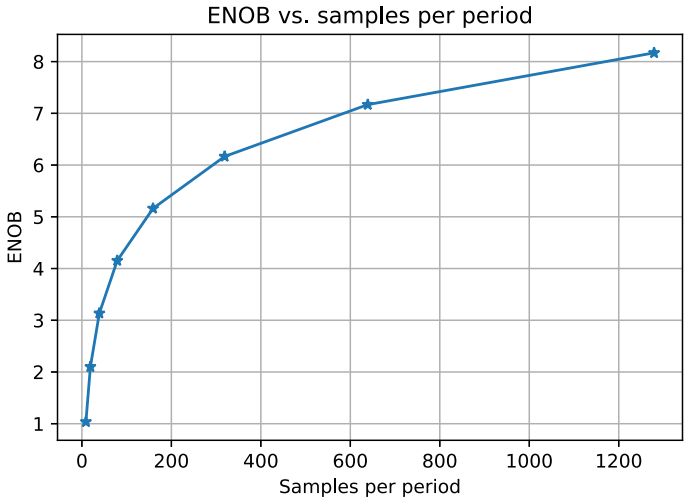


Figure 3.2: Effective number of bits (ENOB) in a piecewise-constant representation of a sine wave, as a function of the number of timesteps per period of the sine wave.

whose sampling points are adjusted by phase interpolators (PIs) with 0.7 ps resolution. Modeling the behavior of the system down to single-bit changes in the PI codes would necessitate oversampling with a 0.7 ps timestep, which in turn would lead to about 70 timesteps per ADC sample.

Using a smaller timestep necessitates taking more timesteps to simulate or emulate a given amount of time. Interestingly, however, small timesteps have a more direct and severe impact on the performance of hardware emulation as compared to computer simulation. We explore that effect in the next two sections.

3.1 Simulation Performance

We first consider the impact of oversampling on the performance of computer-based simulation, as a comparison point for the emulation case. As the basis for this analysis, consider the generic RTL model for a mixed-signal chip design shown in Figure 3.3, which consists of digital circuits updated at a clock rate f_{clk}, as well as an oversampled model of analog circuits, which are updated at a rate f_{os}. We are interested in the simulation performance of such a design in terms of the amount of oversampling.

Imagine that it takes the simulator a time $T_{sim,dig}$ to update the digital circuits after each digital clock event, and a time $T_{sim,ana}$ to update the analog circuits after each oversampling timestep. Since there is bidirectional communication between the digital and analog circuits, the two must stay in

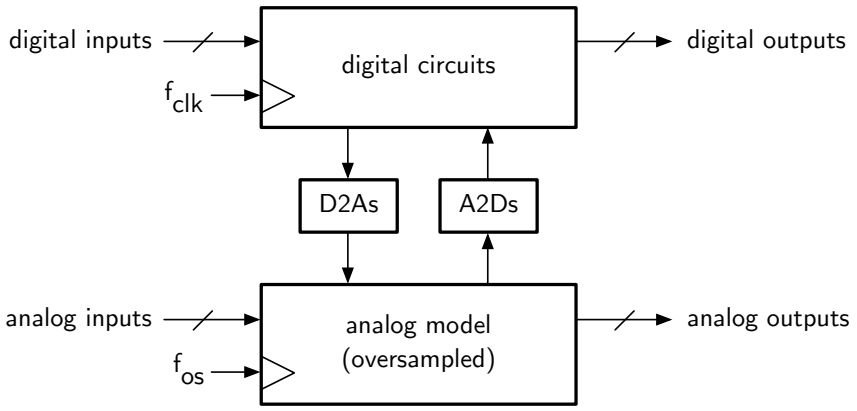


Figure 3.3: Generic model of a mixed-signal chip design used for performance analysis.

sync during simulation. Hence, the total time to simulate one clock cycle is:

$$T_{sim,clk} = T_{sim,dig} + n_{os} \cdot T_{sim,ana} \tag{3.4}$$

where the $n_{os} = f_{os}/f_{clk}$ is the number of oversampling timesteps per digital clock period.

A large chip design may consist of hundreds of thousands of lines of RTL [72], while a simple analog circuit, such as the RC filter just described, can be modeled with just a few lines of Verilog. As a result, $T_{sim,dig}$ may be orders of magnitude larger than $T_{sim,ana}$, meaning that the impact of oversampling in a "big D-little A" design can be insignificant, even if analog models are quite oversampled. In other words, it is often the case that $T_{sim,clk} \approx T_{sim,dig}$.

3.2 Emulation Performance

The situation looks quite different for an FPGA emulator. In that case, the time to emulate one digital clock cycle is related to the critical paths through the digital circuits and synthesizable analog models:

$$T_{emu,clk} = \max\left(T_{crit,dig}, n_{os} \cdot T_{crit,ana}\right) \tag{3.5}$$

where $T_{crit,dig}$ is the critical path through the digital circuits and $T_{crit,ana}$ is the critical path through the analog models. In both cases, *critical path* refers to the longest propagation delay through the FPGA implementation for emulation, not the ASIC implementation for manufacturing.

Since ASIC designs are often quite parallel, the critical path through digital circuits is not necessarily much longer than the critical path through analog models. This is because the path length is dependent on the depth of logical operations (i.e., number in series), rather than the total number of logical operations. As a result, analog models can easily become the dominant factor in emulator performance, i.e. $T_{emu,clk} \approx n_{os} \cdot T_{crit,ana}$.

When this is the case, there is a direct tradeoff between the emulation speed and emulation accuracy: increasing time resolution (n_{os}) by a certain factor will slow down the emulator by that same factor. This is particularly problematic because it has little to do with the relative size of the digital circuits being emulated as compared to that of the synthesizable analog models. That is the fundamental problem with oversampling: even when emulating a large digital system, emulator performance can end up being dictated by the time resolution required by a single analog model.

Chapter 4

The Analog Timestep Vanishes

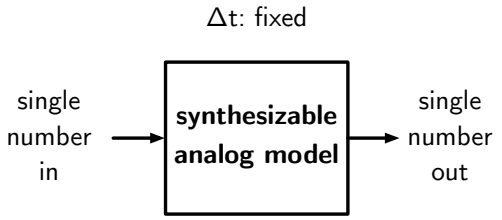
The root cause of the performance bottleneck in oversampling is that it does not provide a particularly descriptive way of representing analog signals. The analog inputs and outputs of synthesizable models are represented by a single value for the full duration of a timestep, as illustrated in Figure 4.1a. This piecewise-constant representation does not line up very well with the true shape of most analog waveforms, which tend to vary smoothly. As a result, oversampled models often need to use small timesteps in order to achieve reasonable accuracy.

To make matters worse, the use of a fixed timestep makes it difficult to achieve reasonable accuracy for analog effects such as clock jitter, since there is a direct tradeoff between the time resolution with which those effects can be represented and the speed of the emulator.

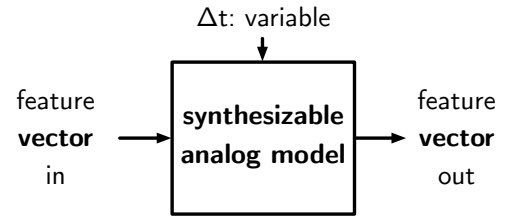
Having observed those issues in my early experiments with analog emulation, I set out to develop a more expressive way of representing the shape of analog waveforms that would enable emulators to take larger timesteps without losing much accuracy. In addition, I realized that accurately representing event timing would require an emulator to take variable timesteps, which in turn would mean that analog models would need to accept the timestep size as an additional input.

This led me to strive towards the general model architecture illustrated in Figure 4.1b: rather than using single numbers as the analog I/O format, represent the shape of analog waveforms in between timesteps with vectors of several numbers. In addition, rather than assuming a fixed timestep at compile-time, allow the timestep to be variable and connect it to each analog model as an input.

In formulating an analog "feature vector" for emulation, I drew inspiration from two related research efforts in the simulation space (i.e., CPU-based, not FPGA-based). The first approach, by S. Liao et al. [40], used a feature vector consisting of two numbers: an offset and a slope, used to approximate an analog waveform as piecewise-linear. The second approach, by J. Jang et al. [33], used a feature vector consisting of coefficients of exponential basis functions. Both approaches demonstrated impressive performance in CPU-based simulation.

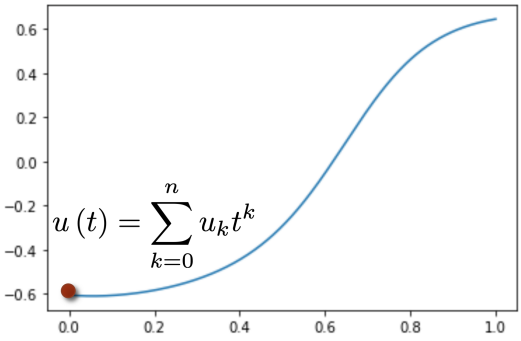


(a) Traditional oversampling approach: the emulation timestep is fixed at compile-time, and analog I/O is represented as single numbers.

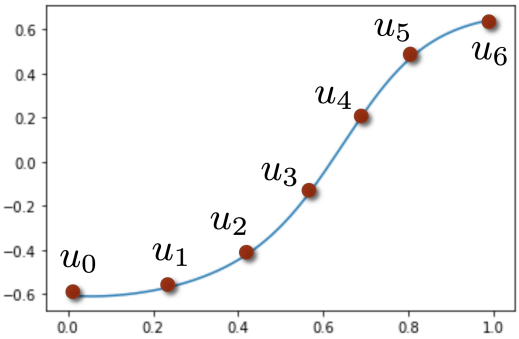


(b) Proposed feature vector approach: Represent analog waveforms in emulation using vectors of several numbers, with the timestep allowed to vary from one emulation cycle to the next.

Figure 4.1: Comparison of the traditional and proposed approaches to synthesizable analog modeling.



(a) Extrapolation: the feature vector consists of the coefficients of a polynomial waveform that extrapolates the shape of an analog waveform from the beginning of a timestep.



(b) Interpolation: the feature vector consists of points along a spline with an implicit interpolation method.

Figure 4.2: Two possibilities for an analog "feature vector."

However, the unique constraints of FPGA emulation meant that I could not directly translate either approach to a synthesizable implementation and call it a day. The PWL approach still would have required a fair number of intermediate timesteps for analog blocks, which, as I illustrated in the previous chapter, are detrimental to the performance of an FPGA emulator. The exponential basis function approach, on the other hand, can represent waveforms over longer durations, but carries a high computational cost that would limit the performance achievable on an FPGA.

As a result, I considered two more emulation-friendly ways of representing analog shapes: polynomial extrapolation (Figure 4.2a) and polynomial interpolation (Figure 4.2b). For extrapolation, the feature vector would consist of coefficients of a polynomial that projects the shape of analog waveform out from a starting point in time. For the interpolation approach, the feature vector would consist of several points along a spline, with an implicit interpolation method that describes how to connect the dots.

I explored both approaches, but found two key advantages for the interpolation approach: (1) it

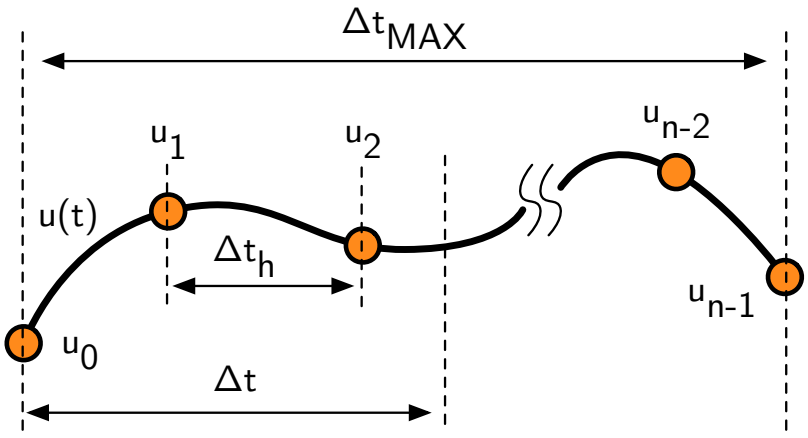


Figure 4.3: In the interpolation approach, the shape of an analog waveform $u(t)$ is projected out by an amount of time Δt_{MAX}, using n sample points and an implicit interpolation method.

allows for a simpler and more resource-efficient implementation of nonlinearities, and (2) it is less susceptible to numerical stability problems at large timesteps. The reason for the second issue is that increasing the timestep size for extrapolation requires increasing the polynomial order, whereas in the interpolation approach, additional spline points can be used, without increasing the order of the interpolating polynomial.

As a result, I decided to move forward with the interpolation approach. The rest of this chapter provides a more detailed description of the interpolation approach, along with descriptions of how three general types of analog blocks can be modeled with interpolation: (1) a static nonlinearity, (2) a state-space system, and (3) a system defined by an impulse response. I conclude the chapter with the experimental results of applying these modeling techniques to the analog frontend of a representative high-speed link design.

4.1 Interpolation

The proposed interpolation approach, illustrated in Figure 4.3, represents an analog waveform using n spline points, connected by an implicit interpolation method. The waveform's shape is projected out by an amount of time Δt_{MAX}; the emulator may take any timestep, as long as it is less than that amount.

Although it is not a fundamental requirement of this approach, I found it convenient to define the spline points with equal spacing $\Delta t_h = \Delta t_{MAX}/(n-1)$. The "h" subscript refers to the fact that the spline points are in a sense "hidden" timesteps. However, unlike traditional oversampling, the hidden timesteps are computed in parallel over one emulator cycle. Hence, the interpolation approach is a tradeoff of resource utilization and performance: it spends extra FPGA resources

to parallelize analog timesteps, with the aim of being able to project analog waveforms over the gaps between digital events, without taking extra timesteps. For a "big D, little A" chip design, it is likely worth spending the extra FPGA resources to prevent analog models from dominating emulator performance.

The interpolation between spline points is implicit in the sense that the emulator does not fill in the curve between spline points; the feature vector used for analog I/O consists solely of the spline points themselves. However, the interpolation must be defined for two reasons: (1) to allow the evaluation of a waveform at a specific point in time (e.g., for modeling the sampling behavior of an ADC), and (2) for modeling analog blocks with dynamics, whose internal state depends on the shape of the entire waveform, not just its value at specific points.

Many interpolation methods are possible, but I chose polynomial interpolation because it lends itself well to a resource-efficient, fast implementation with DSP slices on an FPGA. If we assume all spline points are connected by m-order polynomials, the analog waveform represented is defined by:

$$u\left(t\right) := \sum_{k=0}^{m} U_{jk} \cdot \left(\frac{t - j\Delta t_h}{\Delta t_h} \right)^k \quad \text{for} \quad j\Delta t_h < t < (j+1) \cdot \Delta t_h \tag{4.1}$$

In other words, the analog waveform is represented by $n-1$ polynomial segments that run between the spline points, which are separated by an amount Δt_h, where the coefficients of each polynomial segment are specified in the matrix U. To avoid numerical scaling issues, the argument to each polynomial segment is normalized to the interval $[0, 1]$.

The matrix U is computed by solving a system of equations. However, since there are $(m+1) \cdot (n-1)$ unknowns in U, simply requiring continuity at spline points, which introduces $2 \cdot (n-1)$ equations, does not sufficiently constraint the polynomial coefficients when $m > 1$ (i.e., when the interpolation order is quadratic or higher). Perhaps the best-known solution to this problem is to constrain derivative continuity up to the $(m-1)$-th order, but this has the drawback of making the interpolation non-local. In other words, every spline point affects the interpolation at any point along the curve.

Why might this be problematic? The issue has to do with calculating output spline points of analog models with internal state. Every output spline point must depend on all input spline points that precede it, but if the implicit interpolation method is non-local, then it also depends on the values of all spline points that come afterward. Hence, non-locality in the interpolation method means that the input-to-output spline points computation is full-connected, which is more resource-intensive than fundamentally necessary by causality.

To solve that problem, I created a simple interpolation method that I termed "overlapping interpolation," in which each polynomial segment is constrained to hit $m+1$ neighboring spline points. For example, for cubic interpolation, spline segment j is constrained to hit spline points u_{j-1}, u_j, u_{j+1}, and u_{j+2}. More generally, we can define each row of U as the solution to a system

of linear equations:

$$\begin{bmatrix} \vdots & & & \\ (-1)^0 & (-1)^1 & (-1)^2 & \dots \\ (0)^0 & (0)^1 & (0)^2 & \dots \\ (1)^0 & (1)^1 & (1)^2 & \dots \\ \vdots & & & \end{bmatrix} \cdot \begin{bmatrix} U_{j0} \\ U_{j1} \\ U_{j2} \\ \vdots \end{bmatrix} = \begin{bmatrix} \vdots \\ u_{j-1} \\ u_j \\ u_{j+1} \\ \vdots \end{bmatrix} \tag{4.2}$$

Hence, once we solve the systems of equations, each entry in U is a linear combination of the input spline points:

$$U_{jk} = \sum_{i=0}^{n-1} W_{jki} u_i \tag{4.3}$$

Where W_{jki} maps the i-th spline point to the k-th coefficient of the j-th polynomial segment.

The tensor W is computed at compile time, and effectively defines the interpolation method, since it dictates how spline points shape polynomial segments. But its use is by no means limited to the overlapping interpolation method I just described. W could just as well describe other interpolation methods such as the natural spline or not-a-knot [10].

On that note, after developing the overlapping interpolation method, I learned about classic local methods such as cubic Hermite [10], cubic Bessel [10], and parabolic blending [54]. If preferred, any of those approaches can be used simply by changing the definition of W, and the open-source framework described later in this thesis has been designed to make that change straightforward.

4.2 Modeling approaches

Having described how spline points can be used as a descriptor for analog waveforms, I continue on to describe how various common types of analog models can be implemented such that they consume spline points as input and produce spline points as output. This enables spline points to be used as a sort of "universal language" between analog models, allowing them to be snapped together in any combination.

The three cases I consider are (1) static nonlinearity, (2) a system described by state-space equations, and (3) a system best described by an impulse response, such as a lossy channel. These use cases are inspired by the blocks needed to model the analog front-end of a high-speed link, but are quite general and can describe a wide range of analog behaviors.

4.2.1 Static Nonlinearity

Interestingly, modeling of static nonlinearities is the simplest of the three cases. Suppose we have a static nonlinearity $y = f(x)$. If the input spline points are $\vec{x} = \{x_0, x_1, \dots\}$, then the output spline

points are simply $\vec{y} = \{f(x_0), f(y_1), \ldots\}$.

Of course, there are limitations to this approach, since we are saying that distorting the interpolation between the spline points $\vec{x}$ is approximately the same as interpolating between distorted spline points. The approximation holds well as long as the bandwidth of the analog input $x(t)$ is not too fast compared to the spacing between spline points, Δt_h, and as long as the nonlinearity is not too hard.

The first requirement, that the spacing between spline points cannot be too large, is true across all of the spline-based modeling approaches, and is related to the Nyquist-Shannon sampling theorem [63], which implies that the maximum spacing between spline points for an analog signal of bandwidth f_{bw} is $\Delta t_h = 1/(2f_{bw})$. However, since this presumes perfect interpolation over an infinite sequence of points, in practice the spacing between spline points will need to be smaller. In addition, most analog signals do not cut off abruptly at a specific bandwidth, so this bound on Δt_h serves only as an approximate guide.

The second requirement can be interpreted in two ways. One way of thinking about it is that harder nonlinearities generate more high-frequency content, which requires closer spacing between interpolation points. Another way of looking at it is that very hard nonlinearities, such as clipping, cannot be represented accurately by a polynomial segment of any reasonable order, and therefore we should not attempt to project such behavior using spline interpolation. In that case, it would be better to start a new timestep at the nonlinear breakpoint, as suggested for CPU simulation by Jang et al. [32].

Nonetheless, for a large number of analog systems, their bandwidth is reasonable compared to the rate of digital processing, and nonlinearities are fairly soft, as they represent nonidealities, rather than fundamental behaviors. This means that we can often get away with a fairly small number of spline points to project analog behavior between digital events, without requiring many, if any, analog-only timesteps.

4.2.2 State Space Modeling

The next type of system that we consider is one that is described by state-space equations. This will allow us to model linear circuits, as well as transfer functions, which can be realized as an equivalent state-space description, such as the controllable canonical form or the observable canonical form [34].

For the purpose of the discussion here, we consider a single-input, single-output (SISO) system, since it is so commonly encountered in signal processing chains. However, the techniques discussed here can be applied to multi-input, multi-output (MIMO) systems without fundamental changes. With that in mind, the standard form of the state-space equations for a SISO system is given by [58]:

$$\dot{x}(t) = A \cdot x(t) + b \cdot u(t) \tag{4.4}$$

$$y = c \cdot x(t) + d \cdot u(t) \tag{4.5}$$

where the input signal is $u(t)$ and the output signal is $y(t)$. The vector x represents the internal state of the system; its length, s, is the number of state variables in the system. Since the system is SISO, A is $s \times s$, b is $s \times 1$, c is $1 \times s$, and d is a scalar.

Our goal in the analysis that follows is to transform the standard state-space equations into a form that takes a vector of spline points $\vec{u} = \{u(0), u(\Delta t_h), u(2\Delta t_h), \ldots\}$ as input and produces a vector of spline points $\vec{y} = \{y(0), y(\Delta t_h), y(2\Delta t_h), \ldots\}$ as output. In doing that, we also need to provide a way to update the internal state of the system after each timestep. The end result will be a set of update equations that looks similar to the standard form:

$$x(\Delta t) = \tilde{A} \cdot x(0) + \tilde{B} \cdot \vec{u} \tag{4.6}$$

$$\vec{y} = \tilde{C} \cdot x(0) + \tilde{D} \cdot \vec{u} \tag{4.7}$$

where the first equation represents the state update, and the second equation represents the output update. Derivations for these equations follow in the next two subsections.

4.2.3 State update

To derive the state update formula given an input of spline points, we start with the solution to the state-space equations in standard form [58]:

$$x(t) = e^{tA}x(0) + \int_0^t e^{(t-\tau)A} bu(\tau)\, d\tau \tag{4.8}$$

In other words, this equation represents the update to the internal state of the system after a timestep of size t, given an arbitrary input waveform $u(\tau)$.

To transform this equation into its spline points form, observe that the input waveform is defined by the input spline points u as described in a previous section:

$$u(\tau) = \sum_{j=0}^{n-2} \sum_{k=0}^{m} \sum_{i=0}^{n-1} W_{jki} u_i \left(\frac{\tau - j\Delta t_h}{\Delta t_h} \right)^k \left(H(\tau - j\Delta t_h) - H(\tau - (j+1)\Delta t_h) \right) \tag{4.9}$$

where H is the Heaviside step function. In other words, the input u is made up of $n-1$ polynomial segments of order m, whose coefficients are generated by the spline points u_i through the tensor W.

The Heaviside function may look complex, but is simply defining the windows over which each polynomial segment is valid. To make the subsequent equations easier to read, let us introduce the function $\tilde{H}_j(\tau)$ as the window over which the j-th polynomial segment is valid:

$$\tilde{H}_j(\tau) := H(\tau - j\Delta t_h) - H(\tau - (j+1)\Delta t_h) \tag{4.10}$$

When we plug the expression for the input in terms of spline points into the standard state-space

update equation, we arrive at the following:

$$x(t) = e^{tA}x(0) + \sum_{i=0}^{n-1} u_i \sum_{j=0}^{n-2} \sum_{k=0}^{m} W_{jki} \int_0^t e^{(t-\tau)A} b \left(\frac{\tau - j\Delta t_h}{\Delta t_h} \right)^k \tilde{H}_j(\tau)\, d\tau \tag{4.11}$$

$$= \tilde{A}(t)\, x(0) + \sum_{i=0}^{n-1} \tilde{b}_i(t)\, u_i \tag{4.12}$$

where:

$$\tilde{A}(t) := e^{tA} \tag{4.13}$$

$$\tilde{b}_i(t) := \sum_{j=0}^{n-2} \sum_{k=0}^{m} W_{jki} \int_0^t e^{(t-\tau)A} b \left(\frac{\tau - j\Delta t_h}{\Delta t_h} \right)^k \tilde{H}_j(\tau)\, d\tau \tag{4.14}$$

Since $\tilde{A}$ and $\tilde{b}_i$ are strictly functions of the timestep (which is allowed to vary), they can be sampled at compile-time over the range of possible timesteps, which is $[0, \Delta t_{MAX}]$. The open-source emulation framework discussed later in this thesis provides a straightforward way of building synthesizable functions this way.

Observe that the tensor W comes into play in the computation of $\tilde{b}_i$: this is one of the ways that the interpolation method, while implicit, influences model behavior. The W tensor will appear in a similar fashion in the output update formula, where it influences the computation of output spline points.

As a final note, while the expression for $\tilde{b}_i$ involves an integral, direct numerical integration is not the fastest or most accurate way of computing its value. Instead, it is better to use a direct formula for the integral of a matrix exponential times a polynomial, which is derived in Appendix A. (The open-source emulation framework discussed later in this thesis uses that approach.)

4.2.4 Output update

The output spline points are samples of the output waveform taken at times 0, Δt_h, $2\Delta t_h$, etc. In the analysis that follows, we consider the calculation of a general spline point y_p, which is the sample taken at time $p\Delta t_h$.

To proceed, plug the expression for $x(t)$ from the previous section (Equation 4.12) into the definition of the system's output, $y = du + cx$, setting $t = p\Delta t_h$:

$$y_p = du_p + c\tilde{A}(p\Delta t_h)\, x(0) + \sum_{i=0}^{n-1} c\tilde{b}_i(p\Delta t_h)\, u_i \tag{4.15}$$

$$= \tilde{c}_p x(0) + \sum_{i=0}^{n-1} \tilde{d}_{pi} u_i \tag{4.16}$$

where:

$$\tilde{c}_p = c\tilde{A}\left(p\Delta t_h\right) \tag{4.17}$$

$$\tilde{d}_{pi} = d\delta_{pi} + c\tilde{b}_i\left(p\Delta t_h\right) \tag{4.18}$$

Unlike the state update, these coefficients do not depend on the timestep size, and can therefore be computed directly at compile time.

Observe that there is a mapping from every input spline point to every output spline point through the coefficients d_{pi}, requiring n^2 multiplications. It turns out that this is not problematic for three reasons. First, n is typically small, because the shape of analog waveforms between digital events is not generally complex. Second, a large FPGA may have thousands of hardware multipliers, so these multiplications can often be performed in parallel.

The last reason has to do with the choice of interpolation method. With local interpolation, the value of a given output spline point will not depend on the values of much later input spline points. This gives rise to a roughly triangular matrix mapping input spline points to output spline points, approximately cutting the number of multiplications in half. However, since it takes $m+1$ spline points to define a polynomial of order m, that matrix is fully connected when $m+1$ spline points are used. Hence, the computational advantage of local interpolation only starts to become apparent when a larger number of spline points is used.

4.3 Impulse Response Modeling

We now consider how to model a system whose behavior is described by an impulse response. While it is possible to approximate such behavior with state-space equations, it is not necessarily efficient. To understand why, consider a system such as a transmission line, where reflections and distributed attenuation lead to behavior that is not well-described by a small number of states. In such cases, I have found it effective to build models directly from impulse responses. This not only leads to a more resource-efficient implementation, but can also be more convenient from a modeler's perspective, since they can work directly with measured data, such as a step response or S-parameter measurements, rather than having to go through an additional step to approximate the system by state-space equations.

With that in mind, suppose that we want to model a system whose impulse response is $f\left(\tau\right)$. The well-known response of such a system to a continuous-time input $u\left(t\right)$ is [53]:

$$y\left(t\right) = f\left(t\right) * u\left(t\right) = \int_{-\infty}^{\infty} f\left(t-\tau\right) \cdot u\left(\tau\right) d\tau \tag{4.19}$$

We'll first consider how to implement the behavior of this system for the specific, but important, case of piecewise-constant input, before moving on to the general case of spline input. In both cases,

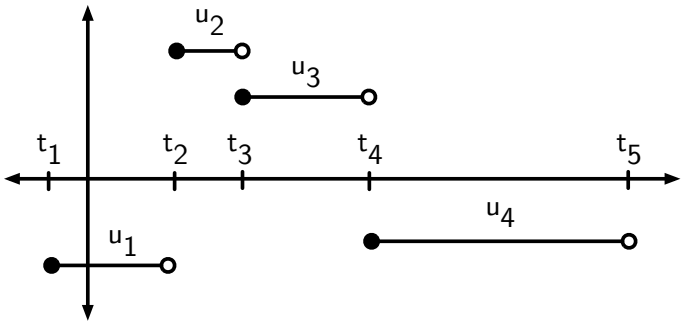


Figure 4.4: Notation used for a piecewise-constant signal: the i-th constant segment has a value u_i, held from time t_i to t_{i+1}. All times are relative to the first spline point.

the mathematical derivations simply show how to compute the output at each spline point in an FPGA-friendly manner.

4.3.1 Piecewise-Constant Input

One of the most important applications of impulse response modeling is modeling the lossy channel in a high-speed link. In that case, the channel input is typically piecewise-constant (PWC), as it represents digital data being transmitted. Even though high-speed transmitters often employ feedforward equalization, the result is still a discrete-valued, and therefore PWC, channel input.

Hence, suppose that a system's input is approximately PWC, with the i-th segment holding the value u_i from t_i to t_{i+1} (Fig. 4.4). For notational convenience, time is defined such that the first output spline point occurs at $t = 0$, with subsequent spline points at Δt_h, $2\Delta t_h$, etc.

Applying Eqn. 4.19, the p-th output spline point is given by the following expression:

$$y_p = y\left(p\Delta t_h\right) = \sum_i u_i \int_{t_i}^{t_{i+1}} f\left(p\Delta t_h - \tau\right) d\tau \tag{4.20}$$

$$= \sum_i u_i \cdot \left(\tilde{f}_p\left(t_i\right) - \tilde{f}_p\left(t_{i+1}\right)\right) \tag{4.21}$$

where $\tilde{f}_p\left(t\right)$ is a flipped and shifted version of the step response:

$$\tilde{f}_p\left(t\right) = \int_0^{p\Delta t_h - t} f\left(\tau\right) d\tau \tag{4.22}$$

Hence, Eqn. 4.21 states that each spline point is computed as a weighted sum of pulse responses, each computed as the difference of two step responses.

In practice, the summation in Eqn. 4.21 must be limited to a finite number of terms, h, which represents the length of the input history maintained by the emulator. In other words, h is the

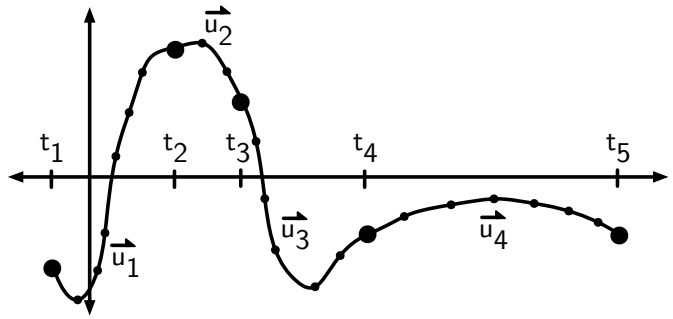


Figure 4.5: Notation used for the general case of impulse response modeling, where the input and output are both splines. The shape of the input waveform is described by a history of spline point *vectors*, with the vector $\vec{u}_i$ describing the input shape from t_i to t_{i+1}. All times are relative to the first *output* spline point.

number of PWC segments that the emulator can "remember"; older segments are considered to have a negligible effect.

The resource utilization of this modeling scheme is related to both h and the number of spline points, n. It might appear that the number of step response evaluations required would be $2nh$, since there are two function evaluations in each term in Equation 4.21. However, each distinct function evaluation appears in two successive terms. For example, $\tilde{f}_0(t_1)$ appears in the summation when $i = 0$ (where it is being subtracted) and when $i = 1$ (when it is being added). Hence, the number of function evaluations required is only nh.

4.3.2 General Case

For completeness, I now describe the general case of impulse response modeling: spline input, spline output. The result is resource-intensive, so it must be applied selectively. For example, one might use the spline-in, spline-out model for the most recent bits sent through a lossy channel, before switching to a simpler model for bits that occurred longer ago.

For this analysis, suppose that the input waveform is described by a history of spline point vectors, $\vec{u}^{(0)}, \vec{u}^{(1)}, \ldots$, with the input shape between t_i and t_{i+1} described by the vector $\vec{u}^{(i)}$ (illustrated in Fig. 4.5).

We can then compute the p-th output spline point by convolving the impulse response with each input spline, and summing up the results:

$$y_p = \sum_i \sum_{l=0}^{n-1} u_l^{(i)} \sum_{j=0}^{n-2} \sum_{k=0}^{m} W_{jkl} \int_{t_i}^{t_{i+1}} f\left(p\Delta t_h - \tau\right) \cdot \left(\frac{\tau - j\Delta t_h - t_i}{\Delta t_h}\right)^k \cdot \tilde{H}_j\left(\tau - t_i\right) d\tau \tag{4.23}$$

$$= \sum_i \sum_{l=0}^{n-1} u_l^{(i)} \cdot \tilde{f}_{pl}\left(t_i, t_{i+1}\right) \tag{4.24}$$

where the functions $\tilde{f}_{pl}$ are defined as:

$$\tilde{f}_{pl}\left(t_i, t_{i+1}\right) = \sum_{j=0}^{n-2} \sum_{k=0}^{m} W_{jkl} \int_{t_i}^{t_{i+1}} f\left(p\Delta t_h - \tau\right) \cdot \left(\frac{\tau - j\Delta t_h - t_i}{\Delta t_h}\right)^k \cdot \tilde{H}_j\left(\tau - t_i\right) d\tau \tag{4.25}$$

Equation 4.24 maps every spline point in every input feature vector to every output spline point. As a result, the number of distinct function evaluations required is n^2h, rather than nh as in the piecewise-constant case. This could quickly become expensive from a resource perspective, particularly because the functions to be evaluated are functions of two variables.

Fortunately, I have not seen a scenario where it is necessary to go to this level of modeling. One might think that the general spline-in, spline-out approach would be needed when an impulse response follows another impulse response or a state-space system, because those blocks produce spline points as their output. However, because impulse responses and state-space systems are linear, they can be composed to form a single effective impulse response. As long as the input of that impulse response is discrete-valued, the special case from Section 4.3.1 can be used. For example, in my research for ICCAD 2018 [30], I computed a number of effective channel + CTLE impulse responses for various CTLE settings in a high-speed link. The general spline-in, spline-out approach is needed only if the system being modeled includes a nonlinearity in the signal chain that *precedes* the impulse response.

4.4 High-Speed Link Experiment

In order to test the three spline-based modeling techniques just described, I applied them to the emulation of a high-speed link design (Figure 4.6), consisting of a lossy channel and three stages of continuous linear equalization (CTLE). Each CTLE was followed by a weak saturation nonlinearity to represent nonidealities in the transistor-level implementation.

4.4.1 Modeling

The lossy channel was represented using the impulse response-based technique just described, with an impulse response computed from an S-parameter dataset. The channel represented by that dataset

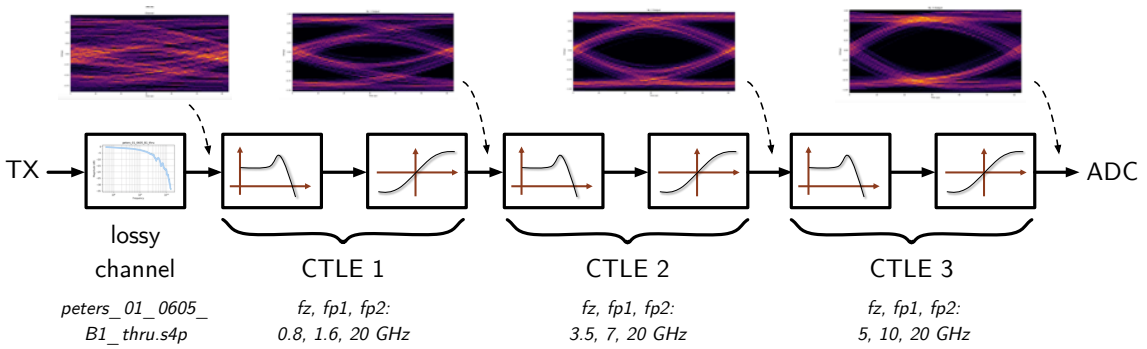


Figure 4.6: Test case for studying the spline points concept: the main analog signal path of a high-speed link, consisting of a lossy channel, followed by three CTLE stages. Each stage has a saturating nonlinearity equivalent to 1 dB loss at 1V input.

(peters_01_0605_B1_thru.s4p [56]) is the most challenging case across the six publicly available S-parameter datasets I know of, not only because its dynamics are fast, but also because it has a complex impulse response with multiple reflections.

Each CTLE was modeled as a transfer function with one zero and two poles:

$$H(s) = \frac{1 + s/\omega_z}{(1 + s/\omega_{p1}) \cdot (1 + s/\omega_{p2})} \tag{4.26}$$

The CTLE stages have their zeros and poles positioned to compensate for the attenuation caused by the channel in different frequency bands. As a result, the eye diagram for the system goes from fully closed to quite open from the beginning of the signal chain to its end.

Each saturation nonlinearity was modeled as a hyperbolic tangent function characterized by a saturation voltage v_s:

$$y = v_s \cdot \tanh x/v_s \tag{4.27}$$

All saturation nonlinearities were set to 1 dB, meaning that v_s was selected to provide 1 dB of compression at a test voltage of 1V (the TX output range was ±1V).

On the transmitter side, the unit interval (UI), or spacing between bits, was set to 62.5 ps (16 Gb/s). To exercise the variable-timestep capability of the proposed modeling approach, uniform jitter was added to the TX bit clock, causing it to vary from 90-100% of its nominal value.

4.4.2 Spline Points

In designing the emulator, two key decisions were: (1) how many spline points to use, and (2) what value to choose for the maximum timestep. Initially, I explored using a maximum timestep of 1 UI, which would allow for analog timesteps to be completely eliminated. Since bit transitions have an approximate "S" shape, I used four spline points with cubic interpolation. What I found is that,

Table 4.1: High-Speed Link Resource Utilization (Spline Approach, ZC706)

Resource	Number	Avail.	% Util.
LUT	36,077	218,600	16.5%
FF	12,649	437,200	2.9%
BRAM	215	545	39.4%
DSP	561	900	62.3%

while this coarse application of spline modeling could roughly approximate the system's behavior, an accuracy improvement of more than 10x could be achieved by shrinking the maximum timestep to 1/2 UI.

In theory, this reduces emulator throughput by 2x. However, in a full system, the effect would be less pronounced, because there are often multiple digital events during a single unit interval, such as the rising and falling edges of TX and RX clocks. As a result, the spacing between digital events should rarely exceed 1/2 UI, and hence there is little practical downside to dropping the max timestep to a half-UI.

4.4.3 Results

Using the open-source framework that will be described later in this thesis, I implemented the high-speed link analog signal chain on a Xilinx ZC706 FPGA board. This section discusses the resource utilization, accuracy, and performance achieved in that experiment.

Resource Utilization

The emulator build completed in 41 minutes using Vivado 2020.1, and the resulting resource utilization is shown in Table 4.1. The design fit comfortably on the FPGA, with low utilization of LUT and FF resources, which are the main resources that need to be reserved for emulating digital parts of a design. Block RAM (BRAM) utilization was higher at 39.4%, and DSP slices, which contain integer multipliers, were most utilized (62.3%).

For BRAM, 44% of the utilization was due to waveform memory in an Integrated Logic Analyzer (ILA) used for debugging purposes. If necessary, much of that BRAM could be freed up by multiplexing signals sent to the ILA, or by shortening its memory length.

Of the remaining BRAM slices, the vast majority (85 slices) can be attributed to the lossy channel model, where they were used to store lookup tables of the channel's step response. The calculation for each of the channel's four output spline points involved a memory of 38 piecewise-constant segments, meaning that 152 individual step response evaluations were needed. That works out to 0.56 BRAM slices per step response evaluation, which is close to the minimum allocation size (0.5 BRAM) in the Zynq-7000 series.

In terms of DSP utilization, 62% went to the channel model, 31% went to the three CTLE models, and most of the remaining DSP slices went to the saturation models. For the channel model, the utilization works out to 2.3 DSP slices per step response evaluation, which is close to what is expected, because one multiplication is needed for linear interpolation in calculating the step response, and another multiplication is needed to scale the step response by one of the PWC segments from the channel memory.

Accuracy

I characterized the accuracy of the emulator by capturing spline points generated by the emulator and comparing them to waveforms generated by a baseline simulation implemented with very fine oversampling (0.1 ps timestep). Since the emulator generates spline points and not a continuous waveform, I used the implicit interpolation method to "connect the dots" over each emulator timestep.

Although I looked at all analog waveforms in the signal chain, the most challenging waveform to model was the final output of the system (i.e., the output of the saturation nonlinearity of CTLE #3). This is because bit transition edges become faster as the analog signal propagates through the CTLE stages, since they exhibit high-pass behavior. However, that is not the only factor at play, since errors accumulate as each successive analog model is applied. Since there are seven distinct blocks in the signal chain, the last block has the most accumulated error.

With that in mind, Figure 4.7 shows a small piece of the emulator waveform from the final CTLE stage, which demonstrates that even at the most challenging point in the system, the difference between the emulator waveform and the baseline waveform is barely noticeable. To quantify the accuracy, I calculated the root mean square (RMS) error between the two waveforms, finding it to be 4.8 mVrms, or 0.24% of the full-scale signal swing. That works out to an effective number of bits, or ENOB [2], of 6.9, which is likely higher than the ENOB of the ADC that would sample the CTLE's output (e.g., S. Kim 2020 [37]).

Performance

The underlying emulator clock frequency (i.e., the rate at which emulator timesteps advanced) was 10 MHz, leading to an emulator throughput of 5 Mb/s (because of the 0.5 UI max timestep). Since the slack on the 10 MHz clock path was 9.57 ns, the emulator throughput could perhaps have been pushed as high as 5.5 Mb/s, but that is not much higher than the real performance achieved.

To put the emulator performance in perspective, RTL-level simulations of high-speed links typically top out around 10 kb/s (e.g., B. Lim 2016 [41]), so this represents a speedup of 500x over state-of-the-art CPU-based approaches. In fact, we would expect a speedup of that order, based on an early concept study I conducted in 2018 [30] that also achieved a 5 Mb/s emulation rate, albeit on a simpler system that did not include nonlinearity or the composition of multiple analog models.

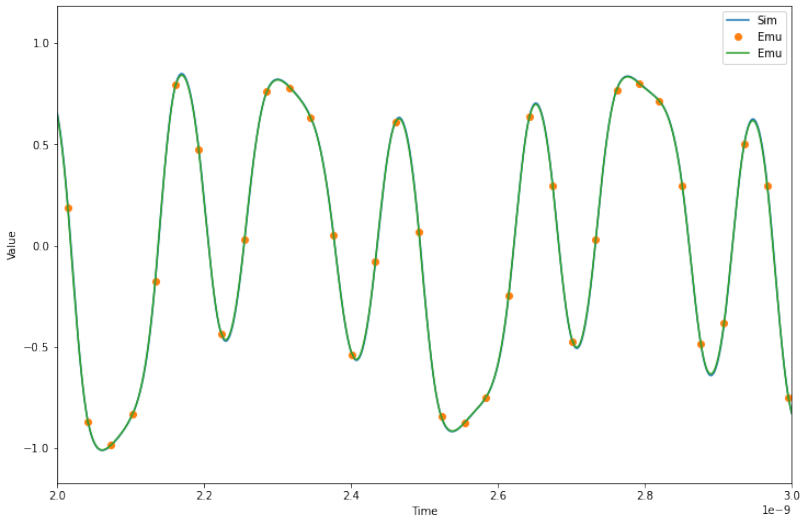


Figure 4.7: Comparison of the waveforms from the emulator and simulator, taken at the output of the final CTLE stage. Orange dots represent timesteps taken by the emulator, while the green waveform represents the implicit waveform between those points. The blue waveform, barely visible due to overlapping, is the simulation baseline.

Another way of looking at the emulator performance is to examine the speedup made possible by spline modeling, rather than the traditional oversampling approach. To do that, I determined the maximum oversampling timestep that could be taken while achieving the same level of accuracy as the spline-based model. Through computer simulation, I determined this max timestep to be 0.3 ps, meaning that the spline approach requires 104x fewer timesteps. In fact, this might be an underestimate of the advantage, because the oversampling baseline still used a pulse-response-based approach to calculating channel dynamics (CTLEs were simply oversampled, though).

Design Space Exploration

With four spline points spread over 1/2 UI, the spacing between successive spline points in this emulator was 1/6 UI. By varying the number of spline points, keeping the spacing the same, the max timestep could be changed without significantly affecting accuracy (as long as the interpolation method was also kept the same). This, in turn, enabled a tradeoff between throughput and resource utilization: increasing the number of spline points could improve throughput but would increase resource utilization.

To explore that tradeoff, I looked at two variants of the original emulator design: one using seven spline points over 1 UI and another using two spline points over 1/6 UI. The seven-point version used cubic interpolation, as with the original design, but only linear interpolation could be used for the two-point version.

Table 4.2: Design Space Exploration Summary

Spline Points	Maximum Timestep	Interpolation Method	Throughput (Mb/s)	Error (mVrms)	Build Time (min)
2	1/6 UI	Linear	1.67	30.4	21
4	1/2 UI	Cubic	5.00	4.8	41
7	1 UI	Cubic	10.00	5.3	46

Table 4.2 summarizes the results of this design space exploration: the seven-point version ran twice as fast as the four-point version, with comparable accuracy, while the two-point version ran several times slower, with noticeably worse accuracy. The accuracy degradation was likely caused by the fact that linear, rather than cubic, interpolation had to be used, since two points do not provide enough degrees of freedom to specify a higher-order polynomial. The one upside of the two-point version was that, as a much simpler design, it compiled in half the time taken by the other designs.

While all three designs fit on the ZC706 board, resource utilization was a strong function of the number of spline points. Fig. 4.8 shows a breakdown of how each FPGA resource was used by each design. For the two- and four-point versions, plenty of resources were left over for emulating other blocks, but the seven-point version completely exhausted all DSP slices on the FPGA. In fact, it needed so many multipliers that some ended up being built from LUTs, which partly explains the sharp jump in LUT utilization from four to seven spline points.

Since the seven-point version has twice the throughput as the four-point version, it might make sense to use that more resource-intensive model on a larger FPGA board. However, as pointed out earlier, digital events are not generally spaced farther apart than 1/2 UI, so the performance impact would probably be smaller in practice. Considering the poor accuracy of the two-point version, it appears that the "sweet spot" for emulating high-speed links is four spline points spaced over 1/2 UI.

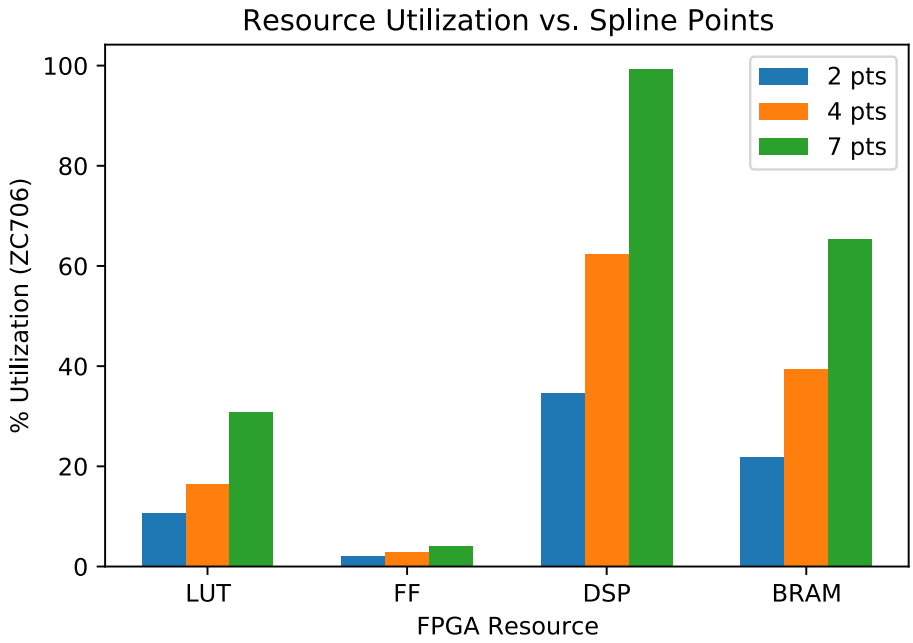


Figure 4.8: Resource utilization of three spline-based emulator designs: one using two spline points, another using four spline points, and a third using seven spline points.

Chapter 5

Dial E for Emulation

Through several years of working on AMS emulation, I have found that it is not yet used broadly in "deep" way (i.e., incorporating detailed models of analog behavior). One of the key reasons for this is that AMS emulation requires an unusual combination of skills, namely experience in analog modeling and experience in FPGA design. In many chip companies, analog modeling experts work within simulation-based verification teams, while FPGA design experts reside in application or test teams. Hence, the two skillsets are typically separated in the company hierarchy.

That said, knowing about analog modeling and FPGA design is only necessary, but not sufficient, for building AMS emulators effectively. This is because the emulation process is typically ad-hoc, relying on hand-built synthesizable analog models and emulator infrastructure. The result is that, even with the right background, building AMS emulators is a time-consuming, error-prone process. That fact, in combination with the expertise barrier, means that many companies do not realize the full potential of AMS emulation.

Having made that observation, I set out to develop a complete flow for building AMS emulators that would speed up the process and reduce barriers to entry. This framework is the complement of my research about efficient AMS emulation, in that it provides a practical way of realizing those techniques.

Figure 5.1 depicts the framework, which consists of three tools: **msdsl** [28], **svreal** [29], and **anasymod** [60]. All three tools will be described in detail in the subsequent sections, but here is a brief description of how the entire flow works: users write models for analog blocks in Python, using **msdsl**, which are compiled in synthesizable Verilog models, leveraging the **svreal** for real-number operations. Once these synthesizable models are swapped into the DUT, the result is a fully synthesizable model of the DUT (since its digital blocks are already synthesizable). At that point, **anasymod** wraps control infrastructure around the DUT, generates a bitstream, and programs an FPGA board.

In dividing the framework up into three tools, my intent was to provide a degree of modularity,

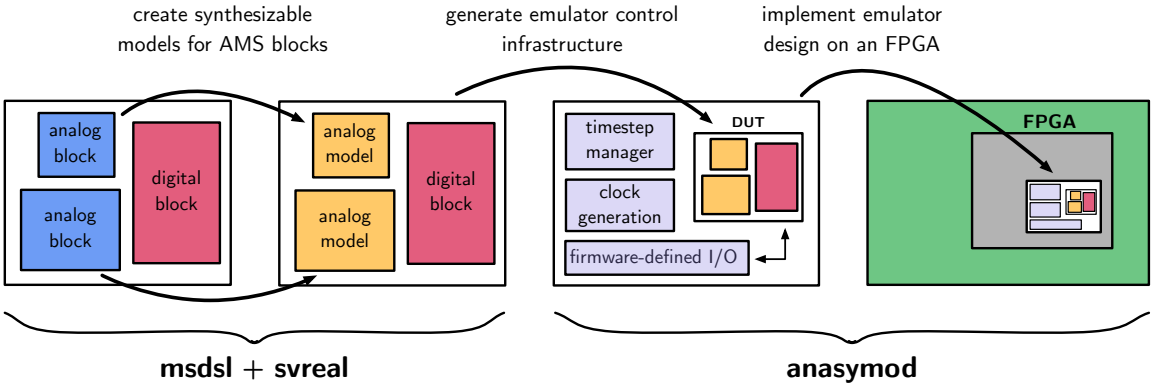


Figure 5.1: Overview of the AMS emulation framework. Analog models are described in Python and compiled into synthesizable SystemVerilog using **msdsl**, leveraging **svreal** to implement real-number operations. **anasymod** then wraps emulator control infrastructure around the DUT and automates EDA tools to produce an FPGA bitstream.

so that users could leverage each tool in a standalone fashion. For example, **svreal** can be used by itself as a library for synthesizable fixed- and floating-point operations. In addition, the synthesizable models produced by **msdsl** could be used in a commercial emulator, rather than targeting an FPGA board through **anasymod**. Finally, **anasymod** can be used for emulating completely digital designs, since it provides abstractions for FPGA tools. Judging from GitHub feedback, it appears that some members of the public have in fact been using one tool, **svreal**, in a standalone fashion.

On that note, all three tools are freely available as open source, and can be installed either from GitHub or by using the Python package manager (pip). I chose to go the open-source route for several reasons. First and foremost, it makes the research more accessible to the public, since there are no cost or licensing barriers. That was particularly important to me, because innovation in the chip design industry has historically been hampered by those limitations. But there are additional benefits to the project being open-source, such as getting feedback from a broader user base, and unlimited regression testing resources from GitHub.

I was fortunate to count Infineon as a collaborator throughout the development of the framework. Infineon sent visiting researchers to Stanford in mid-2018, just as I was starting to think about tools for improving the productivity of AMS emulation. That goal resonated with Gabriel Rutsch, one of the visiting researchers, and he invited me to Munich for three months at the beginning of 2019 to work on the framework in the context of Infineon's commercial applications. At that point, we split the framework into three tools, with Gabriel leading **anasymod** and me leading **msdsl** and **svreal**. The three sections that follow provide a detailed description of those tools.

Listing 5.1: Basic **svreal** example.

```
1  `include "svreal.sv"
2  `MAKE_REAL(a, 5.0);
3  `MAKE_GENERIC_REAL(b, 10.0);
4  `ADD_REAL(a, b, c);
5  initial begin
6      `FORCE_REAL(1.23, a);
7      `FORCE_REAL(4.56, b);
8      #(1ns);
9      `PRINT_REAL(c);
10 end
```

5.1 svreal

svreal is at the lowest level of the emulation framework, providing a synthesizable Verilog library for real-number number operations that is used by **msdsl**'s Verilog code generator. The main focus of the **svreal** library is fixed-point support, but all operations have synthesizable floating-point implementations, too.

Like the fixed-point system in Chisel [7, 43], **svreal** uses an interval-based approach to automated fixed-point formatting and provides a method to switch between fixed- and floating-point representations. However, since the purpose of **svreal** is to facilitate the integration of AMS models into existing RTL, rather than being used to generate entire chip designs, it differs in two ways. First, its fixed-point formats are parameterized in SystemVerilog, which simplifies the process of passing real-valued signals between various models. Second, it aims to be both human-readable and human-writable at the SystemVerilog level, which aids in model debugging and makes it easier to construct the glue logic that is often needed for emulation.

5.1.1 Basic Usage

Listing 5.1 shows a basic example of how the **svreal** library is used. Briefly, this code snippet creates fixed-point signals `a` and `b`, adds them, and stores the result in `c`. In the `initial` block (which is not synthesizable), the values of `a` and `b` are set to "1.23" and "4.56," respectively. Following that, the value of `c` is printed. Hence, the value printed will be "5.79."

Looking at the code at a more detailed level, **svreal** is included as a header file on line 1. That header file contains the full **svreal** implementation as a means to simplify integration; there are no additional sources that have to be specified.

On the next line, a macro is used to declare a real-number variable `a` whose range is ± 5; the range information is used in determining fixed-point formats, as will be described shortly. In a similar fashion, the real number `b` is declared with range ± 10 on line 3.

On line 4, "a" and "b" are added and stored in a new variable called "c." Since the ranges of "a" and "b" are specified, **svreal** automatically calculates the range of "c" to be ±15. Under the hood, **svreal** keeps track of range information using parameters that are declared and accessed when macros such as `MAKE` and `ADD` are invoked.

One may notice that throughout this entire code example, no fixed-point formatting details were explicitly specified. In fact, **svreal** is not limited to fixed-point operation; if the user specifies a compile-time option called `HARD_FLOAT`, all real-valued signals will switch to floating-point, with operations implemented by the synthesizable Berkeley HardFloat library [25]. In a similar fashion, defining `FLOAT_REAL` will cause the SystemVerilog **real** type to be used.

This is one of the key benefits of implementing **svreal** as a macro library, because it provides significant flexibility when changing the underlying real-number format. Since macros can be fragile, however, I did an extensive exploration of whether modern SystemVerilog features, such as parameterized interfaces, could be used to achieve a similar effect. Unfortunately, due to limitations in the SystemVerilog standard itself, as well as bugs in simulation and synthesis tools, I found macros to be the more reliable approach.

5.1.2 Fixed-Point Format

When **svreal** is used in fixed-point mode (the default), it represents a real number, r, as a twos-complement signed integer s of width w, with an implicit exponent p:

$$r \approx s \cdot 2^p \tag{5.1}$$

For example, the real number represented by the 8-bit signal `0xC8` and implicit exponent -7 is $r = -56 \cdot 2^{-7} = -0.4375$.

svreal can automatically specify most widths and exponents at compile-time, which is why the previous code example did not contain that information. As with the range information, widths and exponents are stored in parameters that are created under the hood by **svreal** macros.

In determining fixed-point widths, **svreal** picks a default based on the type of signal: it sets multiplicative constants to a width of `SHORT_WIDTH_REAL` and all other signals `LONG_WIDTH_REAL`. The reason for this distinction has to do with the architecture of DSP multipliers on FPGAs, whose inputs are generally of unequal length. For example, in the Xilinx Zynq 7000 series, the DSP multiplier has one input that is 18 bits wide, while the other is 25 bits wide [74].

Mapping multiplicative constants to the shorter multiplier input, while using the longer input for real number variables, means that a multiplication-by-constant operation will consume exactly one DSP slice. The extra bits of the longer input are given to the variable since it may benefit from extra dynamic range. Representing constants with the shorter input does not significantly degrade constant accuracy, since even the 18-bit short input in the Zynq-7000 architecture allows constants

to be represented with better than 10 ppm accuracy.

This optimization for constant multiplication is important for two reasons: first, DSP multipliers are a relatively limited resource on FPGAs (typically no more than a few thousand), and second, multiplication by a constant is a very common operation in modeling analog blocks. The reason is that many analog behaviors can be characterized by linear dynamics, which, as shown in Section 4.2.2, can be accurately discretized without cross-terms between variables.

Once **svreal** has determined the fixed-point width, w, it moves on to the calculation of the implicit exponent, p. Its goal is to maximize precision while avoiding overflow. More specifically, the real-number range that can be represented is a function of w and p:

$$-2^{w-1} \cdot 2^p \leq r \leq \left(2^{w-1}-1\right) \cdot 2^p \tag{5.2}$$

svreal aims to make p as small as possible (i.e., highest resolution), while still covering a range at least as large as the range of r (i.e., avoiding overflow). If we denote the range of r as $\pm R$, then the optimal value of p is given by:

$$p = \left\lceil \log_2 \frac{R}{2^{w-1}-1} \right\rceil \tag{5.3}$$

As an example, consider the variable `a` from Listing 5.1. Since the width of `a` is not specified by the user, and it is not a multiplicative constant, it defaults to `LONG_WIDTH_REAL` (25). The range of `a` is set to ± 5, so the exponent is calculated to be $p = \left\lceil \log_2 \frac{5}{2^{25-1}-1} \right\rceil = -21$. Hence, the format for `a` is a bit oversized, since it could hold values up to about ± 8, even though its range is only ± 5. However, if we were to have picked the next smallest exponent, $p = -22$, the range of `a` would have only been about ± 4, which would be too small.

Since the maximum range of a signal is determined by its width and exponent, one might wonder why it is beneficial to separately keep track of the signal's true range, which can be up to the maximum range. To see why, suppose that the formatting of `c` were determined by inferring the range of `a` and `b` and from their widths (both 25) and exponents (-21 and -20, respectively), rather than keeping track of their ranges separately. We would then infer the range of `a` to be ± 8 and the range of `b` to be ± 16, meaning that the range of `c` is ± 24. From Equation 5.3, that would cause `c` to be assigned an exponent of -19, when in reality -20 could be used without overflow, because the true range of `c` is ± 15. In other words, we can double the precision of `c` with no change in its width, simply by keeping track of ranges more carefully at compile time.

The problem would become even more accentuated through repeated computations, because with an exponent of -19, the range `c` would be inferred to be ± 32 by subsequent calculations, which is more than twice its true range. Hence, range inaccuracies can build quickly if the range of real-number signals is not tracked separately from their widths and exponents.

Viewed broadly, **svreal**'s fixed-point auto-formatting strategy means that the range of a signal directly controls its resolution; a signal occupying a small range will use a finer resolution than a

signal covering a large range. Our simple analysis has an obvious drawback: signal ranges may grow unrealistically through repeated arithmetic operations, reducing resolution more than truly necessary. This is the reason that other systems do more sophisticated analyses to better estimate variable range (e.g., running floating-point simulations of an algorithm to estimate signal ranges [38]). However, we have found that this is rarely an issue for two reasons. First, the default fixed-point width (i.e., `LONG_WIDTH_REAL`) is large enough that range overestimates of even 100x do not generally have much impact on emulating system-level behaviors. Second, chains of arithmetic operations tend to be fairly short in AMS models, so there is not much opportunity for overestimates to build up.

5.1.3 Real-Number Constants

Constants, such as those on lines 6 and 7 of Listing 5.1, are automatically converted to the underlying real-number format. For example, when a fixed-point format is used, the real-number value r of a constant is converted to a fixed-point representation with exponent p with the following equation:

$$s = \text{round}\left(r \cdot 2^{-p}\right) \tag{5.4}$$

where p is computed by evaluating (5.3) with $R = |r|$.

Similarly, when `HARD_FLOAT` is defined, the real-number constant is automatically converted to the Berkeley HardFloat "recoded" format, and when `FLOAT_REAL` is defined, real-number constants are used directly.

5.1.4 Debugging

Debugging accuracy problems in a fixed-point model can be hard, because it is not always clear whether a problem is caused by a numerical issue, such as insufficient precision, or another issue, such as delay alignment in a pipeline. To narrow the search, users can define the `FLOAT_REAL` flag, which switches the **svreal** real-number type to the native SystemVerilog **real** type. (In almost all cases, no additional changes to the user's SystemVerilog code are needed to apply that flag.)

If the problem goes away after applying the `FLOAT_REAL` flag, then the problem is likely a fixed-point numerical issue. The remaining question at that point is whether the bug is due to overflow or a precision issue. To check for overflow, a user can specify both `FLOAT_REAL` and `RANGE_ASSERTIONS`, which attaches checkers to all real-number values to ensure they stay within their specified ranges. If there is an overflow, **svreal** throws an error that can be traced back to the problematic signal.

If there are no overflow errors, then there is likely a fixed-point precision issue. Users can test that theory by removing the `FLOAT_REAL` flag (i.e., going back to fixed-point mode) and then increasing `LONG_WIDTH_REAL` and/or `SHORT_WIDTH_REAL`. If increasing `SHORT_WIDTH_REAL` helps, then one or more multiplicative constants need to be higher-resolution. Otherwise, one or more fixed-point signals or additive constants need more resolution.

Declaration and Memory	Assignment and Unary	Arithmetic	Conditional and Comparison	Type Conversion
MAKE_REAL	ASSIGN_REAL	MIN_REAL	ITE_REAL	REAL_TO_INT
MAKE_CONST_REAL	ASSIGN_CONST_REAL	MAX_REAL	LT_REAL	INT_TO_REAL
DFF_REAL	ABS_REAL	ADD_REAL	LE_REAL	
	NEGATE_REAL	SUB_REAL	GT_REAL	
	COMPRESS_UINT	MUL_REAL	GE_REAL	
		ADD_CONST_REAL	EQ_REAL	
		MUL_CONST_REAL	NE_REAL	

Table 5.1: Common **svreal** operations.

It is worth briefly explaining why `FLOAT_REAL` should be used in conjunction with `RANGE_ASSERTIONS`. Going back to Listing 5.1, suppose that we try to assign the value "12.34" to `a`, which is far outside its specified range of ±5. Since the exponent of `a` is -21, the fixed-point representation of 12.34 should be 25,878,856. However, this overflows the 25-bit signed integer format and will be interpreted as -7,675,576. That, in turn, leads to a real-number interpretation of -3.66, which has not only the wrong value, but also the wrong sign.

Unfortunately, an automated range checker cannot know that an overflow has occurred, because the value of `a` is in fact within its specified range (±5). In other words, overflows cannot be reliably detected when `svreal` is in fixed-point mode, so `RANGE_ASSERTIONS` should always be used with the `FLOAT_REAL` flag. While the `HARD_FLOAT` flag could be used for a similar purpose, `FLOAT_REAL` is generally better for debugging because it simulates faster than `HARD_FLOAT` (at least 10x faster).

5.1.5 Operations Supported

We next take a deeper look at some of the operations supported by **svreal**. The library supports over 20 distinct operations, summarized in Table 5.1.

Declaring signals

The basic macro for declaring a real-number signal is `MAKE_REAL(name, range)`. In fixed-point mode, the signal width will default to `LONG_WIDTH_REAL`, and the implicit exponent will be computed as described previously. Under the hood, **svreal** stores the width, exponent, and range as parameters for use by subsequent operations. Several variants of `MAKE_REAL` exist, such as `MAKE_CONST_REAL` for declaring real-number constants, and `MAKE_GENERIC_REAL` for declaring real-number signals with a non-default width.

Assigning signals

The basic macro for moving around real-number data is `ASSIGN_REAL(from, to)`. In general, this macro must be used instead of the built-in Verilog **assign** command, because `from` and `to` may have

different formats (i.e., different width and/or exponent). **svreal** automatically performs realignment as necessary. In particular, if `ASSIGN_REAL(a, b)` is invoked, with `a` and `b` having exponents p_a and p_b, respectively, then `b` is assigned with an arithmetic right shift:

$$b := a \ggg (p_b - p_a) \tag{5.5}$$

In that equation (and others that follow in this section), an arithmetic right shift by a negative amount is interpreted as a left shift. For example, if $p_b < p_a$ (i.e., `b` has a higher resolution than `a`), then `a` is left-shifted prior to assignment to `b`.

Addition

The basic addition operation is `ADD_REAL(a, b, c)`, which assigns the sum of `a` and `b` to a new auto-formatted signal, `c`. In fixed-point mode, the strategy used by **svreal** is to re-align `a` and `b` to the implicit exponent of `c` prior to addition:

$$c := (a \ggg (p_c - p_a)) + (b \ggg (p_c - p_b)) \tag{5.6}$$

Another option would be to re-align `a` and `b` to the more precise exponent, add them, and then re-align the result to `c`. This would produce a slightly more accurate result, but at the expense of higher resource utilization. Since most real-number signals have excess resolution (25 bits) to begin with, this optimization does not seem to be worthwhile in practice.

Multiplication

In a similar fashion, the macro for **svreal** multiplication is `MUL_REAL(a, b, c)`. As with addition, the inputs `a`, `b`, and `c` will generally have different formats, so realignment is necessary. The formula used by **svreal** is:

$$c := (a * b) \ggg (p_c - p_a - p_b) \tag{5.7}$$

Comparison

svreal has various operations for comparing two real numbers, `a` and `b`, and storing the Boolean result in a 1-bit signal, `c`. For example, one can check if `a` is less than `b` with the macro `LT_REAL(a, b, c)`. Since `a` and `b` can have arbitrary formats, alignment is again an important issue. In this case, unlike with addition, I found it worthwhile to spend the extra hardware to align `a` and `b` to the more precise exponent, i.e.

$$(a \ll (p_a - \min(p_a, p_b))) \quad \text{compared to} \quad (b \ll (p_b - \min(p_a, p_b))) \tag{5.8}$$

This ensures reasonable behavior when two values are close. If we instead re-aligned `a` and `b`

to the *less* precise exponent, the two values may become equal, even if they are simply close, but unequal. That, in turn, can make it difficult to implement transition points in behavioral models.

Type conversion

Sometimes it is necessary to convert a signed integer into an **svreal** type or vice versa. Within an AMS modeling context, for example, these types of conversions are needed in A/D and D/A interfaces.

To convert a signed integer to an **svreal** type, use the macro `INT_TO_REAL(in, width, out)`. The second argument is the width of the signed integer being converted; I explored extracting its width using the SystemVerilog functions `$size` and `$bits`, but found that those functions were unfortunately not implemented consistently across simulation and synthesis tools. As a result, it is left up to the user to manually provide the width.

In a similar fashion, the macro `REAL_TO_INT(in, width, out)` converts the real-number input `in` to a signed integer `out` of width `width`.

Pseudo-logarithmic compression

svreal contains a special macro, `COMPRESS_UINT(x)` that computes a "pseudo-logarithm" of an unsigned integer `x`:

$$\text{plog}_2(x) = \lfloor \log_2 x \rfloor + x/2^{\lfloor \log_2 x \rfloor} \tag{5.9}$$

$$\approx (\log_2 x) + 1 \tag{5.10}$$

This is very useful for implementing highly nonlinear functions such as division or inverse cumulative distribution functions (CDFs), as described later.

In fixed-point mode, **svreal** implements the pseudo-logarithm by first measuring the "width" of `x` (i.e., the minimum number of bits needed to represent it). For example, the width of `0b0110` is 3, the width of `0b0001` is 1, and the width of `0b0000` is 0. This measurement is equal to $1 + \lfloor \log_2 x \rfloor$ (and zero when `x` is zero), but can be implemented efficiently in hardware as a priority encoder. The width, which we denote m, is then added to the value of `x`, with its top bit clipped and interpreted as a fractional number (an operation that can be implemented with a single left shift). The result is an implementation of the desired pseudo-logarithm function:

$$\text{plog}_2(x) = m + \left(\frac{x}{2^{m-1}} - 1\right) \tag{5.11}$$

$$= (1 + \lfloor \log_2 x \rfloor) + \left(\frac{x}{2^{\lfloor log_2 x \rfloor}} - 1\right) \tag{5.12}$$

$$= \lfloor \log_2 x \rfloor + x/2^{\lfloor \log_2 x \rfloor} \tag{5.13}$$

Listing 5.2: Using the `INTO` form of an **svreal** macro to assign a result to an existing signal.

```
1 `MAKE_REAL(a, 10.0); // i.e., +/- 10
2 `MAKE_REAL(b, 21.0); // i.e., +/- 21
3 `MAKE_REAL(c, 32.0); // i.e., +/- 32
4 `ADD_INTO_REAL(a, b, c);
```

In synthesizable floating-point mode (`HARD_FLOAT`), the pseudo-logarithm is simply the sum of the exponent and fractional part of the floating-point representation. More precisely, a floating-point number is represented as $x = 2^e \cdot (1 + f)$, where e is the exponent and f is the fractional part, which is in the interval $[0, 1)$.[1] As a result, we have:

$$\lfloor \log_2 x \rfloor = e \tag{5.14}$$

$$x / 2^{\lfloor \log_2 x \rfloor} = 1 + f \tag{5.15}$$

and therefore $\text{plog}_2 x = e + f + 1$.

The takeaway from this analysis is that the pseudo-logarithm provides a hardware-efficient way of compressing numbers that works well for both fixed- and float-point formats. It turns out that it is also invertible, which is a very convenient feature for implementing functions using the pseudo-logarithm. The inverse of the pseudo-logarithm is:

$$\text{plog}_2^{-1}(x) = \left(2^{\lfloor x \rfloor - 1}\right) \cdot (1 + x - \lfloor x \rfloor) \tag{5.16}$$

We'll see how that inverse can be applied in the **msdsl** section, where the pseudo-logarithm is used to accurately implement the inverse CDF of the Gaussian distribution.

Assigning to existing signals

Most **svreal** operations have an alternate form that allows the user to assign the result of an operation to an existing real-number signal. This is indicated by the word `INTO` in the macro name. For example, suppose that we have defined three signals, `a`, `b`, and `c`, and want to assign the sum of `a` and `b` into `c`, without having **svreal** declare and auto-format the signal `c`.

We can do this using a special form of the addition operation, `ADD_INTO_REAL`, as illustrated in Listing 5.2. This special form of the `ADD` operation will not declare a new signal `c`, but instead will assign the result of the addition to the existing signal called `c` (performing alignment shifts as necessary, as before).

In this case, there is a risk that `c` may have been declared with insufficient range to hold the result. As mentioned before, this can be debugged using the `FLOAT_REAL` and `RANGE_ASSERTIONS`

[1] The exponent is typically biased, but that detail does not substantially affect the deriviation that follows.

Listing 5.3: Using the `GENERIC` form of an **svreal** macro to specify the resolution of an arithmetic operation.

```
1 `MAKE_REAL(a, 10.0); // i.e., +/- 10
2 `MAKE_REAL(b, 21.0); // i.e., +/- 21
3 `MUL_REAL_GENERIC(a, b, c, 40);
```

flags, but given the risk of overflow, why use the `INTO` form at all? The most common reason is to assign a signal that appears in the I/O list of the module. Alternatively, one may want to fine-tune the range or resolution of a signal. However, a better way to accomplish that is to use the `GENERIC` form of operations, as described next.

Specifying resolution

Most operations have an alternate form ending with `GENERIC` that allows the user to specify the width of an arithmetic result. Since the range of the operation is determined automatically, the specified width effectively controls the resolution of the result. As an example, suppose we want to multiply two signals, but represent the output with more precision than the default. That can be done with a variant of `MUL_REAL` called `MUL_GENERIC_REAL`, as illustrated in Listing 5.3.

In that code example, the width of the result, `c`, is given the custom value "40," rather than defaulting to `LONG_WIDTH_REAL`. The range of `c` is still determined automatically, meaning that if more precision is required, the user can simply increase the width, rather than having to simultaneously adjust the width and exponent.

5.1.6 Hierarchy

Since Verilog parameters are used to store fixed-point formatting information, some care must be taken when passing **svreal** signals through a hierarchy to ensure that information is not lost. This is not strictly necessary when using `HARD_FLOAT` or `FLOAT_REAL`, but is still good practice because it makes it easier to switch to fixed-point.

Consider the example in Listing 5.4, where an outer block instantiates a module that multiplies two signals to produce an output. First, notice that the parameters and I/O list of the inner module, which has fixed-point I/O, has to be declared in a certain way. Every fixed-point number in the I/O list (regardless of whether it is an input or an output) needs to have a corresponding `DECL_REAL` statement in the parameter list for the module. This declares all of the parameters needed for that fixed-point signal: its width, exponent, and range. Then, in the I/O list for the module, fixed-point inputs and outputs should be declared using `INPUT_REAL` and `OUTPUT_REAL`, respectively.

Going up one level to the outer block, observe that a special macro `PASS_REAL` is needed to pass parameter information for the fixed-point signals `a`, `b`, and `c` into the inner module. The syntax

Listing 5.4: Instantiating a module with **svreal** I/O.

```
1  `include "svreal.sv"
2  module inner #(
3      `DECL_REAL(in0),
4      `DECL_REAL(in1),
5      `DECL_REAL(out)
6  ) (
7      `INPUT_REAL(in0),
8      `INPUT_REAL(in1),
9      `OUTPUT_REAL(out)
10 );
11     `MUL_INTO_REAL(in0, in1, out);
12 endmodule
13 module outer;
14     `MAKE_REAL(a, 10.0); // i.e., +/- 10
15     `MAKE_REAL(b, 21.0); // i.e., +/- 21
16     `MAKE_REAL(c, 32.0); // i.e., +/- 32
17
18     inner #(
19         `PASS_REAL(in0, a),
20         `PASS_REAL(in1, b),
21         `PASS_REAL(out, c)
22     ) inner_i (
23         .in0(a),
24         .in1(b),
25         .out(c)
26     );
27 endmodule
```

of `PASS_REAL` is meant to mimic using dot-notation to connect signals to a module instance; that is, the name of the port on the inner module comes first, followed by the name of the local signal. Finally, note that fixed-point signals are wired up in the I/O list using standard dot notation.

Of course, it would be more convenient if the range and formatting information could be automatically bundled along with signals, rather than having to use `PASS_REAL` separately. As an experimental feature, **svreal** does offer a method for doing that through the use of a parameterized interface, but I have found that this particular SystemVerilog feature is unreliably handled by FPGA synthesis tools.

5.2 msdsl

msdsl is a tool for describing the behaviors of AMS blocks in Python, leveraging the **svreal** library to produce synthesizable HDL as its output. In comparison to more general-purpose HDL generators

such as Xilinx System Generator [76], Xilinx High-Level Synthesis [75], and Simulink HDL Coder [45], it provides high-level abstractions that are geared towards AMS modeling. In addition, since **msdsl** is implemented in Python, it makes it straightforward for users to leverage open-source libraries such as SciPy [70] and NumPy [51], and to reuse modeling code through object-oriented programming.

This section starts with a description of the basic flow through **msdsl**, using one of the simplest analog circuits as an example: an RC filter. In general, the examples in this section are made as minimal as possible to illustrate the tool's capabilities. More advanced, complete applications are covered later in this thesis (Chapter 6).

msdsl has two key parts: low-level "building blocks" for constructing models (e.g., arithmetic operations, arbitrary functions, and pseudorandom noise) and higher-level "input formats" for describing analog circuits, which include a SPICE-style netlist and a system of continuous-time differential equations. Following the preliminary RC filter example, I provide an in-depth discussion of those capabilities.

5.2.1 Basic Flow

Suppose that we wish to create a fixed-timestep, manually-discretized model of an RC filter. We would start by writing down the continuous-time dynamics of the circuit:

$$C \cdot \frac{dy}{dt} = \frac{x - y}{R} \tag{5.17}$$

where the input voltage is x, the output voltage is y, the resistance is R, and the capacitance is C. The well-known solution to that equation over an interval $[t, t + \Delta t]$ in which x is constant is given by the following expression:

$$y\left(t + \Delta t\right) = \alpha \cdot y\left(t\right) + \left(1 - \alpha\right) \cdot x\left(t\right) \tag{5.18}$$

where $\alpha = e^{-\Delta t/(RC)}$.

This manually-derived result can be used in an **msdsl** model as shown in Listing 5.5. The code starts by creating a `MixedSignalModel` instance, followed by a declaration of the analog input `x` and output `y`. These variables are `Signal` objects that can be manipulated with arithmetic operators, as a result of operating overloading within **msdsl**. That capability is highlighted in the implementation of Equation 5.18 on line 8.

Also on line 8, the value of `y` on the next emulation cycle is defined, meaning that `y` is a state variable. As such, it must have an associated reset signal, to set its initial value, and a clock signal, to control when it is updated. When not specified, those signals default to the macros `RST_MSDSL` and `CLK_MSDSL`. Users can define those macros to point to global clock and reset signals for all analog models, or, if they prefer, can specify the clock and reset signals via optional arguments to `set_next_cycle`. The initial value of analog state variables defaults to "0," but can be specified

Listing 5.5: Fixed-timestep modeling in **msdsl**.

```
1 from msdsl import MixedSignalModel, VerilogGenerator
2 from math import exp
3 r, c, dt = 1e3, 1e-9, 0.1e-6
4 m = MixedSignalModel('rc')
5 x = m.add_analog_input('x')
6 y = m.add_analog_output('y')
7 a = exp(-dt/(r*c))
8 m.set_next_cycle(y, a*y + (1-a)*x)
9 m.compile_and_print(VerilogGenerator())
```

when the signal is defined. In this case, since `y` is a model output, its initial value would be specified in the `add_analog_output` declaration.

The final line generates and prints a synthesizable Verilog implementation, using a `VerilogGenerator` instance. Internally, **msdsl** uses a representation of mixed-signal models that aims to be agnostic to the final implementation language. The `compile_and_print` command accepts any object that implements the `CodeGenerator` interface to convert that internal model representation to concrete HDL. At the moment, however, only the `VerilogGenerator` implementation is provided in **msdsl**. In the future, it might be interesting to provide code generators for HLS C/C++, or even hardware generators such as **magma** [24].

The HDL generated by **msdsl** is shown in Listing 5.6. As can be seen, it leverages **svreal** for real-number operations. Briefly, `x` and `y` are scaled by constants, producing internal signals `tmp0` and `tmp1`, which are summed to produce `tmp2`. Finally, `tmp2` is written to the output `y` with one cycle of delay; the clock and reset signals for that delay are `CLK_MSDSL` and `RST_MSDSL`.

One of the key details of this code sample is that **msdsl** uses the `MUL_CONST_REAL` operation to perform multiplication by a constant, rather than the generic `MUL_REAL` operation. This is important, because `MUL_CONST_REAL` is designed to consume only one DSP slice through careful selection of the widths of the multiplicands. **msdsl** uses a number of other optimizations along those lines to reduce resource utilization and improve code readability, such as compile-time simplification of arithmetic expressions, and generation of ROM files for large lookup tables.

5.2.2 Building Blocks

Having shown the basic flow through **msdsl**, I now provide an in-depth look into the tool's low-level "building blocks": declaring analog and digital signals, assigning values to signals, manipulating signals, and converting between analog and digital data types. I also describe two advanced building blocks: arbitrary functions and pseudorandom noise. Taken together, these building blocks provide the basis for the higher-level modeling abstractions described later. They are also useful in their

Listing 5.6: HDL generated by **msdsl** for the fixed-timestep RC filter model, leveraging the **svreal** library.

```
// Model generated on 2021-04-08 15:30:42.761154

`timescale 1ns/1ps

`include "svreal.sv"
`include "msdsl.sv"

`default_nettype none

module rc #(
    `DECL_REAL(x),
    `DECL_REAL(y)
) (
    `INPUT_REAL(x),
    `OUTPUT_REAL(y)
);
    // Assign signal: y
    `MUL_CONST_REAL(0.9048374180359596, y, tmp0);
    `MUL_CONST_REAL(0.09516258196404037, x, tmp1);
    `ADD_REAL(tmp0, tmp1, tmp2);
    `DFF_INTO_REAL(tmp2, y, `RST_MSDSL, `CLK_MSDSL, 1'b1, 0);
endmodule

`default_nettype wire
```

own right as a platform for experimenting with AMS emulation techniques.

Signals

The RC filter example only used analog signals for external I/O, but **msdsl** signals can be declared as analog or digital, and internal or external. Digital signals default to 1-bit, unsigned, but their width and signedness can be specified. Analog signals are real values with a specified range that is used to compute fixed-point formats, as described in the **svreal** section. It is generally only necessary to specify ranges for model I/O and state variables, since **svreal** can automatically figure out the rest.

Several examples for signal declarations are shown in Listing 5.7. Note the optional `init` argument for state variables; although not shown in the code sample, that argument is also available for digital signals.

Listing 5.7: Signal declarations in **msdsl**.

```
1 from msdsl import MixedSignalModel
2 m = MixedSignalModel('model')
3 a = m.add_analog_input('a')
4 b = m.add_digital_output('b', signed=True, width=8)
5 c = m.add_analog_state('c', init=1.23, range_=4.56)
6 d = m.add_digital_signal('d', width=4)
```

Listing 5.8: Signal assignments in **msdsl**.

```
1 from msdsl import MixedSignalModel
2 m = MixedSignalModel('model')
3 a = m.add_analog_input('a')
4 b = m.add_analog_output('b')
5 c = m.add_analog_state('c', init=1.23, range_=4.56)
6 m.set_next_cycle(c, 0.9*c + 0.1*a)
7 d = m.set_this_cycle('d', 6.78*c + 7.89*a)
8 m.set_this_cycle(b, 0.88*d)
```

Assignments

The two basic types of assignments in **msdsl** are `set_this_cycle` and `set_next_cycle`. `set_this_cycle` acts like a Verilog `assign` statement, while `set_next_cycle` acts like a synchronous assignment in a Verilog `always` block. Examples of both are shown in Listing 5.8.

A same-cycle assignment can either create a new signal with a given name, as on line 7, or be made to an already-declared signal, as on line 8. In the former case, the new signal created is automatically given a numeric format that can hold the range of values produced by the expression being assigned to it. However, next-cycle assignments can only be made to existing signals.

The reason is that next-cycle assignments can contain feedback loops, which means that it is not in general possible to automatically determine the range of the signal being assigned. As an example, consider the integrator $y[n+1] = y[n] + x[n]$: for any non-zero range of x, there is no self-consistent range of values for y. This limitation is not often a problem, because state variables in AMS models often correspond to physical signals, such as voltage and current, which have ranges that are well-defined by the safe operating area (SOA) of devices in the circuit. That said, **msdsl** does have a feature for estimating the range of state variables, which will be discussed later, in the context of spline-based modeling.

Operators

Many Python operators for arithmetic, comparison, and bitwise operations are overloaded in **msdsl**, allowing users to write down expressions conveniently. Currently supported operators include `+`, `-`,

Listing 5.9: Type conversions in **msdsl**.

```
1  from msdsl import *
2  m = MixedSignalModel('model')
3  vref = 1.2
4  # DAC
5  d_in = m.add_digital_input('d_in', width=8)
6  a_out = m.add_analog_output('a_out')
7  m.set_this_cycle(a_out, vref*(d_in/256))
8  # ADC
9  a_in = m.add_analog_input('a_in')
10 d_out = m.add_digital_output('d_out', width=9, signed=True)
11 m.set_this_cycle(d_out, to_sint((a_in/vref)*256, width=9))
12 m.compile_and_print(VerilogGenerator())
```

`*`, `~`, `&`, `|`, `^`, `<<`, `>>`, `<`, `>`, `<=`, `>=`, `==`, and `!=`. The true division operator, `/`, is only partially supported: dividing by constants works, but dividing by variables does not.

msdsl also provides some operators that cannot be directly overloaded from Python, such as `min_op` and `max_op` for finding the minimum or maximum value among several signals. In both cases, the computation is automatically arranged into a tree fashion to minimize the critical path length in the generated logic. The same strategy is automatically applied for basic arithmetic operations such as addition and multiplication.

Another special operator provided by **msdsl** is the ternary operator, `if_(condition, then, else)`, which returns `then` if the `condition` (1-bit signal) holds, and otherwise returns `else`. The `if_` operator is in fact a special case of the more generic operator `array(elements, index)`, which returns a value from a list of signals (`elements`) that is indicated by the signal `index`.

Type Conversion

Although we have mainly looked at the representation of analog signals, there are three basic numeric types in **msdsl**: unsigned integers, signed integers, and real numbers (represented using **svreal**). All three types can be mixed freely in arithmetic expressions; **msdsl** will automatically promote unsigned integers to signed integers, and signed integers to real numbers. Demotion, on the other hand, must be invoked as an explicit cast, because it can result in the loss of information. **msdsl** provides the functions `to_uint` and `to_sint` for that purpose.

Listing 5.9 shows how type conversions can be used to build simple models of a DAC and an ADC. In general, modeling DACs is simpler because it involves a type promotion (integer to real), which is done automatically, as opposed to an ADC, which requires an explicit type demotion.

The Verilog generated by this code example is shown in Listing 5.10, revealing that **msdsl** promotes the unsigned integer `d_in` to signed, prior to converting it to a real number. The reason

Listing 5.10: HDL generated by **msdsl** for the DAC and ADC example.

```
1  module model #(
2      `DECL_REAL(a_out),
3      `DECL_REAL(a_in)
4  ) (
5      input wire logic [7:0] d_in,
6      `OUTPUT_REAL(a_out),
7      `INPUT_REAL(a_in),
8      output wire logic signed [8:0] d_out
9  );
10     // Assign signal: a_out
11     logic signed [8:0] tmp1;
12     assign tmp1 = {1'b0, d_in}; // UInt -> SInt
13     `INT_TO_REAL(tmp1, 9, tmp0);
14     `MUL_CONST_REAL(0.0046875, tmp0, tmp2);
15     `ASSIGN_REAL(tmp2, a_out);
16     // Assign signal: d_out
17     `MUL_CONST_REAL(213.33333333333334, a_in, tmp4);
18     `REAL_TO_INT(tmp4, 9, tmp3);
19     assign d_out = tmp3;
20 endmodule
```

is that the **svreal** macros `INT_TO_REAL` and `REAL_TO_INT` operate only on signed integers, but this implementation detail is abstracted from **msdsl** users.

msdsl also automatically performs compile-time arithmetic simplifications, so even though the expressions for `a_out` and `d_out` each involve a division, followed by a multiplication, the generated HDL is only a single multiplication by a constant (Listing 5.10, lines 14 and 17). **msdsl** provides this optimization so that users are free to write arithmetic expressions in Python as they see fit, without having to worry about how term grouping will impact the hardware implementation.

Arbitrary Functions

msdsl provides a utility for transforming arbitrary Python functions into synthesizable implementations for FPGA emulation. This feature can be used to model static nonlinearities and discretize analog dynamics subject to variable timesteps, as described in Chapter 4.

As an example, consider modeling an RC filter with variable timesteps. Since the previously-derived solution for the RC filter dynamics (Equation 5.18) is valid for any timestep over which the input is constant, the variable-timestep update equation for the RC filter is:

$$y[k+1] = \alpha(\Delta t_k) \cdot y[k] + (1 - \alpha(\Delta t_k)) \cdot x[k] \tag{5.19}$$

where $\Delta t_k = t_{k+1} - t_k$ and $\alpha(\Delta t) = e^{-\Delta t/(RC)}$. Note the implicit assumption that $x(t)$ holds the

Listing 5.11: Variable-timestep modeling in **msdsl**.

```
1  import numpy as np
2  from msdsl import *
3  r, c = 1e3, 1e-9
4  m = MixedSignalModel('rc')
5  x = m.add_analog_input('x')
6  dt = m.add_analog_input('dt')
7  y = m.add_analog_output('y')
8  func = lambda dt: np.exp(-dt/(r*c))
9  f = m.make_function(func, domain=[0, 10*r*c], numel=512, order=1)
10 a = m.set_from_sync_func('a', f, dt)
11 x_prev = m.cycle_delay(x, 1)
12 y_prev = m.cycle_delay(y, 1)
13 m.set_this_cycle(y, a*y_prev + (1-a)*x_prev)
14 m.compile_and_print(VerilogGenerator())
```

value $x[k]$ from t_k to t_{k+1}.

The big problem with this update equation is that it involves a nonlinear function of the time, $\alpha(\Delta t)$, which cannot be mapped to the basic arithmetic operations described so far. We use **msdsl**'s method for converting ordinary Python functions into synthesizable approximations to address that problem.

Building functions in **msdsl** is a two-step process: a user first constructs a piecewise-polynomial approximation of a function using `make_function`, and then applies the function one or more times using `set_from_sync_func` or `set_from_async_func`.

Listing 5.11 shows an example of how that process works in the context of the variable-timestep RC filter model. In that case, the function that needs to be implemented is $\alpha(\Delta t)$, which is defined as an ordinary Python function on line 8. Then, on the next line, `make_function` creates a piecewise-polynomial approximation of that function over ten RC time constants (via the `domain` argument), which captures almost all of the settling behavior of the step response. The `order` of the approximation is set to `1`, meaning "piecewise-linear" (PWL), and the number of PWL segments used, `numel`, is 512.

Finally, on line 10, `set_from_sync_func` generates an instance of the hardware shown in Fig. 5.2, which converts an input value to a real-number address between `0` and `numel-1`. The integer part of that address is used to read out the coefficients of a piecewise-polynomial segment, which are multiplied by the fractional part of the address raised to the zeroth, first, second power, etc.; the resulting products are summed to produce the output value.

ROM implementation Although not shown in Fig. 5.2, the ROMs that store the coefficients of the piecewise-polynomial approximation can be either synchronous (i.e., one clock-cycle delay

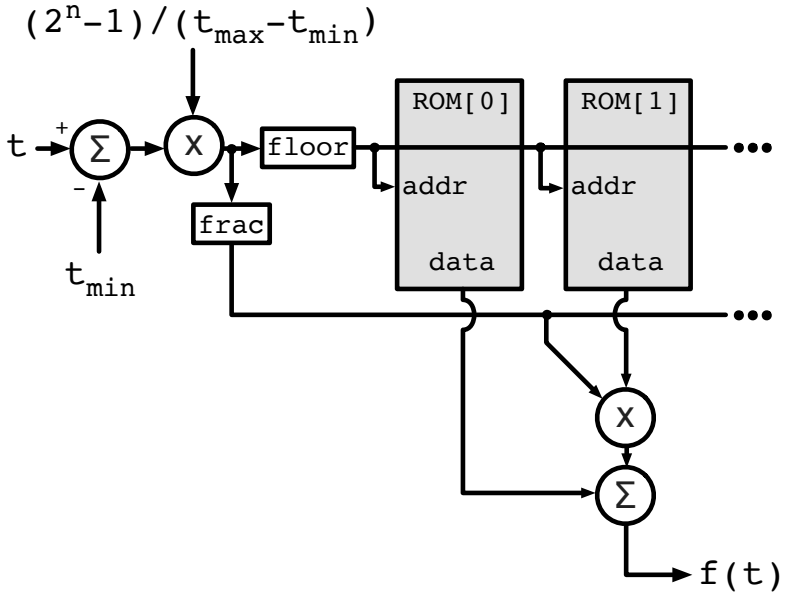


Figure 5.2: **msdsl** approximates a function $f(t)$ with a piecewise-polynomial representation defined over the domain $[t_{min}, t_{max}]$, using 2^n-element lookup tables for the coefficients.

from input to output) or asynchronous (i.e., computation completes in the same clock cycle); `set_from_sync_func` specifies the synchronous implementation while `set_from_async_func` specifies the asynchronous implementation.

In general, it easier to work with the asynchronous version, because one does not have to take into account the clock cycle of delay in implementing the surrounding logic. However, synchronous ROMs are more resource-efficient on an FPGA, because they are mapped to BRAM.

As an example of that difference in efficiency, consider the ROMs that will be instantiated by `set_from_sync_func` on line 10 of Listing 5.11. Since the function being constructed is piecewise-linear, there will be two lookup tables: one for the offset of each segment, and one for the slopes. By default, each coefficient is an auto-formatted 18-bit fixed-point number, and there are 512 piecewise-linear segments, so the total ROM size needed is 512x36. That is exactly the size of a half-BRAM slice in the Zynq-7000 series, which is the minimum amount of BRAM that can be allocated. Taking the Z-7045 FPGA as an example, 0.5 BRAM represents 0.0917% of the total BRAM on-chip.

Now let's consider the resource utilization for `set_from_async_func`. Since `BRAM` is synchronous, it cannot be used to implement an asynchronous ROM, so LUTs have to be used instead. In the Zynq-7000 series, LUTs are 6-input, meaning that 8 LUTs are needed to produce each bit of each piecewise-polynomial coefficient. The total number of coefficient bits is $2 \cdot 18 = 36$, so a total of 288 LUTs would be needed to implement the ROM in the asynchronous case. On a Z-7045 FPGA, that represents 0.132% of the available LUTs.

Hence, asynchronous functions can be almost 50% more resource-hungry than synchronous functions. But perhaps more importantly, asynchronous functions consume LUTs, while synchronous functions consume BRAMs. Since emulating digital parts of a chip design is often more reliant on LUT resources than BRAM, synchronous functions may be able to better "stay out of the way" of the resources needed for digital emulation.

Runtime-updatable functions One of the other advantages of using BRAM to implement functions is that its contents can be updated at runtime. This is very useful when the function being implemented represents an analog behavior, such as a step response, that users might want to vary. Since asynchronous functions are LUT-based, they can only be changed by rebuilding the bitstream, which is time-consuming and inconvenient. For synchronous functions, on the other hand, **msdsl** allows users to alter functions at runtime via additional signals passed to `set_from_sync_func` (such as write address, write data, etc.).

That said, it is not possible to completely redefine functions at runtime, at least when **msdsl** is generating fixed-point models, because the ranges of coefficients in the ROMs need to be known at compile-time in order for **svreal**'s fixed-point auto-formatting to work. This means that functions can only be altered to the extent that their coefficients stay within ranges that are declared at compile-time.

To make it easier to construct runtime-updatable functions with sufficient flexibility, **msdsl** provides a special type of function declaration called a `PlaceholderFunction`, which takes the place of the `make_function` invocation on line 9 of Listing 5.11. When instantiating a `PlaceholderFunction`, a user need only specify the *ranges* of coefficients at compile-time, not their values.

It might seem difficult to estimate those ranges ahead of time, but since all polynomial terms are computed from the fractional part of an address (Fig. 5.2), which runs from 0 to 1, the range of a coefficient corresponds directly to the largest contribution it can make to the function evaluation.[2] Hence the range of the zeroth coefficient should be the range of function values that can be produced, and the range of other coefficients should be set to the biggest jump in function value that can occur from the start of one function value to the next. (To be safe, those ranges should be padded a bit.)

Of course, if more is known about the family of functions to be represented, the coefficient ranges can be constrained more precisely, which in turn increases their resolution, due to the direct connection between range and resolution in **svreal**. In practice, however, I have not found it necessary to use such advanced knowledge when constructing runtime-updatable functions.

Address calculation At this point, one might wonder why **msdsl** builds functions using a fixed spacing between polynomial segments, since variable spacing would permit a more efficient representation, in terms of the number of segments required. The answer can be seen by examining

[2] To see why, observe that $|ct^n| \leq |c|$ when $t \in [0, 1]$, so long as $n \geq 0$.

Listing 5.12: Single-input, multiple output functions in **msdsl**.

```
import numpy as np
from msdsl import *
m = MixedSignalModel('model')
x = m.add_analog_input('x')
y1 = m.add_analog_output('y1')
y2 = m.add_analog_output('y2')
func1 = lambda t: np.sin(t)
func2 = lambda t: np.cos(t)
f = m.make_function(
    [func1, func2], domain=[-np.pi, np.pi], numel=512, order=1)
m.set_from_sync_func([y1, y2], f, x)
m.compile_and_print(VerilogGenerator())
```

the function implementation illustrated in Fig. 5.2: fixed-spacing lends itself to efficient address calculation hardware.

One can imagine a strategy that strikes a balance between truly arbitrary spacing and constant spacing, by breaking apart the function domain into several sections, each with its own fixed spacing. A function such as a step response would benefit from such an approach, since it varies rapidly only over a small part of the function domain. However, since BRAM is relatively abundant on modern FPGAs, I found that using a fixed spacing, even for such functions, did not require an unreasonable resource utilization to achieve good accuracy.

One area that did benefit from optimization, however, was the sharing of address calculation hardware for single-input, multiple-output functions. I discovered this issue when implementing variable-timestep models, which often need to evaluate several functions that depend on the timestep (described in more detail in Chapter 4). The problem is that a naive implementation, built from single-input, single-output functions, will re-implement the calculation of the ROM address from the function input many times. That, in turn, causes unnecessary DSP slice utilization, since a multiplier is used in the address calculation.

To solve that problem, **msdsl** allows users to declare and apply lists of functions, in addition to single functions. Listing 5.12 shows an example of how that works, for a simple model that simultaneously computes $\sin(x)$ and $\cos(x)$. Using that feature enabled me to cut the DSP utilization of the high-speed link example described in Chapter 4 by about 35%.

Pseudorandom Noise

msdsl includes a facility for generating pseudorandom noise. This is important not only for modeling voltage-domain noise, but also time-domain noise such as random jitter. **msdsl** generates uniform noise by scaling the output of a pseudorandom number generator (PRNG), but can also

Listing 5.13: Uniform noise generation in **msdsl**.

```
1 from msdsl import *
2 m = MixedSignalModel('model')
3 y = m.add_analog_output('y')
4 m.set_this_cycle(y, m.uniform_signal())
5 m.compile_and_print(VerilogGenerator())
```

produce arbitrarily-distributed noise, and includes a special mode for Gaussian noise generation that provides high-accuracy modeling of the tails of that distribution. This section describes the usage and implementation of those three modes, concluding with an example showing how the Gaussian noise feature can be used to model Johnson-Nyquist noise in an RC filter.

Uniform Noise To generate uniform noise distributed between 0 and 1, a user can simply invoke the `uniform_signal()` method of a `MixedSignalModel`. This returns a `Signal` object that can be assigned to a model output or an internal signal, as illustrated in Listing 5.13. The range of the pseudorandom variable can easily be adjusted to something other than $[0, 1]$ by providing optional arguments `min_val` and `max_val` to `uniform_signal`.

Under the hood, **msdsl** generates uniform noise by scaling the integer output of a PRNG. More precisely, it scales the n-bit output of the generator, x, to produce a real number, y, according to:

$$y := \texttt{min\_val} + \frac{\texttt{max\_val} - \texttt{min\_val}}{2^n - 1} \cdot x \tag{5.20}$$

Three different PRNG implementations are provided: a linear feedback shift register (LFSR), a linear congruential generator (LCG), and a Mersenne Twister implementation (MT19937). LFSR is the default, since it is simple and resource-efficient, although it does not produce very "random" results.

The output of an LFSR is periodic; its period can be up to $2^n - 1$, where n is the number of registers in the LFSR. In general, the LFSR period should be made as long as possible to flatten the power spectral density (PSD) of the generated noise. Achieving the maximum LFSR period requires careful selection of the bits used in the LFSR feedback equation, and the optimum equation depends on the length of the LFSR. That said, **msdsl** users don't have to worry about this, because they can simply specify the LFSR length, and **msdsl** will look up the optimum equation from a reference document published by Xilinx [4]. LFSR lengths from 3 to 168 are supported.

Although the output of an LFSR is periodic, it can start at any point, and hence approximately independent pseudorandom variables can be constructed by initializing n-bit LFSRs with different values. Since that is likely what users want, the default behavior of **msdsl** is to generate a random initialization for each LFSR at compile-time. However, users can manually specify LFSR initial

values via an optional argument, `lfsr_init`, if needed. In fact, the same argument controls the initialization of LCG and MT19937 PRNGs.

On that note, the PRNG can be changed to an LCG implementation by setting an optional argument, `gen_type` to `'lcg'`. The LCG PRNG is primarily included to make it easier to match emulation results to Verilog simulators, which tend to generate random numbers according to the following 32-bit LCG in the Verilog-2001 specification [1]:

$$x\left[k+1\right] := \left(1 + 69069 \cdot x\left[k\right]\right) \mod 2^{32} \tag{5.21}$$

Even though that explicit LCG definition is not present in more recent specifications of the Verilog language, the output of `$urandom` on several simulators suggests that the original Verilog-2001 LCG is often still used today. Hence, **msdsl** provides the option of using that LCG PRNG.

The LCG and LFSR generators have the advantage of having a low resource footprint, but they are not very "random," in the sense that the sequence of integers they produce can have noticeable patterns, called "runs." This is most significant for an LFSR, which when initialized to "1," will see its magnitude approximately double on each update cycle, until that "1" reaches the end of the shift register. To make matters worse, that behavior is guaranteed to occur at some point during the operation of an LFSR, since changing the initialization only shifts the output sequence in time.

To produce a better approximation of randomness, **msdsl** lets users switch to MT19937 for generating pseudorandom integers, using a synthesizable implementation from Alex Forencich [18]. MT19937 has an extremely long period, $2^{19937} - 1$, and performs much better than LCG and LFSR in tests of randomness, such as Diehard [44]. Despite its improved performance, MT19937 is more resource-intensive, consuming hundreds of LUTs and FFs, as well as a few BRAM slices. In addition, it takes tens of thousands of cycles to start up, which can significantly increase the time required to simulate an emulator design, a common activity during the development process.

At this point, one might wonder if high-fidelity random number generation is really necessary for mixed-signal emulation, which tends to be geared towards functional verification and FW/SW development. The answer ultimately depends on the type of system being emulated: consider a high-speed link design, for example. In such systems, analog circuits, digital circuits, firmware, and software all interact to achieve a very low bit error rate (BER), such as 1e-12 for PCIe [55]. At that level, if we hope to estimate the BER even to within an order of magnitude, we must have accurate pseudorandom variable modeling far into the tails of probability distributions (e.g., 7-8 standard deviations for a Gaussian distribution), and avoid unrealistic "runs," even if they only occur rarely. Hence, it is useful to provide users with the option for high-fidelity random number generation, even though it is more resource-intensive, because it is needed for at least some types of mixed-signal designs.

Listing 5.14: Arbitrary noise generation in **msdsl**.

```
1 from msdsl import *
2 from scipy.stats import truncnorm
3 m = MixedSignalModel('model')
4 y = m.add_analog_output('y')
5 inv_cdf = lambda x: truncnorm.ppf(x, -8, +8)
6 inv_cdf_func = m.make_function(inv_cdf, domain=[0.0, 1.0])
7 m.set_this_cycle(y, m.arbitrary_noise(inv_cdf_func))
8 m.compile_and_print(VerilogGenerator())
```

Arbitrary Noise Up until this point, we have only considered the generation of uniform noise, but noise in real systems is often distributed in a more complex fashion. It is well-known that any noise distribution can be modeled by applying its inverse CDF to uniform noise distributed between 0 and 1.

msdsl makes this process straightforward, as illustrated in Listing 5.14: the inverse CDF is defined as a regular Python function (line 5), which is approximated as piecewise-polynomial (`make_function`, line 6), and then passed to a function called `arbitrary_noise` (line 7), which generates a uniformly-distributed random variable and applies the inverse CDF to it.

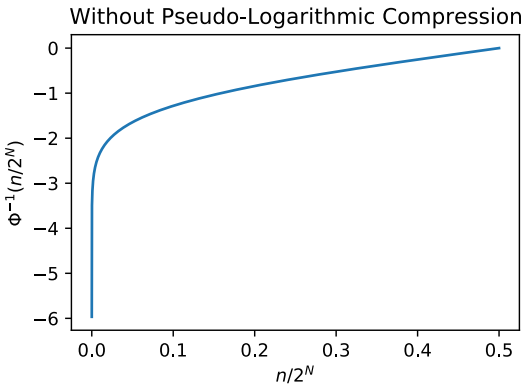
The arbitrary noise modeling feature is useful for working with lab measurements, because a CDF can easily be constructed from a sequence of observations.[3] Since CDFs are always monotonic [8], they can be inverted simply by swapping their x- and y-axes. For distributions that are described analytically, the inverse CDF is often directly available through software packages. SciPy [70], as an example, provides it as a method called `ppf` for dozens of distributions.

Gaussian Noise In theory, we could generate Gaussian noise using the `arbitrary_noise` feature just described, using the inverse CDF of the Gaussian distribution, Φ^{-1}. However, that would not result in a high-fidelity model. The problem has to do with approximating the inverse CDF with equal-length polynomial segments: as others have pointed out, using equally-spaced lookup tables for the inverse Gaussian CDF can severely distort its tails [42, 11].

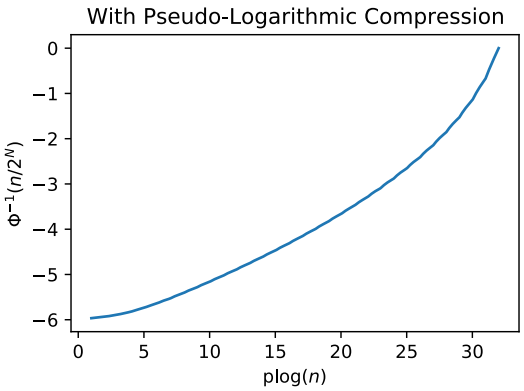
To see why this happens, suppose that we are trying to model the inverse Gaussian CDF as PWL with 512 segments. We will immediately run into a problem, because Φ^{-1} is infinite at the edges of its domain. This problem, at least, can be solved by instead using the inverse CDF of a truncated normal distribution [36], whose values are distributed over a finite range.

However, that will not fix the fundamental issue that the edges of the inverse Gaussian CDF are *extremely* sharp, as illustrated in Fig. 5.3a. Suppose that we were to approximate the Gaussian distribution as truncated between -6 and +6 (i.e. six standard deviations), and let the inverse CDF of that distribution be $\tilde{\Phi}^{-1}$. Since we're using 512 PWL segments from 0 to 1, the spacing between

[3] Given n observations: sort them, place them on the x-axis, then plot with y-values $1/n$, $2/n$, ..., 1.



(a) The inverse CDF of the Gaussian, $\Phi^{-1}(x)$, is extremely sharp near the edges of its domain, $x \approx 0$ and $x \approx 1$. (Only the left side of the domain is illustrated here.)



(b) By applying a pseudo-logarithmic compression to the input of the inverse CDF, the function becomes much smoother, and therefore easier to represent with polynomial segments.

Figure 5.3: Computing the inverse CDF of the Gaussian distribution (truncated to $\pm 6\sigma$)

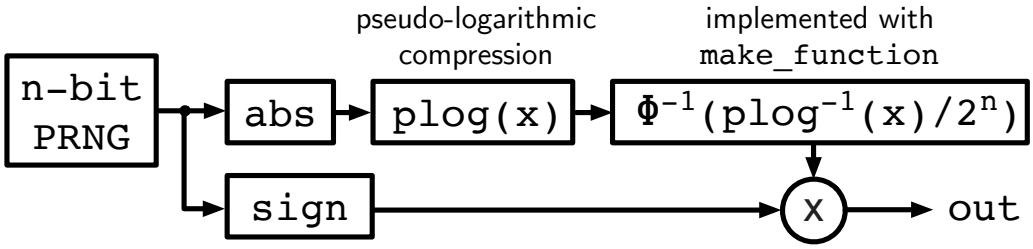


Figure 5.4: Gaussian noise generation in **msdsl**. A PRNG produces an n-bit signed integer, the absolute value of which is run through a pseudo-logarithmic compression, followed by an appropriately distorted version of the inverse CDF. The sign of the original integer sets the sign of the output.

sampling points of $\tilde{\Phi}^{-1}$ is 1/512. Hence, if we evaluate $\tilde{\Phi}^{-1}(1/1024)$ on the emulator, we will get the average of $\tilde{\Phi}^{-1}(0)$ and $\tilde{\Phi}^{-1}(1/512)$: namely, -4.4. However, the true value is only -3.1, which is a huge error, considering that the maximum error of the very next segment is only 0.017, almost two orders of magnitude smaller.

The impact of this error is to make unlikely values appear more frequently. In this case, a value of -4.4 or smaller will appear with probability 1/1024, whereas in reality such a value should only appear with probability 5.4e-6, or 180 times less frequently.

To solve this problem, **msdsl** implements Gaussian noise by distorting the output of a PRNG with **svreal**'s pseudo-logarithmic function, plog, prior to applying an appropriately distorted version of the inverse Gaussian CDF, as illustrated in Fig. 5.4. As a result, the function that needs to be implemented by **msdsl** is $\Phi^{-1}\left(\text{plog}^{-1}(x)/2^n\right)$, a function that is much flatter than Φ^{-1} (Fig. 5.3b).

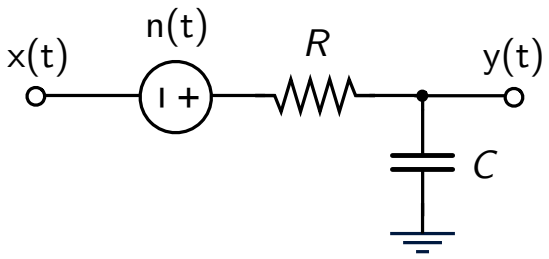


Figure 5.5: RC filter model including Johnson-Nyquist noise.

One of the advantages of this technique, as opposed to a Gaussian-specific approach such as Box-Muller [12], is that it could be applied without modification to any symmetric unimodal distribution. Asymmetric unimodal distributions (e.g., skew normal) could be handled with a slight modification to the architecture in Fig. 5.4, by having the sign of the input select the (distorted) inverse CDF to be applied. It might even be possible to extend the technique to a multimodal distribution by partitioning its domain into single-tail regimes, each with its own inverse CDF.

Example: Noisy RC filter I will close out this section by showing how **msdsl**'s noise modeling features can be used to implement Johnson-Nyquist noise in an RC filter. Such noise is effectively a voltage source, $n(t)$, that appears in series with the input voltage, as illustrated in Fig. 5.5. The power spectral density (PSD) of the noise is flat over frequency and equal to $4k_BTR$, and its amplitude distribution is approximately Gaussian [39]. The question is: how much noise should be added in a discrete-time model to approximate this continuous-time noise source?

We can answer that question by matching the autocorrelation of the continuous-time noise (i.e., the inverse Fourier transform of its PSD), to the autocorrelation of the discretized noise. As shown in an application note from Cadence [66], that approach ends up calling for the standard deviation of discretized noise to be set to $\sigma_n = \sqrt{2k_BTR/\Delta t}$.

This result is used in the full implementation of a noisy RC filter shown in Listing 5.15. The main action happens on line 9, which creates a zero-mean Gaussian-distributed signal `n` with a standard deviation σ_n. The rest of the model is adapted from the noiseless fixed-timestep RC filter model (Listing 5.5), with noise added to the input on Line 11.

5.2.3 Input Formats

So far, we have considered **msdsl**'s low-level building blocks: operator overloading, synthesizable functions, and pseudo-random noise. These features are useful for experimenting with modeling techniques, but are not necessarily the most convenient way to implement a model. For example, it is possible to model an RC filter by hand-discretizing a differential equation, using **msdsl**'s operator

Listing 5.15: Modeling a noisy RC filter in **msdsl**.

```
1  from msdsl import *
2  from math import exp, sqrt
3  r, c, dt = 1e3, 1e-9, 0.1e-6
4  k, T = 1.38e-23, 300
5  m = MixedSignalModel('rc')
6  x = m.add_analog_input('x')
7  y = m.add_analog_output('y')
8  rms = sqrt(2*k*T*r/dt)
9  n = m.set_gaussian_noise('n', std=rms)
10 a = exp(-dt/(r*c))
11 m.set_next_cycle(y, a*y + (1-a)*(x+n))
12 m.compile_and_print(VerilogGenerator())
```

Listing 5.16: Differential equation modeling in **msdsl**.

```
1  from msdsl import *
2  r, c = 1e3, 1e-9
3  m = MixedSignalModel('rc', dt=0.1e-6)
4  x = m.add_analog_input('x')
5  y = m.add_analog_output('y')
6  m.add_eqn_sys([c*Deriv(y) == (x-y)/r])
7  m.compile_and_print(VerilogGenerator())
```

overloading, but not necessarily convenient.

To solve that problem, **msdsl** provides a number of higher-level "input formats" constructed from the low-level building blocks. These include: symbolic systems of linear differential equations, including systems with conditional expressions (switched systems), a SPICE-like netlist format, rational transfer functions, and tools for implementing the spline-based modeling technique introduced in Chapter 4.

Symbolic Differential Equations

Let's start by looking at **msdsl**'s support of symbolic linear differential equations. Listing 5.16 shows how an RC filter can be modeled using this feature: the model and I/O declaration are similar to the hand-discretized RC filter (Listing 5.5), but the original differential equation, rather than its solution, is provided to the model generator (Line 6). Operator overloading, along with a special `Deriv` operator, allows users to write down such equations succinctly.

Although the RC filter's dynamics are quite simple, **msdsl** can handle systems of differential equations with any number of inputs, outputs, and state variables. Internally, **msdsl** rearranges all

user-provided equations into the standard form of a linear dynamical system (LDS):

$$\dot{x} = Ax + Bu \tag{5.22}$$
$$y = Cx + Du \tag{5.23}$$

where u is a vector of inputs, x is a vector of state variables, and y is a vector of outputs. **msdsl** then solves the state update equation over an interval $[t, t + \Delta t]$, resulting in the well-known solution [58]:

$$x\left(t + \Delta t\right) = \tilde{A} \cdot x\left(t\right) + \tilde{B} \cdot u\left(t\right) \tag{5.24}$$

where, assuming A is invertible:

$$\tilde{A} = e^{\Delta t \cdot A} \tag{5.25}$$
$$\tilde{B} = A^{-1} \cdot \left(\tilde{A} - I\right) \cdot B \tag{5.26}$$

This illustrates one of the key strategies employed by **msdsl**: running expensive computations, such as matrix exponentiation, at compile-time to produce expressions that can be implemented efficiently on an FPGA. For differential equations, the result of that pre-computation is a summation of multiplications by constants, which can be mapped efficiently to DSP slices and adder logic on an FPGA.

Switched Systems

msdsl supports a generalization of linear dynamics, in which a system's state can evolve according to different linear dynamics at each timestep. This capability is particularly important for modeling a switching power converter, which can be approximated as an LDS whose dynamics are selected by the ON/OFF state of its transistors and diodes.

The strategy employed by **msdsl** to model such systems is to solve each linear operating mode, as described in the previous section, resulting in a set of equations that describes the time evolution of the system in each operating mode k:

$$x\left(t + \Delta t\right) = \tilde{A}_k \cdot x\left(t\right) + \tilde{B}_k \cdot u\left(t\right) \tag{5.27}$$

At each time step, one of the operating modes is selected, and the system evolves over a time interval Δt according to the dynamics of that mode, starting from the state computed in the previous timestep. From an FPGA implementation perspective, that is not much less efficient than the implementation of a single-mode LDS described previously, since constants are replaced by lookup tables, without impacting the utilization of DSP slices or adder logic.

An example of such a switched system is shown in Fig. 5.6, which is an RC filter whose resistance

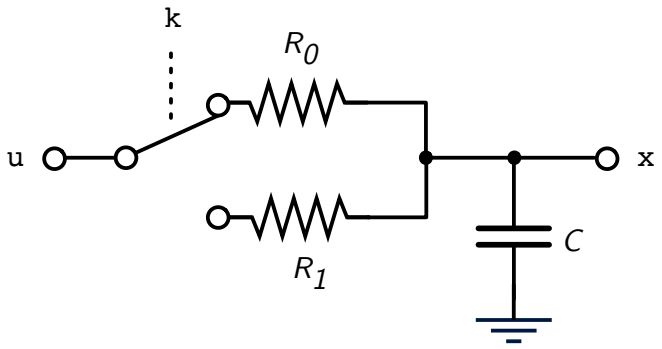


Figure 5.6: Example of a circuit with distinct operating modes, each of which behaves according to linear dynamics. When `k` is asserted, the resistance of the RC filter is R_1; otherwise it is R_0.

Listing 5.17: Switched system modeling in **msdsl**.

```
1 from msdsl import *
2 r0, r1, c = 1234, 2345, 1e-9
3 m = MixedSignalModel('rc', dt=0.1e-6)
4 u = m.add_analog_input('u')
5 k = m.add_digital_input('k')
6 x = m.add_analog_output('x')
7 g = eqn_case([1/r0, 1/r1], [k])
8 m.add_eqn_sys([c*Deriv(x) == (u-x)*g])
9 m.compile_and_print(VerilogGenerator())
```

can be switched between R_0 and R_1, depending on the switch control signal, `k`. Listing 5.17 shows how that system can be modeled in **msdsl** by using `eqn_case`, which indicates a mode-dependent expression in a system of equations.

More precisely, lines 7 and 8 of that code sample specify the differential equation:

$$C \cdot \frac{dx}{dt} = \frac{u-x}{R_1} \quad \text{if} \quad \texttt{k} \quad \text{else} \quad \frac{u-x}{R_0} \tag{5.28}$$

Hence, with reference to Equation 5.27, the coefficients of the switched LDS solution are:

$$\tilde{A}_k = e^{-\Delta t/(R_k C)} \tag{5.29}$$

$$\tilde{B}_k = 1 - e^{-\Delta t/(R_k C)} \tag{5.30}$$

Although in this example, the switch condition is a single bit, in general it can consist of multiple bits. As an example, suppose the switch position is specified by two bits, `k0` and `k1`, capable of selecting one of four resistances for the RC filter. Other than adding those model inputs, the only

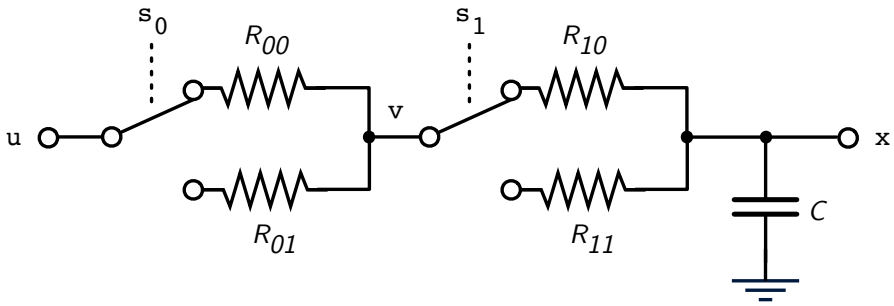


Figure 5.7: Modeling example in which two independent switches can alter the dynamics of the RC filter, resulting in four distinct linear operating modes.

other change required would be to specify more equation cases and more switch condition bits:

```
g = eqn_case([1/r0, 1/r1, 1/r2, 1/r3], [k0, k1])
```

In fact, if there are multiple `eqn_case` statements, **msdsl** will enumerate all unique combinations of switch conditions and solve the system of differential equations for each combination. To demonstrate that capability, consider the circuit illustrated in Figure 5.7, where two independent switches control the resistance of an RC filter, resulting in four distinct linear operating modes.

Listing 5.18 demonstrates how such a circuit can be modeled in **msdsl**. The two switchable resistances are modeled as conditional expressions on lines 8 and 9, using separate switch conditions. Then, on lines 11-14, they are used in a system of differential equations that describes the behavior of the circuit according to KVL and KCL. This is the first example involving more than one equation, and also the first involving an internal variable (v, in this case).

Internal variables are declared as `AnalogSignal` instances, and are particularly convenient when working with more complex systems, since they reduce the amount of hand analysis that is required on the part of the user. For example, one could write down a single differential equation describing the behavior of the two-switch system by solving for the effective resistance of the series combination of the switched resistors. However, it is likely more convenient to simply apply KCL at the intermediate node, v.

From a performance or resource utilization perspective, there is no downside to introducing stateless internal variables, like the one in this example, because **msdsl** performs a compile-time optimization to eliminate internal variables that are not used elsewhere in the model. Since the signal v falls into that category, it will be eliminated. However, if a capacitor were attached to that node, it would have to be calculated and stored for the next emulation cycle.

At this point, one might wonder how many distinct operating modes can be represented on an FPGA. The answer depends on the amount of LUTRAM on the FPGA, as well as the complexity of the system being modeled. For example, suppose we are working with a Xilinx XC7Z045 FPGA,

Listing 5.18: Modeling systems of differential equations with multiple conditional statements.

```
from msdsl import *
r00, r01, r10, r11, c = 123, 234, 345, 456, 1e-9
m = MixedSignalModel('rc', dt=0.1e-6)
u = m.add_analog_input('u')
s0 = m.add_digital_input('s0')
s1 = m.add_digital_input('s1')
x = m.add_analog_output('x')
g0 = eqn_case([1/r00, 1/r01], [s0])
g1 = eqn_case([1/r10, 1/r11], [s1])
v = AnalogSignal('v')
m.add_eqn_sys([
    (u - v) * g0 == (v - x) * g1,
    (v - x) * g1 == c * Deriv(x)
])
m.compile_and_print(VerilogGenerator())
```

which is a higher-end FPGA in the Zynq-7000 series, containing about 14 Mb of LUTRAM ($2^6 \cdot 218,600$ bits). We could store about 780,000 18-bit coefficients in that space, although to avoid hogging all of the LUT resources, 100,000 coefficients is a more realistic number. Assuming that the system being modeled has a small number of I/Os and state variables, requiring a half-dozen distinct coefficients, we could handle around 14 independent switch conditions (i.e., $2^{14} = 16,384$ coefficients in each lookup table).

The takeaway is that only switched systems with a small number of switch conditions can be modeled efficiently. As a rule of thumb, resource utilization will start to grow quickly once the number of switch conditions exceeds the number of address bits for an individual LUT (e.g., 6 bits for the Zynq-7000 series).

Handling systems with a larger number of switch conditions requires the system to be partitioned into subcircuits, each with a small number of switch conditions. An example is shown in Fig. 5.8, where two independently-switched RC filters are connected through a buffer. Due to the buffer, there is no loading interaction between the two subcircuits, which means that as long as the intermediate signal v is accurately represented over a single timestep (by making the timestep small enough or using spline points), partitioning will not significantly degrade the model's accuracy. Of course, in real circuits, there is always some loading interaction between subcircuits; accurate partitioning requires dependence of that interaction on the switch condition to be negligible.

As a side note, we could have used block RAM (BRAM) tiles instead of LUTRAM. However, the one-cycle latency of BRAM would have effectively cut the emulation throughput in half, since we would have to determine switch conditions in one emulation cycle before evaluating coefficients in affected switched systems. To make matters worse, the output of a switched system could itself

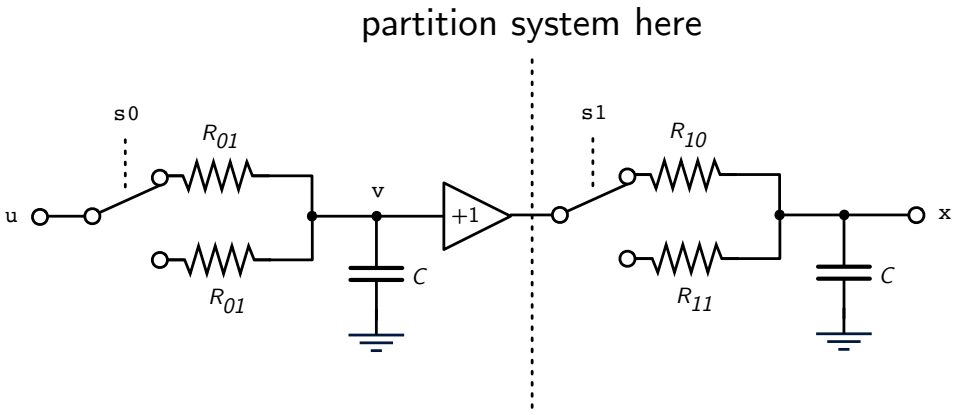


Figure 5.8: Example of a switched system that can be partitioned into subcircuits, each with a smaller number of switch conditions.

generate a switch condition signal that would require an additional emulation cycle to process, resulting in an even greater slowdown.

Netlist Interface

It may be convenient to describe simple circuits using a manually-created system of differential equations, but that approach quickly becomes cumbersome as circuit complexity grows. As a result, **msdsl** provides a netlist interface for building models, in which users can instantiate basic electrical components. **msdsl** compiles the netlist into a system of differential equations, with transistors and diodes introducing multiple operating modes as described in the previous section. The system of equations is solved at compile-time; unlike a CPU-based SPICE simulator, the FPGA emulator does not attempt to invert matrices at runtime.

Linear circuits Listing 5.19 shows how the netlist interface can be used to model a basic RC filter. Line 6 creates a `Circuit` object to contain the netlist, and lines 8-10 instantiate the capacitor, resistor, and voltage source representing the filter input in that netlist. **msdsl** then builds a system of differential equations representing the system using KVL and KCL.

Since the voltage across the capacitor is a state variable, its range must be specified by the user, since state variable ranges cannot be automatically determined by **svreal**, as discussed earlier. This can be done in a flexible way, using the `RangeOf` operator to extract the ranges of existing signals. In this case, we know that the output range cannot exceed the input range, so the capacitor voltage range is set to `RangeOf(x)`. However, in cases where the capacitor voltage range is not obvious, it can be set to a small multiple of the capacitor breakdown voltage, since the capacitor's voltage

Listing 5.19: Netlist-level modeling of an RC filter in **msdsl**.

```
1  from msdsl import *
2  r, c = 1e3, 1e-9
3  m = MixedSignalModel('rc', dt=0.1e-6)
4  x = m.add_analog_input('x')
5  y = m.add_analog_output('y')
6  circ = m.make_circuit()
7  gnd = circ.make_ground()
8  circ.capacitor('net_y', gnd, c, voltage_range=RangeOf(x))
9  circ.resistor('net_x', 'net_y', r)
10 circ.voltage('net_x', gnd, x)
11 circ.add_eqns(AnalogSignal('net_y') == y)
12 m.compile_and_print(VerilogGenerator())
```

should not exceed that level in normal operation (and if it does, the capacitor will not be well-modeled as a linear element). Similarly, when specifying the range of an inductor's current (i.e., its state variable), the saturation current can be used as a guideline.

Since **msdsl**'s netlist entry format is just a frontend for generating a system of symbolic differential equations, additional symbolic equations can be added to that system. This is illustrated on line 11, where the model output `y` is set equal to the voltage of `net_y`. At the moment, this is the only way to wire voltages and currents from within a `Circuit` to internal signals and I/O in a `MixedSignalModel`. Since that is a very common operation, in the future it would be nice to add a netlist primitive for probing voltage and current signals.

Transistors and diodes The netlist input format supports not only linear components, but also transistors and diodes, as long they can be approximated as on/off switches. For circuits with these nonlinear elements, **msdsl** automatically generates a system of differential equations with conditional statements, leveraging the **msdsl**'s support of switched systems. In other words, it "solves" the circuit at compile time for all on/off combinations of transistors and diodes.

At runtime, it is straightforward to determine which transistors are on or off, since they are directly controlled by gate voltage signals. Diodes, however, pose a bit of a challenge, because they are not controlled by an explicit signal. As a result, **msdsl** must generate the control logic to determine the on/off status of each diode.

It is tempting to use a simple voltage threshold to determine whether a diode should turn on or off. However, that does not work well for a switch-modeled diode, because once the diode is on, its voltage is essentially clamped. As an extreme example, suppose that the diode were modeled as an ideal switch, with zero current through it when off and zero voltage across it when on. Now suppose that the diode switch condition is $v \geq 0$ and the diode is initially off. Once the voltage across the

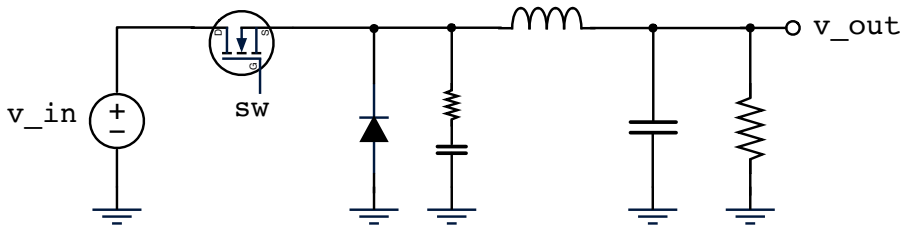


Figure 5.9: Buck converter circuit used as an example of modeling transistors and diodes with **msdsl**'s netlist interface.

diode goes slightly positive, its voltage will be clamped to zero, latching the diode on forever. The problem cannot be solved by simply changing the switch condition to $v > 0$, since that will cause the diode to shut itself off as soon as it turns on.

As I mentioned, this is an extreme example, since a more realistic switch model for a diode would have finite on and off resistances. However, a simple voltage threshold is still a poor choice in such cases, often causing delay and chatter in diode transitions. To solve that problem, **msdsl** uses a voltage threshold to determine when a diode should turn on, and a current threshold to determine when it should turn off, which is a common approach in computer simulation of power circuits (e.g., as described by Verghese [69]).

As a practical use case of switch-level modeling of transistors and diodes, consider the buck converter topology in Fig. 5.9, whose **msdsl** implementation is shown in Listing 5.20. As shown on lines 13 and 14, transistors and diodes can be instantiated with a single line of code, just like any other device in **msdsl**'s netlist mode. Since both are ultimately modeled as switches, they are given on and off resistances.

One might wonder why this simple example includes an RC snubber, since snubbers, while common in switched power converters to reduce EMI, are not fundamental to the buck topology. The reason is fairly interesting, because it has to do with the intersection of auto-formatted fixed-point numbers and analog circuit behavior. If the snubber were not there, during the short period that both the transistor and diode are off, the inductor current would flow through the parallel combination of their off resistances, which is likely at least 10 kΩ, if not much greater. Since the inductor current may well be on the order of 10 A, the range calculated by **svreal** for the switch node could be greater than 100 kV. Since the real range of switch node voltage is more likely on the order of 10 V, this would result in a resolution reduction of 10,000x or more for the fixed-point number used to represent the switch node.

Of course, the switch node would never reach such voltages in real life, not only because the diode would turn on long before the switch node reached that point, but also because there is always some parasitic capacitance on the switch node that limits how fast it can rise. As a result, this problem can be solved by increasing the level of realism of the buck converter with some capacitance on the

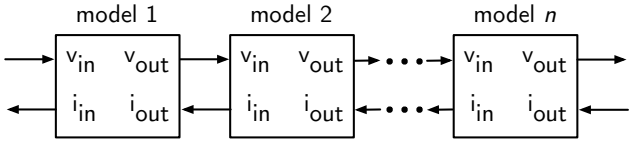
Listing 5.20: Modeling a buck converter using the netlist interface

```
from msdsl import *
# declare I/O
m = MixedSignalModel('buck', dt=0.1e-6)
sw = m.add_digital_input('sw')
v_in = m.add_analog_input('v_in')
v_out = m.add_analog_output('v_out')
# create circuit
c = m.make_circuit()
gnd = c.make_ground()
# input
c.voltage('net_v_in', gnd, v_in)
# transistor + diode
c.switch('net_v_in', 'net_v_sw', sw, r_on=1.0, r_off=10e3)
c.diode(gnd, 'net_v_sw', r_on=1.0, r_off=10e3)
# snubber
c.capacitor('net_v_sw', 'net_v_x', 100e-12,
    voltage_range=100.0)
c.resistor('net_v_x', gnd, 300)
# inductor + capacitor
c.inductor('net_v_sw', 'net_v_out', 2.2e-6,
    current_range=20.0)
c.capacitor('net_v_out', gnd, 10e-6, voltage_range=10.0)
# load
c.resistor('net_v_out', gnd, 5.5)
# assign outputs
c.add_eqns(v_out == AnalogSignal('net_v_out'))
# print output
m.compile_and_print(VerilogGenerator())
```

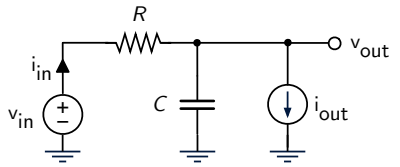
switch node.

With the RC snubber in place, the switch node voltage range, when both the diode and transistor are off, is reduced to $(R_{snub} + \Delta t/C_{snub}) \cdot I_L$. The dependence on the timestep essentially means that the amount of "mandatory snubbing" is reduced with shorter timesteps.

To close out the discussion on modeling transistors and diodes, it's worth noting that the on/off switch-level modeling concept could be extended to a more general form, where the I-V relationship of a nonlinear device is represented with several piecewise-linear segments. In that case, each segment would represent a distinct linear mode of the device. While this could potentially yield better accuracy, the resource utilization of such an approach would grow quickly, as the total number of switched modes for a given system would be the product of the number of PWL segments for each device.



(a) In some cases, `Circuit` models can be connected together without losing loading effects, by including the current drawn by input as an *output* and similarly accepting an *input* representing current drawn from the output.



(b) Example of a circuit-level model that includes input and output currents for modeling loading.

Figure 5.10: Modeling loading in **msdsl**.

Modeling loading In certain specific cases, it is possible to connect multiple `Circuit` models while preserving loading effects. As illustrated in Fig. 5.10a, this can be done by including an *output* for the current drawn by the circuit's input, and an *input* for the current drawn from the circuit's output.

In general, tearing netlists this way is a bad idea; it is much better to break circuits at high-impedance inputs such as MOS gates. This is because loading forms a feedback loop between subcircuits, which must be interrupted by a one-cycle delay to avoid a combinational logic loop. Such time-delayed loading will fail unless the timesteps taken by the emulator are small compared to circuit time constants.

That said, when modeling switching power converters, it is sometimes necessary to break the converter into multiple pieces (e.g., power factor corrector, flyback, etc.) so that there aren't too many transistors and diodes in any one subcircuit. That would be problematic, because it could cause the number of switched modes to become very large. Fortunately, in a digitally-controlled switching power converter, the operating frequency of the digital controller is often orders of magnitude faster than analog time constants. Hence, for that specific case, it is possible to capture loading effects between **msdsl** `Circuits`.

As an example of how loading effects can be added to a `Circuit`, consider the RC filter model in Fig. 5.10b.[4] As shown in Listing 5.21, only a few changes need to be made to the original RC `Circuit`. First, the current drawn from v_{in} is measured (line 12) and assigned to a model *output* (line 15). In a complementary fashion, a current source is connected to v_{out}, whose value is set by a model *input*.

Focusing on the current measurement (line 12), observe that creating a voltage source returns a symbolic variable corresponding to the current through the voltage source. That return value wasn't used in the previous RC model (and as a result was not explicitly computed in the generated model), but here it is assigned to a model output. Since current measurements are defined as positive going

[4] In practice, one should never tear apart a cascaded RC circuit like this to model loading. This is simply a small example showing how to measure current and apply current.

Listing 5.21: Modeling an RC filter with "hooks" to capture the effect of loading.

```
1  from msdsl import *
2  r, c = 1e3, 1e-9
3  m = MixedSignalModel('rc', dt=0.1e-6)
4  vin = m.add_analog_input('vin')
5  iin = m.add_analog_output('iin')  # note: output!
6  vout = m.add_analog_output('vout')
7  iout = m.add_analog_input('iout')  # note: input!
8  circ = m.make_circuit()
9  gnd = circ.make_ground()
10 circ.capacitor('net_vout', gnd, c, voltage_range=RangeOf(vin))
11 circ.resistor('net_vin', 'net_vout', r)
12 c_iin = circ.voltage('net_vin', gnd, vin)
13 circ.current('net_vout', gnd, iout)
14 circ.add_eqns(
15   iin == -c_iin,
16       vout == AnalogSignal('net_vout')
17 )
18 m.compile_and_print(VerilogGenerator())
```

into the positive terminal of a voltage source, a negative sign is needed on line 15.

Transfer Functions

Although we have looked at various ways to model schematic-level designs over the past few sections, it is sometimes preferable to describe analog behavior using a transfer function, rather than a model derived from a circuit implementation. To support that modeling style, **msdsl** provides an operator called `set_tf`, which defines a transfer function relationship between two signals. Transfer functions are specified by lists of coefficients for numerator and denominator polynomials, which means that any rational transfer function can be represented.

Listing 5.22 illustrates the use of that operator, defining the RC filter transfer function:

$$Y(s)/X(s) = 1/(sRC+1) \tag{5.31}$$

The transfer function polynomials are specified in descending order from the highest-order term, so in this case the numerator is written as `[1]`, while the denominator is written as `[RC, 1]`.

As with the previous features described, the underlying theme is that **msdsl** performs computationally expensive operations at compile-time to generate hardware-efficient representations for FPGA implementation. In this case, **msdsl** uses the `cont2discrete` function from SciPy to perform the transfer function discretization.

Listing 5.22: Transfer function modeling in **msdsl**.

```
1 from msdsl import *
2 r, c = 1e3, 1e-9
3 m = MixedSignalModel('rc', dt=0.1e-6)
4 x = m.add_analog_input('x')
5 y = m.add_analog_output('y')
6 m.set_tf(x, y, [[1], [r*c, 1]])
7 m.compile_and_print(VerilogGenerator())
```

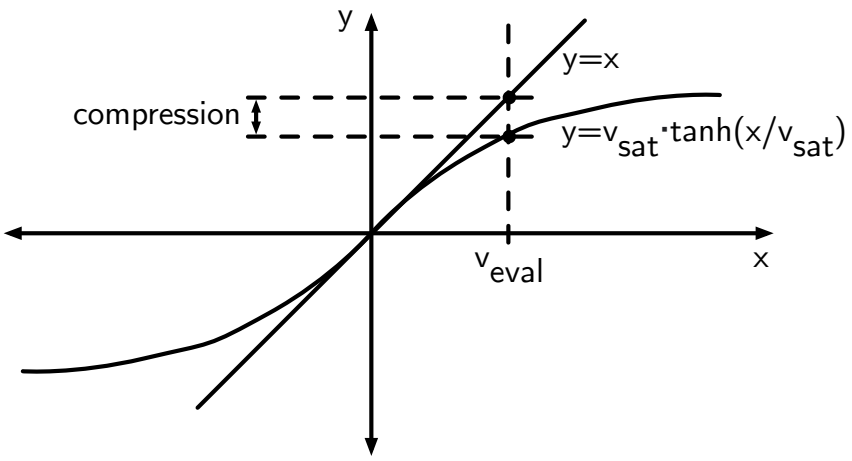


Figure 5.11: `SaturationModel` models a unity-gain buffer with a hyperbolic tangent-shaped compression. The amount of compression is controlled by the parameter v_{sat}.

Spline Modeling

At the highest level of abstraction, **msdsl** provides spline-based implementations of the three types of AMS blocks discussed in Chapter 4: a saturation nonlinearity, a SISO state-space system, and a step response system. Each one is a subclass of the generic AMS model type, `MixedSignalModel`.

Saturation Nonlinearity The simplest of these models is `SaturationModel`, which implements a unity-gain buffer with hyperbolic tangent-shaped compression. As shown in Fig. 5.11, the amount of compression is quantified by evaluating the voltage transfer curve at an input v_{eval} and comparing it to the ideal output of a linear buffer. For example, if the model's output is 0.9 V for a 1.0 V input, we would say that its compression is $20 \cdot \log_{10} 0.9/1.0 \approx -0.915\,\text{dB}$.

Users specify the amount of compression they want when instantiating `SaturationModel`. For example, if the model should exhibit -1.0 dB compression for a 1.0 V input, they would type:

```
m = SaturationModel(-1, 'dB', veval=1.0)
```

Behind the scenes, **msdsl** solves for the value of v_{sat} (from Fig. 5.11) that results in the specified compression.

SISO state-space system **msdsl** also provides a spline implementation for a generic SISO state-space system, `LDSModel`:

```
m = LDSModel(A, B, C, D, num_spline, spline_order)
```

The first four arguments are the matrices of the state-space equations, `num_spline` is the number of spline points to use, and `spline_order` is the order of the implicit interpolation method.

Since it can be cumbersome to write down state-space equations directly, **msdsl** provides a subclass of `LDSModel`, `TFModel`, that accepts polynomial coefficients for the numerator and denominator of a rational transfer function, as with `set_tf`:

```
m = TFModel(num, den, dtmax, num_spline, spline_order)
```

From a usability perspective, a challenge in supporting state-space systems is determining state variable ranges, which are needed for **svreal**'s fixed-point auto-formatting system. This is particularly problematic for `TFModel`, because users do not directly control the state-space representation, and therefore cannot provide range information.

To solve that problem, **msdsl** uses a common approach described by Oppenheim and Schafer [52]. Discrete-time impulse responses, $h_j[n]$, are calculated for each state variable; the state variable ranges are then calculated as the $\ell 1$ norms of those impulse responses times the input range, $\pm R$. In other words, the range of the j-th state variable is $\pm R \cdot \sum |h_j[n]|$.

Since the system being modeled is continuous-time, the discrete-time impulse responses for state variables are computed by oversampling the system's state update equation with a timestep Δt_{ov}:

$$h[n] = e^{\Delta t_{ov} A} \cdot h[n-1] + \int_0^{\Delta t_{ov}} e^{(\Delta t_{ov} - \tau)A} \cdot b \cdot \delta[n] \, d\tau \tag{5.32}$$

$$= e^{\Delta t_{ov} A} \cdot h[n-1] + A^{-1} \cdot \left(e^{\Delta t_{ov} A} - I\right) \cdot b \cdot \delta[n] \tag{5.33}$$

assuming that A is invertible. For notational convenience, h is defined as a vector containing the individual impulse responses for each state variable.

Further assuming that the impulse response is causal, $h[-1] = 0$, and therefore $h[0] = A^{-1} \cdot \left(e^{\Delta t_{ov} A} - I\right) \cdot b$. Therefore, by induction, the full impulse response is given by:

$$h[n] = \left(e^{(k+1)\Delta t_{ov} A} - e^{k \Delta t_{ov} A}\right) \cdot A^{-1} \cdot b \tag{5.34}$$

In general, the impulse response is infinitely long, so its $\ell 1$ norm can only be approximated by a finite sum. This results in an underestimate of the norm, and hence the worst-case state variable ranges, because the terms in the $\ell 1$ summation are nonnegative. To account for that,

msdsl multiplies range estimates by a factor called `state_range_safety`, which defaults to "10," but can be overridden by the user. The default is overkill, but only ends up throwing away a few bits of resolution to essentially guarantee that overflow will be avoided. Since the default fixed-point width of state variables is large (`LONG_WIDTH_REAL`), this conservative approach does not significantly degrade accuracy.

Step response system The final type of spline model supported by **msdsl** is for a system characterized by a step response:

```
m = ChannelModel(t_step, v_step, dtmax, num_spline, module_name)
```

where `v_step` is a list of values sampled from the step response and `t_step` is a list of times at which those values were sampled. As with `TFModel`, the arguments `dtmax` and `num_spline` indicate the maximum timestep and number of spline points.

Being able to provide a step response as time-value pairs is powerful, because it allows the step response to be calculated analytically, computed from S-parameters, drawn by hand, etc. However, since channels are so often described by S-parameter measurements, **msdsl** provides a subclass of `ChannelModel`, `S4PModel`, that accepts a Touchstone file [19] as input.

```
m = S4PModel(s4p_file, dtmax, num_spline, module_name)
```

Specifically, it works with S4P Touchstone files, which contain differential S-parameter measurements (i.e., 4x4 complex matrices) over a range of frequencies. Given a source and load impedance, **msdsl** converts those S-parameters measurements to a step response using the process described in Appendix B.

5.3 anasymod

Once **msdsl**-generated models are swapped into the DUT's RTL, the result is a synthesizable model of the entire chip. **anasymod** picks up the emulation flow from there by wrapping control infrastructure around the DUT and automating the FPGA build process. The development of **anasymod** is led by Gabriel Rutsch of Infineon, which generously agreed to release the work as open source under a BSD license at the start of the project. Since I wasn't the primary developer for **anasymod**, I cover it in a more abbreviated fashion than the previous two tools, with a focus on features that I helped to develop to support mixed-signal modeling.

5.3.1 Basic Usage

anasymod operates on a folder containing design sources (Verilog, VHDL, etc.) and YAML files that configure its behavior, as illustrated in Fig. 5.12. Its basic usage requires only two files, `tb.sv`,

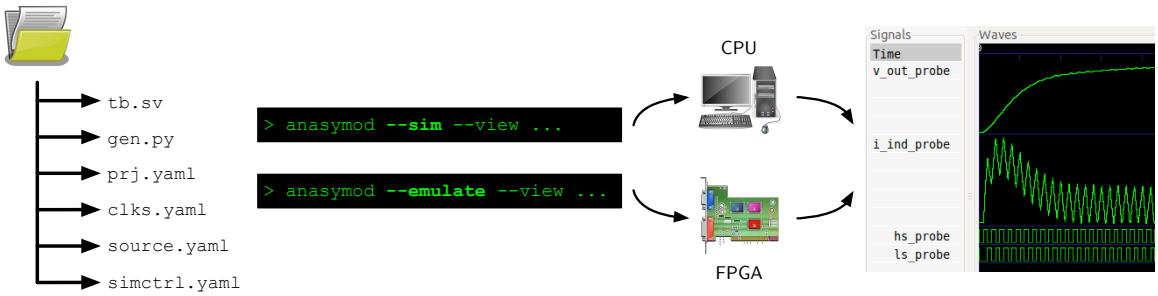


Figure 5.12: **anasymod** operates on a folder containing HDL sources, configured by various YAML files (all optional except `prj.yaml`). It can run a computer-based simulation of the design or an FPGA-based emulation.

representing the top level of the design or synthesizable testbench, and `prj.yaml`, which contains information such as the name of the FPGA board being used and the target emulation frequency.

anasymod can be invoked either as a command-line tool or as a Python library. The command-line mode is useful for simple projects because it minimizes the overhead to get started, whereas the library mode is more suitable for larger projects and those involving complex test stimuli. In either case, **anasymod** can launch a computer simulation or run an emulation on an FPGA board. Although **anasymod** is geared towards emulation, computer simulation is useful for spot-checking the construction of an emulator prior to launching the time-consuming bitstream build process.

The simulators currently supported are Xilinx Vivado, Cadence Xcelium, and Icarus Verilog (open-source). Roughly speaking, Vivado simulations are useful to check that Vivado properly elaborates the design, but are slower than Xcelium simulations. Icarus Verilog is great for small designs, because it has a shorter startup time as compared to the other two, but it is slower for large designs, and doesn't support as many SystemVerilog features. Once the simulation looks OK, the user can switch over to the emulation mode, which automates FPGA tools to build a bitstream and program it to one of the supported FPGA boards.

On that note, **anasymod** currently supports seven Xilinx FPGA boards, spanning a range of capabilities: Arty A7, PYNQ Z1, ZC702, ZC706, ZCU102, ZCU106, and VC707. Changing the target board for emulation often requires only a one-line change in `prj.yaml`. (Occasionally, the emulator frequency needs to be changed as well, which is set in the same YAML file.) Adding support for new boards is usually a quick process, as well, because only a few details are required, such as the FPGA clock pin location(s), whether the clock is single-ended or differential, etc.

Since it's usually not convenient to cram a whole design into the `tb.sv` file, users can list additional source files in a file called `source.yaml`; about a dozen different file types are supported (Verilog, VHDL, EDIF, etc.). This YAML file can also specify certain files as only pertaining to simulation or emulation, which is helpful for cases where FPGA equivalents need to be substituted for IP blocks and standard cells. To make it easier to deal with generated HDL, such as models

produced by **msdsl**, **anasymod** recognizes a file called `gen.py` as containing code that should be run to produce sources necessary for simulation and emulation. Files produced by `gen.py` do not have to be specified in `source.yaml`.

Test stimulus I/O and waveform probing are specified in a file called `simctrl.yaml`. For computer simulation, **anasymod** calls Verilog system tasks as necessary to save the specified waveforms to a VCD file, and routes stimulus I/O to a user-provided stimulus block described in HDL. The emulation mode provides the same functions, but its implementation is different: waveforms are captured by an Integrated Logical Analyzer on the FPGA, which **anasymod** downloads and converts to a VCD file, while test stimulus I/Os are routed to a Virtual I/O (VIO) or Processing System (PS) block and made available to the user through a Python interface.

The final configuration file supported by **anasymod** is called `clks.yaml`, which supports variable timestep management and clock generation, two key features for mixed-signal modeling. I'll describe these features in the next two subsections.

5.3.2 Variable timestep management

anasymod associates a timestep with every emulation cycle; the timestep can either be a static value, determined at compile time (i.e., a fixed timestep), or a dynamic value, determined at run time (i.e., a variable timestep). To support variable timestep operation, **anasymod** lets any AMS model make a request for the size of the next timestep; the timestep actually taken is the minimum of all requests.[5] To make it easier to implement this scheme, **anasymod** generates hardware that gathers all requests, computes the timestep, and communicates that information back to AMS models (Fig. 5.13).

The `clks.yaml` configuration file lets users specify which models need to participate in the timestep management scheme. Some models, such as a variable-timestep filter, need only participate in a read-only fashion: they must know what the emulation timestep is, but do not need to make timestep requests. However, a model such as an oscillator needs read/write access, because it needs to request a timestep corresponding to its next clock edge, and also needs to keep track of emulator timesteps to know how far in the future that edge will occur.

As an example, consider the simple oscillator model in Fig. 5.14. If the oscillator output is currently low, then it makes a timestep request that corresponds to the exact time at which its output should go high, and if the request is granted, then the oscillator output goes high. If the request is not granted, that means the timestep chosen by the timestep manager was smaller than the timestep requested by the oscillator. Hence, the oscillator decrements its timestep request by the difference between the requested and granted timestep. The same procedure is followed for a

[5] A fixed-timestep model can interoperate with variable-timestep models by having it request timesteps that correspond to its sampling frequency. If the fixed-timestep model cannot be modified, a wrapper module can request a timestep for the fixed-timestep model and connect a "timestep granted" signal to the fixed-timestep model's clock enable port.

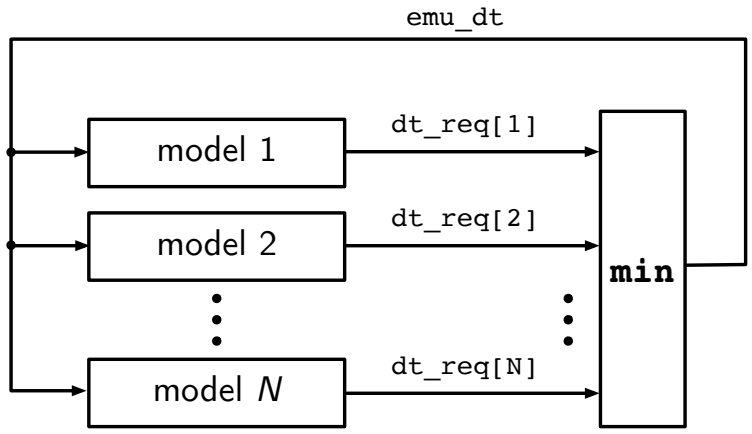


Figure 5.13: **anasymod** automatically generates infrastructure for timestep management, which gathers timestep requests, takes the minimum, and passes that information back to AMS models.

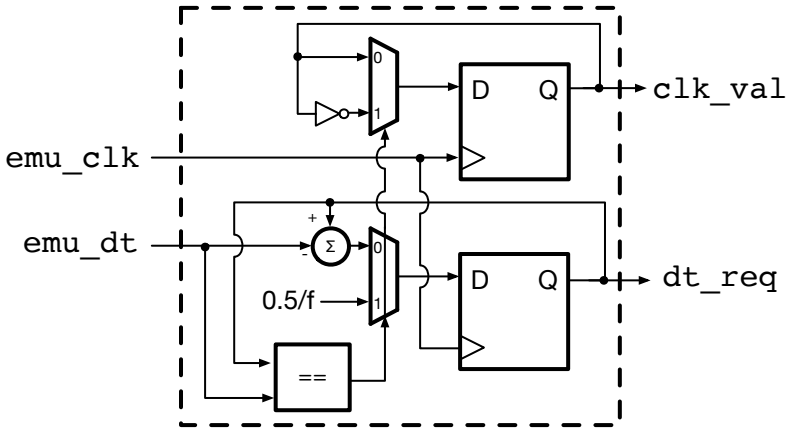


Figure 5.14: Simple oscillator model, operating at a frequency f, that illustrates the timestep request interface.

falling edge transition.

5.3.3 Emulator clock

In the simple oscillator model, there is an input for a signal called the "emulator clock" (`emu_clk`). This is not a real clock signal that would be present in a chip design, but instead marks the transition from one emulator timestep to the next. The need for an emulator clock is not unique to oscillator models; in fact, it is generally required for emulating continuous-time analog dynamics. Considering the RC filter as another example, its physical I/O consists of only an analog input and an analog output. However, in a discrete-time approximation, a clock signal is needed to update its state.

As illustrated in Fig. 5.15, **anasymod** takes care of generating the emulator clock and routing

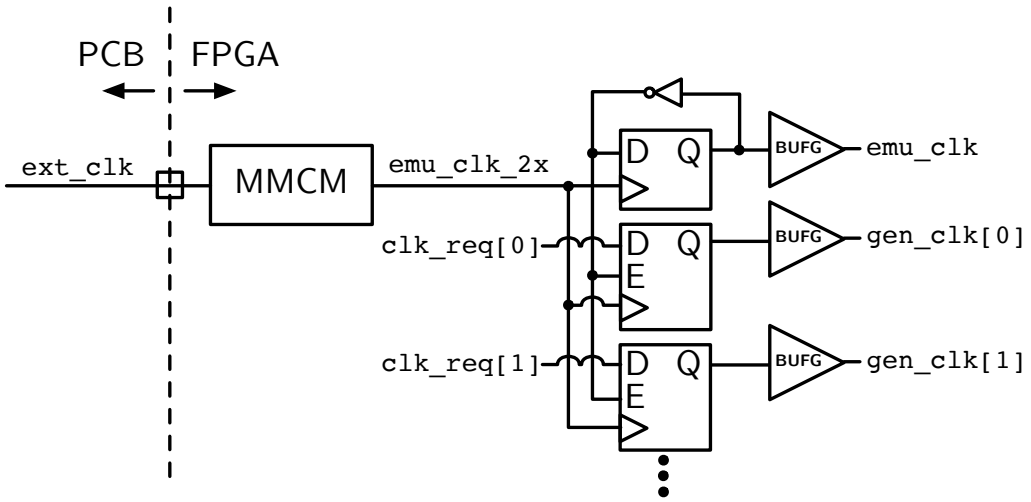


Figure 5.15: **anasymod** generates clock infrastructure that produces an emulator clock signal, along with "true" clock signals that appear in the DUT. The input clock, `emu_clk_2x`, runs at twice the frequency of the emulator clock.

it to AMS models that need it. As with timestep management, users may specify modules that need the emulator clock in the `clks.yaml` file. However, it is not always necessary to do that, as **anasymod** defines the `CLK_MSDSL` macro to point to the emulator clock, which is the default clock used in generated **msdsl** models.

The frequency of the emulator clock is the rate at which timesteps are processed, and it is completely independent of the size of emulation timesteps. Even as emulator timesteps vary over a wide range, the emulator clock continues to run at a fixed frequency, the upper bound of which is determined by the capabilities of the FPGA. In the special case that all emulator timesteps are equal to the period of the emulator clock, the emulator runs in real-time.

5.3.4 Generated clocks

In addition to the emulator clock, AMS designs generally have one or more "true" clocks that are present in the real chip design. In Fig. 5.15, for example, both oscillator outputs drive unmodified digital RTL, as opposed to AMS models. In an emulation context, these "true" clocks must meet four requirements: (1) they must be glitch-free, (2) they should utilize FPGA clock routing resources, (3) their duty cycle should be accurate with respect to "emulator time," and (4) the relationship between the various clocks should not introduce unrealistic timing problems.

The last issue requires a bit of explanation. Suppose, as shown in Fig. 5.16, that an RC filter provides the input to an ADC. The RC filter's dynamics are updated on the emulator clock, `emu_clk`, while the ADC sample is taken on the rising edge of a clock signal from an oscillator model, `clk_adc`. Since `clk_adc` is generated from the emulator clock, it is delayed with respect to `emu_clk`. If that

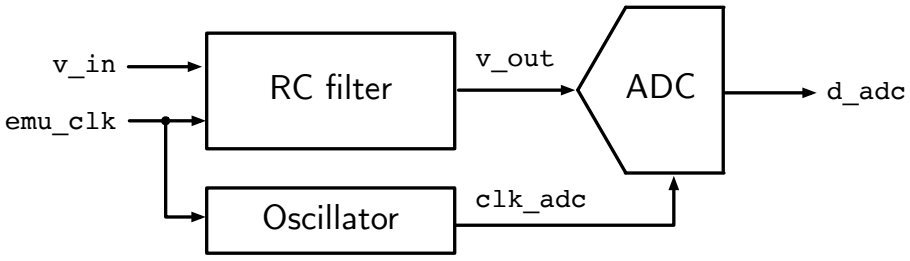


Figure 5.16: Illustration of the hold-time hazard in an AMS emulator with generated clocks. The RC filter output changes on `emu_clk`, but the clock that samples the result, `clk_adc`, is delayed with respect to `clk_adc` and therefore may arrive too late.

delay is large enough, there will be a hold time issue at the ADC input, because the RC filter output could change before the `clk_adc` edge occurs. To make matters worse, since the RC output is represented as a multi-bit digital signal, a hold time violation could cause an arbitrary voltage to be sampled by the ADC; it is not simply a matter of an extra cycle of delay.

I resolved all four clocking issues by having **anasymod** generate the clocking infrastructure illustrated in Fig. 5.15. An AMS model may emit a "clock request," which is the value that the clock signal will have during the next emulator clock cycle. The emulator then uses that signal to generate a glitch-free clock signal whose edges (both rising and falling) are aligned to rising edges of the emulator clock. Finally, **anasymod** wires the generated clock signal, which is driven using an FPGA clock buffer, back into the model that sent the clock value request.

Our scheme is a bit different than how generated clocks are typically handled on an FPGA; clock gating is a more commonly used approach. However, if we had generated the "true" clocks by gating the emulator clock, we could not have given them arbitrary duty cycles with respect to emulator time. This would have been problematic for AMS emulation because the behaviors of some analog blocks vary with the duty cycle of clock inputs. In fact, even in digital emulation, if there are negative edge-triggered circuits in a multi-clock system, clock duty cycle can affect the sequence of operations.

5.3.5 Interactive tests

When running an emulation, there is generally some degree of interaction required between the host computer and the FPGA board to control the test stimulus. While one could implement stimuli entirely as synthesizable RTL, I found that such an approach is not flexible enough for realistic use cases, owing to the long amount of time that it takes to rebuild an FPGA bitstream to change the stimulus. As a result, **anasymod** provides a Python interface for reading and writing signals within a running emulator. Two different implementations of that interface are provided: a VIO-based approach, and a PS-based approach called "firmware-in-the-middle" (FiM).

Listing 5.23: Using **anasymod** to run a VIO-based interactive test on an emulator.

```
1  from anasymod.analysis import Analysis
2  ana = Analysis('path/to/rc')
3  ana.set_target('fpga')
4  ctrl = ana.launch()  # program FPGA
5  ctrl.stall_emu()
6  ctrl.set_param(name='v_in', value=1.0)
7  ctrl.set_reset(1)
8  ctrl.set_reset(0)
9  for _ in range(25):
10     ctrl.refresh_param('vio_0_i')
11     v_out = ctrl.get_param('v_out')
12     t = ctrl.get_emu_time()
13     print(f't: {t}, v_out: {v_out}')
14     ctrl.sleep_emu(0.1e-6)
```

VIO approach

Listing 5.23 shows a simple example of VIO-based interaction with an emulated RC filter, drawn from a unit test in the **anasymod** repository. In this case, the host-emulator interaction consists of letting the filter run for a short period of (emulated) time, pausing the emulator to read the filter output, and repeating the process a number of times.

The FPGA board is programmed via the `launch` command (line 4), which returns a `CtrlApi` object. That object can be used to write signals via `set_param` (line 6) and read signals via `get_param` (line 11). Special methods for controlling the flow of emulated time are also provided, such as `stall_emu` (line 5), `get_emu_time` (line 12), and `sleep_emu` (line 14).

Under the hood, these Python methods are implemented by sending commands to a Vivado TCL interpreter running in the background (which is created by the `launch` command). That background process, in turn, interacts with a VIO instance on the FPGA through a USB-JTAG interface. As one might suspect, all of this overhead means that VIO-based interaction with an emulator is quite slow. Having observed that problem, we started exploring alternatives, which lead to the development of the FiM approach.

FiM approach

With FiM, a Python program running on the host computer sends commands over USB-UART to a processor on the FPGA, whose firmware translates those commands into low-level interactions with the DUT. The processor firmware can be either generated by **anasymod** or customized by the user for maximum performance.

When **anasymod** is used to generate PS firmware, the user receives a PS-based implementation

Listing 5.24: Example of custom firmware running on an FPGA PS.

```
#include "gpio_funcs.h"
// ...
set_tms(1);
cycle();  // Move to Select-DR-Scan state
set_tms(0);
cycle();  // Move to Capture-DR state
set_tms(0);
cycle();  // Move to Shift-DR state
u32 retval = 0;
for (u32 i=0; i<(length-1); i++){
    set_tdi((data_in >> i) & 1);
    retval |= (get_tdo() << i);
    cycle();
}
// ...
```

of the `CtrlApi` interface that can be used as a drop-in replacement for VIO-based interaction. This can immediately provide a speedup of 1-2 orders of magnitude as compared to the VIO approach, because it cuts out several layers of overhead. (Measured speedups will be discussed more precisely in Chapter 6.)

A further speedup of 1-2 orders of magnitude can be achieved by writing custom PS firmware that performs bit-level actions on-FPGA, such as toggling TCK/TDI/TDO/TMS to read and write JTAG registers. **anasymod** streamlines the process of writing custom firmware by generating `get_*` and `set_*` methods for the signals listed in `simctrl.yaml`. An example of using those methods is shown in Listing 5.24, which is a snippet of custom firmware implementing a JTAG SHIFT-DR command.[6]

It might seem difficult to determine how to partition the emulator between Python code, PS firmware, and PL RTL, but that choice can be informed by thinking about the lab equipment that will interact with the physical chip when it returns from fabrication. For example, users typically control JTAG cables through SHIFT-IR and SHIFT-DR commands, so those commands should be implemented on-FPGA, rather than on the host computer.

Comparison

Since FiM can provide a speedup of 3-4 orders of magnitude as compared to VIO, while preserving a good degree of flexibility, it is generally the preferred approach. However, FiM is currently only

[6]Complete JTAG driver firmware can be found within the open-source DragonPHY project: https://github.com/StanfordVLSI/dragonphy2/blob/e78c943271d29511b3ec316281ecd1ef60b0687e/tests/fpga_system_tests/emu_macro/main.c#L47-L153. That said, in the future, firmware for commonly-used interfaces such as JTAG (e.g., I2C, SPI etc.) should be distributed with **anasymod**, rather than copied in from other projects.

supported on Zynq-7000 and Zynq-Ultrascale+ FPGAs, which have ARM cores. In theory, FiM could be extended to other FPGAs by using MicroBlaze, a soft PS core, but **anasymod** does not yet include this feature. Hence, for FPGA boards not supported by FiM, VIO serves as a fallback for running interactive tests.

Of course, whether FiM or VIO is used, DUT stimuli that have to be very fast, such as a PRBS generator, should be implemented in RTL. However, it is still a good idea to make such stimulus blocks configurable by the host computer to reduce the likelihood of having to rebuild the emulator bitstream. For example, one would likely want to be able to change the polynomial and initial state of a PRBS generator from Python running on the host computer, even when the PRBS generator is implemented as RTL.

Chapter 6

The Paradigm Cases

The open-source framework just described has been applied to seven mixed-signal designs: an NFC-powered chip, a firmware-controlled flyback converter, a battery impedance sensor, an automotive magnetic sensor [59], an IGBT driver, a microcontroller-based automotive power management unit (PMU), and the open-source high-speed link DragonPHY [37]. These applications cover a wide range of mixed-signal circuits and emulation use cases, demonstrating the generality of the framework. In this chapter, I will describe experimental results from the flyback, NFC, and DragonPHY applications. The focus will be on DragonPHY, since the design and emulator are both open-source, which means that the results can be shared in detail and are reproducible by others.

6.1 Firmware-Controlled Flyback Converter

The first example we'll consider is a firmware-controlled flyback converter design from Infineon, illustrated in a simplified form in Figure 6.1. The design consists of a conventional flyback topology, along with several auxiliary windings that feed back signals to a processor, which generates the gate drive waveform for a MOSFET on the primary side of the transformer (only one auxiliary winding is shown in the figure).

When I first started working on this project, Infineon already had a mixed-signal simulation of the system running, which used SIMetrix [67] to simulate analog components at SPICE-level, along with firmware, written in C, that ran as a custom SIMetrix plugin. However, this CPU-based simulation was slow, which meant that it could only be used to spot-check a few important use cases, rather than as a platform for firmware development. Infineon asked me to explore whether that problem could be solved by creating an FPGA emulator for the system, with the idea that firmware developers could use the emulator prior to the availability of silicon.

It would not have been trivial to discretize the flyback converter topology by hand, because it contains multiple coupled windings, several diodes, and snubber circuitry. I instead described the

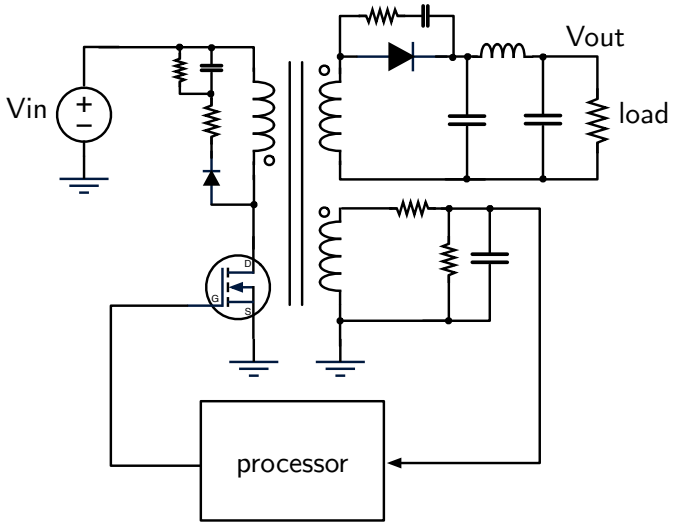


Figure 6.1: Basic firmware-controlled flyback architecture: auxiliary windings (several, but one shown here) feed back signals to a processor, which generates the gate drive waveform.

topology with **msdsl**'s netlist interface, which includes a transformer model, in about 200 lines of Python code. This not only made the code more readable and easier to debug than deriving the discrete-time equations by hand, but also made the model more flexible, since the circuit topology and component values could easily be changed.

I relied on **msdsl**'s handling of switched systems to implement the flyback converter topology, since it contains transistors and diodes that are used in a switch-like fashion. Since the full flyback topology contained five diodes and one transistor, that meant that a total of $2^6 = 64$ different linear operating modes had to be analyzed at compile time (i.e., all combinations of on/off states for all switches). However, this did not take a noticeable amount of time, because the analysis of each mode only requires the computation of a matrix exponential and a few matrix multiplications.

As it turns out, this system could be modeled accurately and efficiently with fixed-timestep oversampling because the processor frequency (10s of MHz) was orders of magnitude faster than analog time constants (10-100 kHz). In other words, the spacing between digital emulator events was so small, that there was no need for "analog-only" timesteps to achieve reasonable accuracy.

I was able to run the completed emulator at a frequency of 35 MHz on a Xilinx ZC706 FPGA board, with each emulator clock cycle corresponding to one cycle of the flyback processor. This represented a speedup of about 2,800x as compared to the existing CPU-based simulation that Infineon was using. In other words, simulations that used to take an hour could run in a few seconds on the emulator.

This application illustrates that the emulation framework lets users generate models that strike

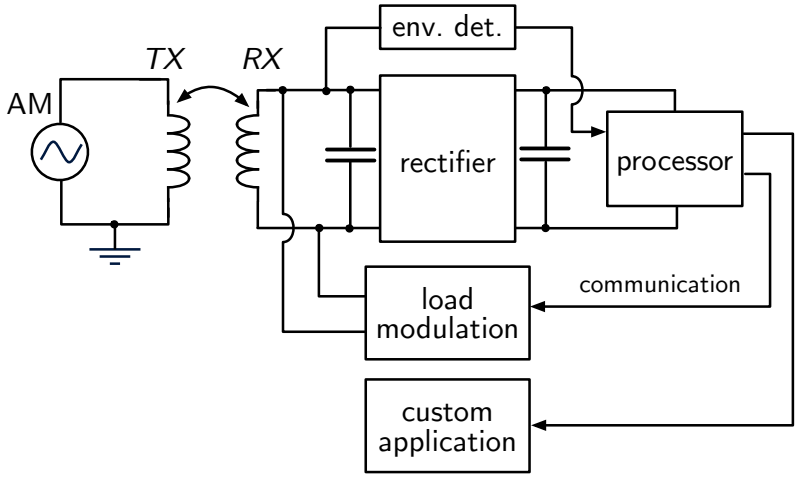


Figure 6.2: Basic NFC architecture; the TX uses amplitude modulation to communicate, while the RX uses load modulation.

a useful balance between accuracy and performance. Given that the flyback emulator was geared toward firmware development, a relatively coarse level of accuracy was sufficient. For example, the flyback model had to exhibit load-dependent ringing on an auxiliary winding, since it was monitored as part of the firmware control algorithm, but the exact shape of the ringing waveform was less important. That meant that we could use the simple on/off models for transistors and diodes discussed earlier, which were fast and resource-efficient.

6.2 NFC-Powered Chip

In another commercial application, we used our framework to build a mixed-signal emulator for the NFC-powered chip sketched in Figure 6.2. In this system, a transmitter (TX) sends a 13.56 MHz carrier to a receiver (RX) coil. The carrier is rectified to supply DC power to a processor and custom application, while amplitude modulation (AM) on the carrier envelope is processed as digital communication from the TX. Finally, the RX communicates by modulating the loading of its coil.

When I first started working on this project, Infineon had already implemented a hardware-in-the-loop (HIL) emulator for the design. The HIL emulator used an FPGA to implement the processor design, while physical analog components on a PCB implemented analog blocks in the chip design. This worked for some types of tests, but had two issues: first, since the analog components used were limited to what was available off-the-shelf, their behavior didn't match that of the on-chip analog blocks (particularly in terms of power consumption). The second issue had to do with varying parameters of the design or use case, since, in some cases, this required modification of the HIL PCB, which made it difficult to conduct parameter sweeps.

As a result, Infineon asked me to explore whether it was possible to make a fully virtual emulator for the NFC-powered chip. In other words, they wanted to emulate the entire design, including both analog and digital blocks, on an FPGA. As a side benefit, this would mean that the emulator could be implemented on an off-the-shelf FPGA board, reducing its cost and making it easier to distribute emulators to developers. (In fact, a fully virtual AMS emulator could even be implemented and distributed using cloud FPGA resources.)

I modeled the NFC receiver using **msdsl**'s differential equation interface in about 70 lines of code. The NFC topology contains various switched elements (envelope detector, rectifier, load modulation), so I made extensive use of the `eqn_case` feature described earlier, which allows for conditional expressions in symbolic differential equations.

Even though the NFC topology is similar to the flyback topology, a key difference between the two is that the analog time constants of the NFC receiver are much faster; they are comparable to the receiver's digital clock period. This is somewhat fundamental, because TX and RX circuits are designed to resonate at the carrier frequency, while the RX processor clock is recovered directly from the carrier waveform. Hence, the spacing between digital events is not small enough that we can use the processor clock directly for fixed-timestep oversampling, as in the previous application.

After making that observation, I realized that we would either need to introduce "analog-only" timesteps to oversample the analog dynamics more finely, or develop a new method that would allow larger timesteps to be taken while still achieving reasonable accuracy. This work preceded the development of the spline-based modeling strategy discussed earlier in this thesis, so I took the former approach (finer fixed-timestep oversampling). That said, I include some thoughts at the end of this discussion on how that might be improved, with the benefit of hindsight.

With that in mind, I introduced 16x oversampling, meaning that there were 16 emulator cycles for every period of the carrier waveform. This directly reduced emulator throughput by the same factor, as pointed out in Chapter 3, but because the oversampling ratio was not particularly high, the emulator still ended up being fast enough for firmware development, and did not warrant further speed optimization at that time.

The fully virtual NFC chip emulator was implemented on a Xilinx VC707 board and achieved a 430x speedup as compared to an existing SystemVerilog simulation that used real number models (RNMs) for AMS blocks. This demonstrates that our framework can virtualize analog blocks in an existing hardware-in-the-loop emulator, while delivering the 2-3 orders-of-magnitude speedup that is expected from emulation.

6.2.1 Taking bigger timesteps

The big challenge in modeling the NFC system is that it contains diodes (in the rectifier and envelope detector), which are not directly turned on or off by digital control signals. Hence, if we only take timesteps corresponding to digital events, the transition of a diode from on to off, or vice versa, has

to be aligned to digital events, even if that transition should occur in between.

When I was working on this application, I addressed this problem by increasing the oversampling ratio until diode transitions were "close enough" that the NFC model exhibited realistic behavior. Since the emulator was fast enough at that point for firmware development, I stopped there.

However, thinking about this system in the context of the spline-based modeling approach I later developed, it would likely be possible to achieve better performance. Suppose that the voltage and current of diodes were described by spline point feature vectors with linear interpolation. It would then possible to determine if a zero crossing (i.e., diode transition condition) occurred between any two successive spline points by checking if they have opposite signs. If so, the emulator could force a new timestep at the estimated time of the earliest zero-crossing.

If the crossing occurs between two spline points u_i and u_{i+1}, separated in time by Δt_h, then the time of the zero-crossing is:

$$t_c = (\Delta t_h u_i) \,/\, (u_i - u_{i+1}) \tag{6.1}$$

This expression does involve a division, which is well-known to be an expensive operation in hardware. However, the division wouldn't have to be very accurate to outperform the coarseness of discretizing diode transitions to a fixed timestep. As a result, one could likely get away with a small lookup table, leveraging the pseudo-logarithmic compression function, plog (x), provided by **svreal**.

6.3 DragonPHY

The last application of the emulation framework that we'll consider is DragonPHY, an academic 16 Gb/s high-speed link design [37]. Both the design and the emulator variants described in this section are available as open-source on GitHub (`https://git.io/dragonphy`).

As shown in Fig. 6.3, DragonPHY uses an ADC-based receiver architecture, consisting of 16 time-interleaved ADCs organized into four banks, with each bank driven by one phase interpolator (PI). The PIs are essentially digitally-controlled delay lines and are adjusted, by changing the PI control codes, so that they produce four equally-spaced clock phases. The ADCs in each bank are activated sequentially, with one sample taken on each rising edge of the PI clock.

The DragonPHY architecture is split between an analog core, which contains the ADCs and PIs, and a digital core, which contains DSP circuits to recover the transmitted data from the ADC samples. Since the digital core is somewhat complex, and needs to be simulated over a long period of time to verify low BER, DragonPHY simulations tend to be slow.

I sought to speed up those simulations by emulating the design on an FPGA. For comparison purposes, I created two distinct implementations: a "low-level" emulator that swaps in synthesizable models for the ADCs, PIs, and channel, and a "high-level" emulator that lumps all of those behaviors together into a single model. The "low-level" architecture uses variable timesteps that are determined at runtime, and its AMS models have a one-to-one mapping to the models that would be used in

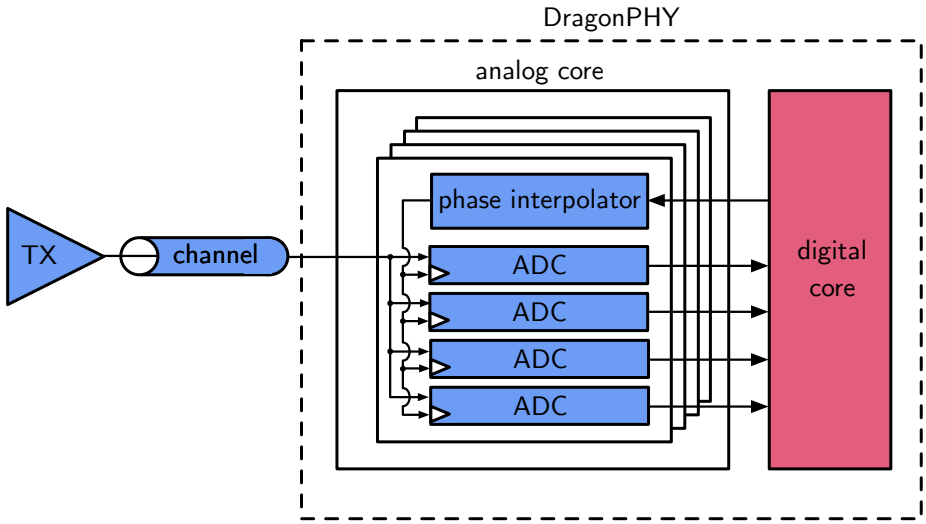


Figure 6.3: The DragonPHY architecture consists of 16 ADCs, organized into four banks, each of which contains a phase interpolator. A digital core processes the ADC samples to recover the transmitted data.

a conventional CPU-based simulation. The "high-level" architecture, on the other hand, has a structure that is optimized for FPGA emulation; it makes better use of parallelism and implements variable timesteps using a computation schedule that is determined at compile time.

We start by discussing variable-timestep modeling of the channel dynamics, which is applicable to both emulator architectures, and then consider each implementation and its relative merits.

6.3.1 Channel modeling

Since the system-level behavior of DragonPHY depends on both clock jitter and sub-picosecond sampling point adjustments, fine time resolution is required in emulating the design. My strategy in achieving this resolution without significantly impacting the emulator is based on ideas from Chapter 4: I have the emulator take large, but precise, timesteps, avoiding the creation of extra timesteps whose only purpose is to update analog state variables.

When using this approach, special attention must be paid to the modeling of channel dynamics. The channel input consists of a piecewise-constant waveform driven by the transmitter (TX), and the channel output is sampled at a non-uniform interval due to jitter in the receiver (RX). The overall channel behavior is described by the following equation:

$$y = \sum_{i=1}^{\infty} x_i \cdot \left(f\left(t_i\right) - f\left(t_{i-1}\right) \right) \tag{6.2}$$

where x_i is the i-th TX data level (most recent first), t_i is the time that has elapsed since the i-th

data transition, and $f(t)$ is the step response of the channel.

A practical implementation must truncate Equation 6.2 to a finite number of terms, n, which is essentially the number of bits that contribute to intersymbol interference (ISI). We can bound the residual error of that truncation, assuming the input is bounded between $\pm R$ as:

$$e \leq \sum_{i=n+1}^{\infty} R \cdot |f(t_i) - f(t_{i-1})| \tag{6.3}$$

If we further assume that $f(t)$ has entered a monotonic settling regime by t_i (i.e., $f(t_i) \geq f(t_{i-1})$), then the error bound simplifies to:

$$e \leq R \cdot (f(\infty) - f(t_n)) \tag{6.4}$$

In other words, the relative error is the amount of settling that remains at the truncation point. For example, modeling a channel response to within 1% requires that the step response has settled to within 99% of its final value by the truncation point.

6.3.2 Low-level emulator

In building an emulator for DragonPHY, I first tried a low-level approach, illustrated in Fig. 6.4. A 16 GHz oscillator model drives a pseudorandom binary sequence (PRBS) generator that produces the channel input; that clock signal is halved twice to produce a 4 GHz clock signal that drives the four PIs. The PIs are modeled as digitally-controlled delay lines with jitter, and they each drive a bank of four ADC models. The ADCs sample the channel output, with Gaussian noise added, to produce measurements that go to the DragonPHY digital core, which was essentially unmodified for emulation.

In this approach, there are four distinct AMS blocks whose models were generated by **msdsl**: the PIs, the ADCs, the channel, and the 16 GHz oscillator. Many of the previously described features of the emulation framework are put to use in the low-level emulator, such as variable timestep emulation, which involves the clock edge events generated by four independent PIs and the 16 GHz oscillator. **msdsl**'s arbitrary function support is used to implement the channel step response, $f(t)$, that appears in Equation 6.2, and the Gaussian noise generator is used to model PI jitter and ADC noise.

This low-level approach was a direct translation of how DragonPHY was being modeled for CPU-based simulation, with the emulator implementing features that would normally be handled by a simulator. As a result, it made the process of porting simulation to emulation conceptually straightforward.

However, after building the emulator, I found that it provided only mediocre performance. Detailed results will be discussed later, but broadly speaking, the lackluster performance was caused

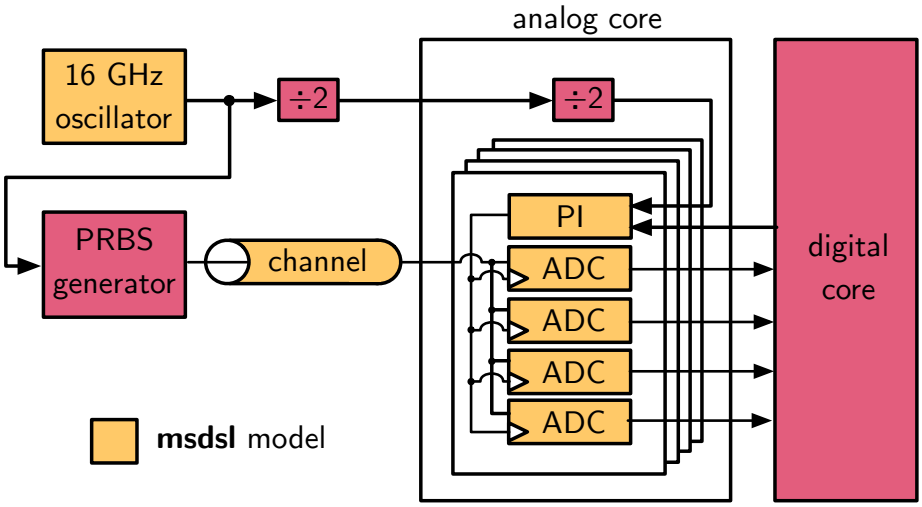


Figure 6.4: Low-level DragonPHY emulator, in which synthesizable AMS models are used within the existing hierarchy of the analog core.

by a mismatch between the parallelism of the DragonPHY design and the parallelism of the emulation models. In one cycle, DragonPHY's digital core takes in 16 ADC samples and produces 16 bit estimates; its emulation throughput could therefore be as high as 16 bits/cycle. But since the low-level emulator produces only one ADC sample per cycle, DragonPHY's digital core was forced to run 16x slower.

6.3.3 High-level emulator

Having made that observation, I wanted to try modeling DragonPHY's analog core as producing 16 ADC samples per emulation cycle, so as to match the parallelism of the digital core. This required the entire analog core to be replaced by a single AMS macromodel, encompassing the behavior of the ADCs, the PIs, and the channel dynamics (Fig. 6.5). As a result, I result call this the "high-level" modeling approach.

In the high-level emulator, time moves forward in increments of the 1 GHz clock rate of the digital core, rather than the 16 GHz clock rate at which bits are being transmitted, keeping the digital core more active than in the low-level emulator. This requires parallel computation of the RX ADC samples, since they must all be provided to the digital core at once.

The most straightforward way to achieve that would have been to create 16 instances of the channel model used in the low-level architecture. However, since this would have required more resources than were available on the FPGA, I had to use a channel model with a lower resource footprint. That model computed the terms in Equation 6.2 in batches, over several clock cycles, with the batch size set as a compile-time parameter.

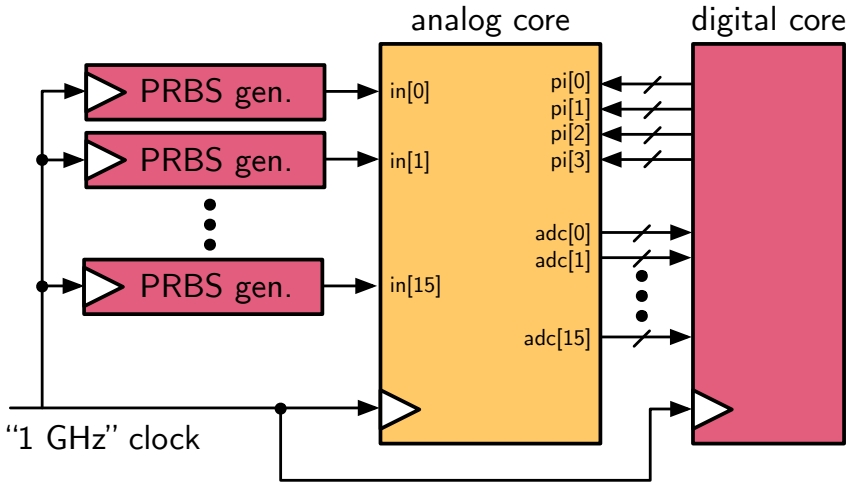


Figure 6.5: High-level DragonPHY emulator, in which the analog core is replaced by a single macromodel, and the behaviors of all 16 ADCs are modeled in parallel.

Since the AMS macromodel produces ADC samples in parallel, the transmitted bits also need to be produced in parallel. Fortunately, because the TX data source is an LFSR-based PRBS generator, this parallelization could be implemented in a straightforward way that generated exactly the same sequence of bits that was produced in the low-level case. I did this by creating 16 instances of the original PRBS generator that were given special seed values, such that the n-th PRBS generator produced a bit sequence equal to the original sequence shifted by n, then decimated by 16. This technique leverages a useful property of maximal-length LFSR sequences, namely that decimation by a power of two is equivalent to shifting the sequence by some amount [22].

6.3.4 Architecture comparison

The key advantage of the high-level architecture is speed. This is due not only to its increased parallelism, but also to the reduction in the infrastructure that is needed to manage variable timestep requests, which allows for a faster emulator cycle time. Although the high-level architecture is not a direct mapping of a CPU-based simulation, I found that it was easier to develop, because only one AMS model needed to be created and tested.

The main drawback of the high-level approach is that it can miss some details in the design. For example, a high-level block might have hundreds of I/Os whose behaviors must be modeled if they are to be included in emulation. In the low-level modeling approach, the effects of those I/Os are often modeled "automatically" through the connections between lower-level models and the digital logic that glues them together.

That said, I have found the high-level modeling approach to be a better fit for emulation. For one, it is easier to integrate and maintain a few high-level models than a large number of low-level

Table 6.1: DragonPHY Throughput Comparison

Approach	Real Number Format	Platform	Throughput (Mb/s)
B. Lim [41]	N/A	CPU	0.0071
DragonPHY Sim.	N/A	CPU	0.0143
High-Level Emu.	Floating-Point	FPGA	4.702
Low-Level Emu.	Fixed-Point	FPGA	4.996
S. Herbst [30]	Fixed-Point	FPGA	10.0
High-Level Emu.	Fixed-Point	FPGA	79.94

models. In addition, given that emulation performs poorly when there are many independent digital events, the high-level approach makes it easier to keep the number of events under control. Even though the high-level approach doesn't provide the same level of modeling fidelity as the low-level approach, it is a good fit for the tasks an emulator has to support: pre-silicon FW/SW development and running long tests. For other tasks, such as connectivity and coverage tests, it is better to use computer simulation, rather than low-level emulation.

6.3.5 Experimental results

Both the low- and high-level DragonPHY emulators were constructed using the open-source emulation framework and run on a Xilinx ZC706 board, which includes a Xilinx XC7Z045 FPGA. Since the framework makes it easy to switch real-number representations, I built two versions of the high-level emulator: one using a fixed-point representation and the other using floating-point. The simulation baseline was run on an Intel Xeon Silver 4214 CPU @ 2.20GHz, with 125 GB RAM, using the Cadence Xcelium simulator.

Throughput Table 6.1 compares the throughput, defined as bits per second processed by the DUT, of our emulators and other CPU- and FPGA-based approaches. The result from B. Lim [41] is the fastest published RTL-level high-speed link simulation of which I am aware; it used a PWL waveform representation as a means to create efficient behavioral models of AMS blocks. I also created my own simulation baseline for DragonPHY using B. Lim's PWL approach, with care taken to use the same level of modeling detail that was used in the low- and high-level emulators.

The fixed-point high-level emulator was 5,590x faster than those CPU-based simulations. To put that in perspective, it could process one trillion bits in 3.5 hours, while CPU simulation would take 2.2 years to complete such a task. Hence, while CPU simulation necessitates extrapolation over orders of magnitude to estimate BER, the high-level emulator brings that task into the realm of direct measurement.

Table 6.2: DragonPHY Emulator Latency Comparison

Approach	Run Time (s)
VIO, bit-level	23,904
FiM, bit-level	186.6
FiM, transaction-level	3.76

The low-level DragonPHY emulator ran about 16x slower than the high-level version for two reasons: first, the high-level architecture could use a 50% faster emulator clock because its critical path was shorter, and second, it reduced the average number of clock cycles per transmitted bit from 4 to 0.375.

In studying how the DragonPHY emulator stacks up to other high-speed link emulators, there is unfortunately only one other published result to serve as a comparison point: my own study from ICCAD 2018 [30]. Since the link design in that study was simpler than DragonPHY, the low-level DragonPHY emulator was about 2x slower than the ICCAD '18 result. However, the high-level DragonPHY emulator was about 8x faster.

As a final throughput comparison, I used **svreal** to switch real-number operations to floating-point in the high-level emulator, resulting in a 17x slowdown. This was due to two factors: first, I had to reduce the emulator clock frequency by 3x because of the slower speed of floating-point operations, and second, I had to reduce the parallelism of channel dynamics calculations by 5.6x because of the increased resource utilization. This illustrates the value of using **svreal**'s automated fixed-point formatting system, which makes it straightforward to boost performance by an order of magnitude as compared to a floating-point implementation.

Latency Since the emulation framework supports both VIO- and FiM-based interactive tests, I used DragonPHY as a test case to explore the relative performance between those two methods, in terms of the time taken to send a series of emulated JTAG commands to the DUT to set up and run a BER test.

Table 6.2 shows the results of that experiment. The VIO-based approach took 6.64 hr, which is clearly too long for real-world use. For the FiM approach, I looked at two different implementations. The first was a bit-level approach, where the host computer toggled individual DUT pins through the PS; this was 128x faster than the VIO-based approach, but still took 3.11 min. The second FiM approach was transaction-level, where the PS firmware received register read/write commands from the host computer and implemented them at bit-level. This brought the runtime for the startup task down to 3.76 sec, which is a speedup of 6,357x as compared to the VIO-based approach.

The takeaway is that switching from VIO to FiM can immediately provide a reduction in latency of two orders of magnitude, and a further speedup of almost two orders of magnitude can be achieved by implementing bit-level tasks in firmware on the FPGA, rather than on the host computer.

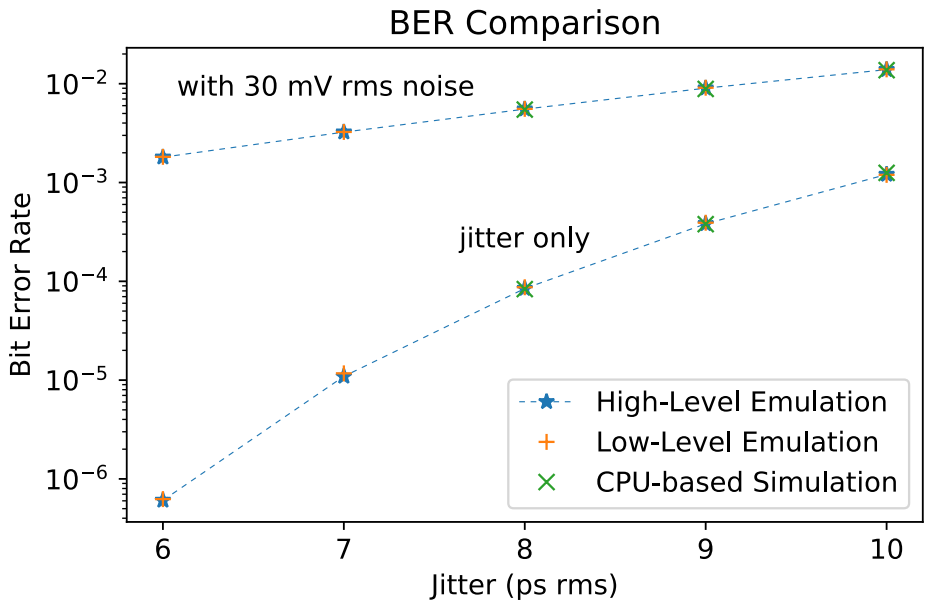


Figure 6.6: Comparison of BER predicted by the DragonPHY simulation baseline and both emulator architectures. The lower curve represents the effect of jitter alone, while the upper curve includes a fixed level of voltage noise.

Accuracy I evaluated the accuracy of the DragonPHY emulator by comparing the BER predicted by the high- and low-level implementations to that of the simulation baseline. I chose to compare BER, rather than individual analog waveforms, because BER is a single number, observable at the system level, that depends on all of the key features provided by the emulation framework: analog dynamics, noise, and jitter.

Fig. 6.6 shows the comparison, using two noise scenarios. The lower curve shows the BER for varying amounts of jitter in the sampling times of the RX ADCs. The upper curve shows the BER when 30 mVrms input-referred noise was added to the ADC. The BER predictions are in good agreement; none vary by more than 7.5%.

For this experiment, the BER was intentionally skewed higher than what would be desirable for a real link by adding fairly large amounts of noise and jitter. This was necessary because accurately measuring the probability of an unlikely event (namely, a bit error) requires a number of observations that is inversely proportional to the event probability; that number quickly becomes too large for the baseline simulation.

More precisely, the 95% Wald confidence interval (CI) of a probability estimate p made from n trials is $\pm 1.96\sqrt{p\,(1-p)\,/n}$.[1] With the BER as p, and assuming that bit errors are very rare, the

[1]The Wald CI is well-known to perform poorly for very likely or very unlikely events, and/or when n is small. However, adding two successes and two failures the observation set significantly improves its performance (Agresti and Coull [3]). Hence, as long as number of observed successes and failures are both much greater than two, the conclusions that follow should hold.

Table 6.3: DragonPHY Emulator Resource Utilization (ZC706)

Approach	Real Number Format	LUT	FF	DSP	BRAM
Low-level	Fixed-point	31.2%	5.6%	20.8%	13.6%
High-level	Fixed-point	44.5%	6.1%	94.4%	36.7%
High-level	Floating-point	75.2%	5.5%	60.4%	13.7%

Table 6.4: Low-Level Emulator: Resource Utilization by Model (ZC706)

AMS Model	LUT	FF	DSP	BRAM	Number of Instances
Lossy Channel	4,529	1,026	75	25	1x
ADC	555	53	5	0.5	18x
Phase Interpolator	864	116	5	0.5	4x
16 GHz Oscillator	94	26	0	0	1x
Total Available	*218,600*	*437,200*	*900*	*545*	-

CI simplifies to $\pm 1.96 \cdot \sqrt{p/n}$. By dividing that expression by p, we arrive at an expression for the relative error bars on the BER: $\pm 1.96\sqrt{1/(pn)}$.

In other words, the relative error in the BER estimate is inversely proportional to the square root of the number of *errors* observed (not the number of trials). I aimed for a minimum of 1,000 observed errors to place the relative error bars at about ±6%. Since the DragonPHY CPU baseline could only run at 14.3 kb/s, this meant that getting a baseline estimate of a BER around 10^{-4} took 11-12 minutes, and with lower BERs quickly becoming impractical.

Resource Utilization The resource utilization for both emulator architectures is shown in Table 6.3. The low-level architecture was fairly lightweight, not using more than about a quarter of any given resource. Its detailed resource utilization is shown in Table 6.4, demonstrating that each AMS model instance has a small footprint, with the most resource-intensive being the channel model. Taken together, AMS models only accounted for about a third of the LUT utilization in the low-level emulator, with most of the remaining LUTs used by DragonPHY's digital core. While AMS models did account for most (70%) of the emulator's DSP utilization, this did not cause any issues, because the total DSP utilization only represented 21% of the FPGA's available DSP slices.

For the fixed- and floating-point implementations of the high-level architecture, I optimized for performance, aiming to max out one or more FPGA resources by adjusting the parallelism of channel dynamics calculations. In the fixed-point case, DSP slices were the bottleneck, since fixed-point arithmetic operations only consumed a few dozen LUTs each. For the floating-point case, LUTs were the bottleneck, as those same operations consumed hundreds of LUTs apiece.

In all cases, BRAM utilization was driven largely by the number of instances of the step response function, since each instance required an independently-addressable coefficient lookup table.

Productivity Although it is difficult to precisely quantify productivity benefits, I used source lines of code (SLOC) as a rough metric. SLOC does not include comment lines and blank lines, so it is a slightly more direct measure of code complexity than a raw line count. However, since SLOC requires a basic understanding of language syntax, its measurement requires a specialized tool; I chose `cloc` [15] because it supports the key languages used in building emulators: Verilog, Python, TCL, C, and YAML.

Unfortunately, there is not much to which I can compare the DragonPHY SLOC measurements. My own work for ICCAD '18 is the only other example of a publicly-available implementation of a high-speed link emulator, so I used that as the comparison point. Since that work involved a simpler design, the SLOC comparison is not very precise, but at least is not biased towards the new emulation framework.

The SLOC measurements for the ICCAD '18 and DragonPHY emulators are shown in Fig. 6.7. I included code for AMS synthesizable models and emulator infrastructure, but not the RTL of digital blocks that were unmodified for emulation. Overall, the amount of AMS emulation code was approximately cut in half by using the open-source framework. Interestingly, the biggest difference was in Python, not Verilog; I had previously used some ad-hoc model generators and build automation that could be implemented more succinctly using the framework.

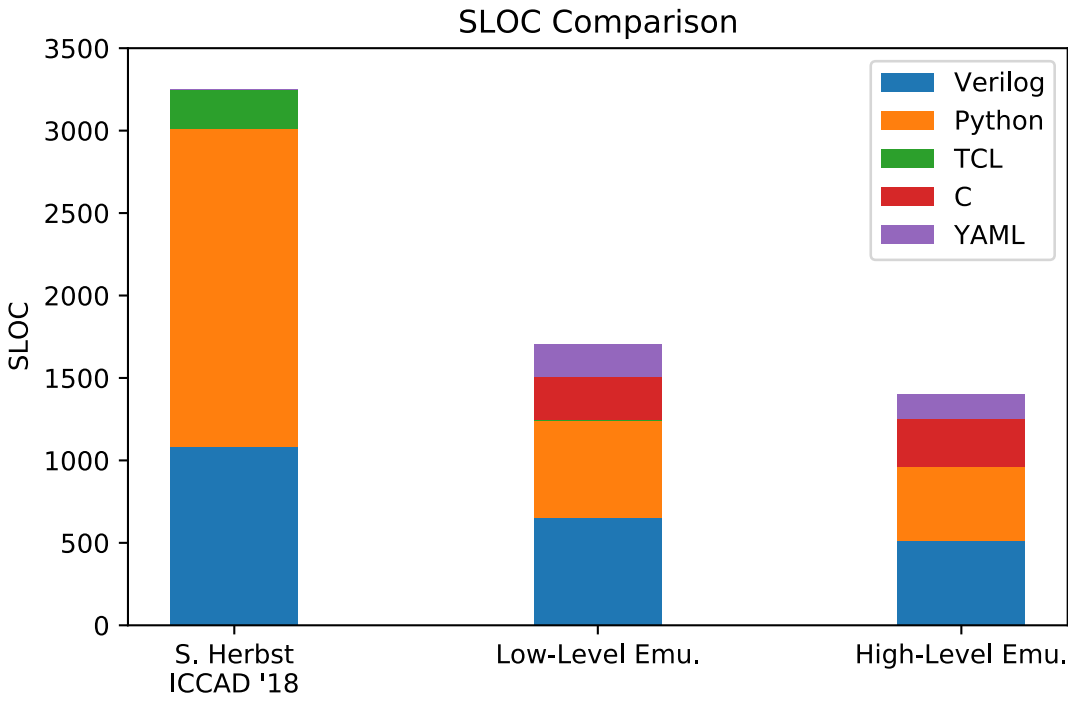


Figure 6.7: Composition of AMS modeling and infrastructure code for several high-speed link emulators.

Chapter 7

Conclusion

In this thesis, I have shown that emulating AMS chip designs on off-the-shelf FPGA boards can achieve speedups of 2-3 orders of magnitude as compared to traditional computer simulation methods. These speedups open a wide range of possibilities for AMS chip development. On one hand, a fast emulator can be used for pre-silicon development of firmware that will run on-chip. But it can also be used for developing software that will interact with the chip, reducing bringup time. Another possibility is to use the emulator as a stand-in for developing factory test infrastructure prior to silicon availability.

However, in order to achieve these speedups, I demonstrated that it is essential to use the right modeling techniques for analog blocks. In particular, much more so than in simulation, it is important to reduce or eliminate "analog-only" timesteps, which are timesteps taken in an emulator while digital parts of the design sit idle. To address that issue, I introduced a new spline-based approach for analog modeling and demonstrated that it can reduce the number of analog-only timesteps by about two orders of magnitude in modeling a high-speed link.

I also suggested that, when possible, one should match the parallelism of analog emulation models to the level of parallelism inherent in the chip's digital circuits. It is possible to achieve a speedup of more than an order of magnitude by applying this technique, as I demonstrated with the "high-level" DragonPHY emulator. In that case, a DSP core processed blocks of 16 ADC samples at a time, so I adapted the analog modeling approach to generate all 16 ADC samples in one emulation cycle, rather than in successive cycles.

In some ways, the spline approach and high-level modeling strategy are similar: both spend additional FPGA resources to increase the level of parallelism of synthesizable models to the extent that they are no longer the bottleneck in emulator performance. It is worthwhile to do this, because unlike in an ASIC design, there is little downside to using more resources on an FPGA, as long as the design still fits. At the same time, as I demonstrated through several experiments, my modeling approaches do not require extravagant resource utilization; they fit on middle-of-the-road FPGA

boards.

As I explored the analog emulation space, I discovered that there was no complete, publicly available framework for emulating AMS chip designs, and this made it difficult to apply traditional analog modeling strategies, as well as to experiment with the new techniques I was developing. This motivated me to develop the open-source AMS emulation framework that is described in this thesis.

The framework consists of three tools: **msdsl** [28], for writing synthesizable AMS models in Python, **svreal** [29], for implementing synthesizable real-number operations efficiently and accurately, and **anasymod** [60], which presents a simulator-like abstraction of FPGA boards. I collaborated with Infineon on the framework to ensure its industry applicability, and they in fact led the development of one of the tools, **anasymod**.

My goal in partitioning the framework was to achieve a degree of modularity, where each tool could provide standalone features that were useful in their own right. This seems to have been somewhat successful for **svreal**, which has attracted some users outside of the emulation space, who are just interested in an easy-to-use fixed-point Verilog type.

As I showed in a lines-of-code comparison between the DragonPHY emulator, implemented with the emulation framework, and a simpler high-speed link emulator, implemented in an ad-hoc fashion, using the emulation framework reduces the amount of code required by at least a factor of two. In the future, even greater productivity benefits might be possible by connecting the emulation framework to an AMS behavioral model generator, such as **fixture** [64], with the goal of generating both simulation and emulation models from a single source of truth.

To close, it is my hope that the new synthesizable AMS modeling techniques I developed will improve the payoff in emulating future AMS designs, and that the open-source emulation framework will reduce barriers to achieving that payoff. Taken together, I believe the work represents a step towards breaking through the verification bottleneck in AMS design, and, as free open-source software, is part of an ongoing, broad effort to democratize hardware design.

Appendix A

Integral of a Matrix Exponential Times a Polynomial

From Chapter 4, the spline implementation of a state-space system includes the integral:

$$\tilde{f}_{jk}(t) = \int_0^t e^{(t-\tau)A} b \left(\frac{\tau - j\Delta t_h}{\Delta t_h} \right)^k \tilde{H}_j(\tau)\, d\tau \tag{A.1}$$

This appendix describes an efficient way to compute this integral, including the derivation of a formula for the integral of matrix times a polynomial, which may be useful for other purposes.

First, observe that since $\tilde{H}_j$ is a rectangular window function between $j\Delta t_h$ and $(j+1)\Delta t_h$, we can eliminate it by changing the integration interval to:

$$[t_-, t_+] = [0, t] \cap [j\Delta t_h, (j+1)\Delta t_h] \tag{A.2}$$

This leads to the following:

$$\tilde{f}_{jk}(t) = e^{tA} \cdot \left(\int_{t_-}^{t_+} e^{-\tau A} \left(\frac{\tau - j\Delta t_h}{\Delta t_h} \right)^k d\tau \right) \cdot b \tag{A.3}$$

Next, make the change of variables $s = (\tau - j\Delta t_h)/\Delta t_h$, resulting in:

$$\tilde{f}_{jk}(t) = \Delta t_h \cdot e^{tA} \cdot \left(\int_{t_-/\Delta t_h - j}^{t_+/\Delta t_h - j} e^{-(s+j)\Delta t_h A} s^k ds \right) \cdot b \tag{A.4}$$

$$= \Delta t_h \cdot e^{(t - j\Delta t_h)A} \cdot \left(\int_{s_-}^{s_+} e^{-s\Delta t_h A} s^k ds \right) \cdot b \tag{A.5}$$

The computation now boils down to the integral $\int e^{sM} s^k ds$, with $M = -\Delta t_h A$. I conclude by

demonstrating how to compute that integral without explicit numerical integration. Using integration by parts, we have:

$$\int e^{sM} s^k ds = M^{-1} e^{sM} s^k - \int M^{-1} e^{sM} k s^{k-1} ds \tag{A.6}$$

$$= M^{-1} e^{sM} s^k - k M^{-1} \int e^{sM} s^{k-1} ds \tag{A.7}$$

This leverages the property that $de^{sM}/ds = Me^{sM}$ [57], and therefore an antiderivative of e^{sM} is $M^{-1}e^{sM}$, so long as M is invertible. (I make that assumption for the rest of this analysis.)

Repeating integration by parts once more yields:

$$\int e^{sM} s^k ds = M^{-1} e^{sM} s^k - k M^{-1} \left(M^{-1} e^{sM} s^{k-1} - (k-1) M^{-1} \int e^{sM} s^{k-2} ds \right) \tag{A.8}$$

$$= M^{-1} e^{sM} s^k - k M^{-2} e^{sM} s^{k-1} + k(k-1) M^{-2} \int e^{sM} s^{k-2} ds \tag{A.9}$$

At this point, the pattern is becoming clear: each successive term picks up a factor of M^{-1} and a factor of k further decremented, and its sign flips. We can generalize this into the following formula:

$$\int e^{sM} s^k ds = \left(\sum_{i=0}^{k} (-1)^{k-i} \cdot \frac{k!}{i!} \cdot M^{-k+i-1} \cdot s^i \right) \cdot e^{sM} \tag{A.10}$$

As a sanity check, one can make M a scalar and verify the result matches the formula that could be found in a textbook on single-variable calculus.

Having gone through this analysis, the original integral in the spline points implementation can be computed without explicit numerical integration. In practice, this makes the difference between a model that compiles in around a second, as compared to several minutes without this optimization.

Appendix B

Generating a Step Response from S-Parameters

Converting measured S-parameters to a step response requires several distinct steps:

1. Convert 4-port S-parameters (i.e. 4x4 matrix) to 2-port S-parameters (i.e., 2x2 matrix).
2. Convert 2-port S-parameters to a transfer function.
3. Convert the transfer function into an impulse response.
4. Integrate the impulse response to obtain the step response.

This appendix covers the first three steps; the last step can be efficiently performed with cumulative trapezoidal numerical integration.

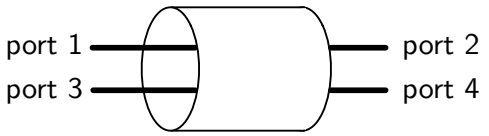
B.1 Computing 2-port S-parameters

S-parameters of a lossy channel are often reported as 4-port measurements, with each wire in a data pair considered a single-ended signal (Fig. B.1a). However, in order to model the behavior of a channel with a single transfer function, we need to calculate the 2-port S-parameters of this system, which represent only the differential part of incident and reflected waves (Fig. B.1b).

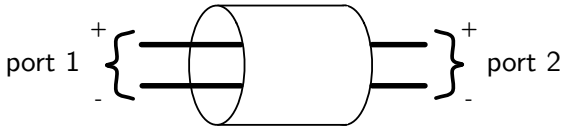
Hence, we are looking for a 2-port differential-to-differential S-parameter matrix:

$$S_{dd} = \begin{bmatrix} S_{d1d1} & S_{d1d2} \\ S_{d2d1} & S_{d2d2} \end{bmatrix} \tag{B.1}$$

As others have shown [16], these S-parameters can be calculated from the 4-port measurements as



(a) 4-port representation of a lossy channel, where each port represents one side of a data pair. The order shown here is the one typically used in Touchstone files.



(b) 2-port representation of a lossy channel, which considers the differential part of incident and reflected waves on each side of the channel.

Figure B.1: S-parameter representations of a lossy channel.

follows:

$$S_{d1d1} = 1/2 \cdot (S_{11} - S_{13} - S_{31} + S_{33}) \tag{B.2}$$

$$S_{d1d2} = 1/2 \cdot (S_{12} - S_{14} - S_{32} + S_{34}) \tag{B.3}$$

$$S_{d2d1} = 1/2 \cdot (S_{21} - S_{23} - S_{41} + S_{43}) \tag{B.4}$$

$$S_{d2d2} = 1/2 \cdot (S_{22} - S_{24} - S_{42} + S_{44}) \tag{B.5}$$

B.2 Computing the transfer function

With a 2-port S-parameter matrix in hand, we can compute the transfer function from one side of a channel to the other, given the source impedance, Z_S, on the transmitter side, load impedance, Z_L, on the receiver side, and characteristic impedance, Z_0, used in reporting the S parameters. Using a formula from MATLAB documentation [47], the transfer function at a given frequency is:

$$H = \frac{Z_S + Z_S^*}{Z_S^*} \cdot \frac{S_{21} \cdot (1 + \Gamma_L) \cdot (1 - \Gamma_S)}{2 \cdot (1 - S_{22} \cdot \Gamma_L) \cdot (1 - \Gamma_{in} \cdot \Gamma_S)} \tag{B.6}$$

where:

$$\Gamma_L = (Z_L - Z_0)/(Z_L + Z_0) \tag{B.7}$$

$$\Gamma_S = (Z_S - Z_0)/(Z_S + Z_0) \tag{B.8}$$

$$\Gamma_{in} = S_{11} + (S_{12} \cdot S_{21} \cdot \Gamma_L / (1 - S_{22} \cdot \Gamma_L))) \tag{B.9}$$

H should be calculated at each frequency where the S-parameters have been measured, resulting in a transfer function, $H(f)$. Since the transfer function is dependent on the source and load impedance, it has reflection behavior "baked in."

B.3 Computing the impulse response

The impulse response of the channel is the inverse Fourier transform of its transfer function:

$$h(t) = \int_{-\infty}^{\infty} e^{i2\pi ft} H(f)\, df \tag{B.10}$$

While the above integral could be computed for various values of t using general-purpose numerical integration, it is more computationally efficient to use the inverse fast Fourier transform (IFFT). However, we have to be careful with shifting and scaling to get the correct result. Truncating the original integration interval to $\pm f_s/2$, and taking advantage of the fact that the Fourier transform of a real signal is conjugate-symmetric:

$$h(t) \approx \int_{-f_s/2}^{f_s/2} e^{i2\pi ft} H(f)\, df \tag{B.11}$$

$$= \int_{0}^{f_s/2} H(f)\, e^{i2\pi ft}\, df + \int_{f_s/2}^{f_s} H^*(f_s - f)\, e^{i2\pi ft}\, df \tag{B.12}$$

$$\approx \sum_{k=0}^{n-1} \tilde{H}(k\Delta f)\, e^{i2\pi k\Delta ft}\, \Delta f \tag{B.13}$$

$$= \frac{f_s}{n} \cdot \sum_{k=0}^{n-1} \tilde{H}\left(\frac{k}{n} f_s\right) e^{i2\pi k f_s t/n} \tag{B.14}$$

where $\Delta f = f_s/n$ and

$$\tilde{H}(f) = \begin{cases} H(f), & \text{for } 0 \le f \le f_s/2 \\ H^*(f_s - f), & \text{for } f_s/2 < k \le f_s \end{cases} \tag{B.15}$$

Comparing this result to the typical definition of the IFFT,[1] we conclude that:

$$h\left(\frac{0}{f_s}\right), h\left(\frac{1}{f_s}\right), \ldots, h\left(\frac{n-1}{f_s}\right) \approx f_s \cdot \text{IFFT}\left\{\tilde{H}\left(\frac{0}{n} f_s\right), \tilde{H}\left(\frac{1}{n} f_s\right), \ldots, \tilde{H}\left(\frac{n-1}{n} f_s\right)\right\} \tag{B.16}$$

In other words, the time resolution of the computed impulse response is the inverse of the sampling frequency, $1/f_s$, while the duration is approximately the inverse of the frequency resolution of measurements.

As a practical matter, $\tilde{H}(0)$ and $\tilde{H}(f_s/2)$ must real in order for the assumption of conjugate symmetry to hold (if not, the impulse response will be complex). I typically accomplish that by clipping the imaginary part of $\tilde{H}(0)$ to zero (since that represents measurement error), and by setting $\tilde{H}(f_s/2)$ to zero, effectively applying a box filter that cuts off at, not above, $f_s/2$.

[1] A factor of $1/n$ is typically included, so the j-th element in the IFFT of Y is $\frac{1}{n} \cdot \sum_{k=0}^{n-1} Y_k \cdot e^{i2\pi jk/n}$.

Bibliography

[1] IEEE Standard Verilog Hardware Description Language. *IEEE Std 1364-2001*, pages 318–319, 2001.

[2] IEEE Standard for Digitizing Waveform Recorders. *IEEE Std 1057-2017 (Revision of IEEE Std 1057-2007)*, pages 1–0, 2018. doi:10.1109/IEEESTD.2018.8291741.

[3] Alan Agresti and Brent A. Coull. Approximate Is Better than "Exact" for Interval Estimation of Binomial Proportions. *The American Statistician*, 52(2):119–126, 1998. URL: http://www.jstor.org/stable/2685469.

[4] P. Alfke. Efficient Shift Registers, LFSR Counters, and Long Pseudo-Random Sequence Generators, 1996. URL: https://www.xilinx.com/support/documentation/application_notes/xapp052.pdf.

[5] A. Amid, D. Biancolin, A. Gonzalez, D. Grubb, S. Karandikar, H. Liew, A. Magyar, H. Mao, A. Ou, N. Pemberton, P. Rigge, C. Schmidt, J. Wright, J. Zhao, Y. S. Shao, K. Asanović, and B. Nikolić. Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs. *IEEE Micro*, 40(4):10–21, 2020.

[6] Sameh Asaad, Ralph Bellofatto, Bernard Brezzo, Chuck Haymes, Mohit Kapur, Benjamin Parker, Thomas Roewer, Proshanta Saha, Todd Takken, and José Tierno. A Cycle-Accurate, Cycle-Reproducible Multi-FPGA System for Accelerating Multi-Core Processor Simulation. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '12, page 153–162, New York, NY, USA, 2012. Association for Computing Machinery. doi:10.1145/2145694.2145720.

[7] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović. Chisel: Constructing hardware in a Scala embedded language. In *DAC Design Automation Conference 2012*, pages 1212–1221, 2012.

[8] D. Bertsekas and J. Tsitsiklis. *Introduction to probability*, chapter 3. Athena Scientific, Belmont, Mass., 2002.

[9] R. Bhattacharya, S. Biswas, and S. Mukhopadhyay. FPGA based chip emulation system for test development of analog and mixed signal circuits: A case study of DC–DC buck converter. *Measurement*, 45(8):1997 – 2020, 2012. URL: http://www.sciencedirect.com/science/article/pii/S0263224112001923, doi:https://doi.org/10.1016/j.measurement.2012.04.022.

[10] C. De Boor. *A practical guide to splines*, chapter 4. Springer-Verlag, New York, 1978.

[11] Emmanuel Boutillon, Jean-Luc Danger, and Adel Ghazel. Design of high speed AWGN communication channel emulator. *Analog Integrated Circuits and Signal Processing*, 34(2):133–142, 2003.

[12] G. E. P. Box and Mervin E. Muller. A Note on the Generation of Random Normal Deviates. *Ann. Math. Statist.*, 29(2):610–611, 06 1958. doi:10.1214/aoms/1177706645.

[13] Cadence. Cadence Palladium. URL: https://www.cadence.com/en_US/home/tools/system-design-and-verification/acceleration-and-emulation/palladium-z1.html.

[14] Steve Carlson. Mixing It Up in Hardware (an Advantest Case Study in Faster Full-Chip Simulations), 2014. URL: https://community.cadence.com/cadence_blogs_8/b/ms/posts/mixing-it-up-in-hardware-steve-carlson-the-low-road.

[15] Al Danial. `cloc`: Count Lines of Code, 2021. URL: https://github.com/AlDanial/cloc.

[16] W. Fan, A. Lu, L.L. Wai, and B.K. Lok. Mixed-mode S-parameter characterization of differential structures. In *Proceedings of the 5th Electronics Packaging Technology Conference (EPTC 2003)*, pages 533–537, 2003. doi:10.1109/EPTC.2003.1271579.

[17] A. Fernandez-Alvarez, M. Portela-Garcia, and M. Garcia-Valderas. FPGA-based HW/SW co-simulation system for mixed-signal circuits. In *2016 Conference on Design of Circuits and Integrated Systems (DCIS)*, pages 1–6, 2016.

[18] Alex Forencich. Verilog Mersenne Twister, 2016. URL: https://github.com/alexforencich/verilog-mersenne.

[19] IBIS Open Forum. Touchstone File Format Specification, 2009. URL: https://ibis.org/touchstone_ver2.0/touchstone_ver2_0.pdf.

[20] SkyWater Technology Foundry and Google. SkyWater Open Source PDK, 2020. URL: https://github.com/google/skywater-pdk.

[21] G. Franklin and J. Powell. *Digital control of dynamic systems*, chapter 3. Addison-Wesley Pub. Co., Reading, Mass., 1980.

[22] Solomon W. Golomb. *Shift Register Sequences*. Holden-Day, 1967. pg. 76.

[23] Mentor Graphics. Mentor Veloce Emulation Platform. URL: https://www.mentor.com/products/fv/emulation-systems/.

[24] P. Hanrahan. magma, 2021. URL: https://github.com/phanrahan/magma.

[25] John Hauser. Berkeley HardFloat, 2019. URL: http://www.jhauser.us/arithmetic/HardFloat.html.

[26] Thomas Henkel and Henriette Ossoinig. Timing-accurate emulation of a mixed-signal SoC using Palladium XP. CDNLive 2013, Munich, 2010.

[27] John L. Hennessy and David A. Patterson. A New Golden Age for Computer Architecture. *Commun. ACM*, 62(2):48–60, January 2019. URL: https://doi-org.stanford.idm.oclc.org/10.1145/3282307, doi:10.1145/3282307.

[28] S. Herbst. msdsl, 2021. URL: https://git.io/msdsl.

[29] S. Herbst. svreal, 2021. URL: https://git.io/svreal.

[30] S. Herbst, B. Lim, and M. Horowitz. Fast FPGA Emulation of Analog Dynamics in Digitally-Driven Systems. In *Proceedings of the International Conference on Computer-Aided Design*, ICCAD '18, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3240765.3240808.

[31] A. Iserles. *A First Course in the Numerical Analysis of Differential Equations.*, volume 2nd ed of *Cambridge Texts in Applied Mathematics*. Cambridge University Press, 2009. URL: https://stanford.idm.oclc.org/login?url=https://search.ebscohost.com/login.aspx?direct=true&db=nlebk&AN=400705&site=ehost-live&scope=site.

[32] J. Jang, M. Park, and J. Kim. An event-driven simulation methodology for integrated switching power supplies in systemverilog. In *Proceedings of the 50th Annual Design Automation Conference*, DAC '13, New York, NY, USA, 2013. Association for Computing Machinery. URL: https://doi-org.stanford.idm.oclc.org/10.1145/2463209.2488903, doi:10.1145/2463209.2488903.

[33] J. Jang, M. Park, D. Lee, and J. Kim. True event-driven simulation of analog/mixed-signal behaviors in SystemVerilog: A decision-feedback equalizing (DFE) receiver example. In *Proceedings of the IEEE 2012 Custom Integrated Circuits Conference*, pages 1–4, 2012. doi:10.1109/CICC.2012.6330558.

[34] T. Kailath. *Linear systems*, chapter 2. Prentice-Hall, Englewood Cliffs, N.J., 1980.

[35] S. Karandikar, H. Mao, D. Kim, D. Biancolin, A. Amid, D. Lee, N. Pemberton, E. Amaro, C. Schmidt, A. Chopra, Q. Huang, K. Kovacs, B. Nikolic, R. Katz, J. Bachrach, and K. Asanovic. FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 29–42, 2018.

[36] D. Kececioglu. *Reliabililty engineering handbook*, volume 1, chapter 7. Prentice-Hall, Englewood Cliffs, N.J., 1991.

[37] S. Kim, Z. Myers, S. Herbst, B. Lim, and M. Horowitz. Open-Source Synthesizable Analog Blocks for High-Speed Link Designs: 20-GS/s 5b ENOB Analog-to-Digital Converter and 5-GHz Phase Interpolator. In *2020 IEEE Symposium on VLSI Circuits*, pages 1–2, 2020.

[38] Seehyun Kim and Wonyong Sung. A floating-point to fixed-point assembly program translator for the TMS 320C25. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 41(11):730–739, 1994. doi:10.1109/82.331543.

[39] E. Lee and D. Messerschmitt. *Digital communication*, chapter 5. Kluwer Academic Publishers, Boston, 1988.

[40] S. Liao and M. Horowitz. A Verilog piecewise-linear analog behavior model for mixed-signal validation. In *Proceedings of the IEEE 2013 Custom Integrated Circuits Conference*, pages 1–5, 2013. doi:10.1109/CICC.2013.6658461.

[41] B. C. Lim and M. Horowitz. Error Control and Limit Cycle Elimination in Event-Driven Piecewise Linear Analog Functional Models. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 63(1):23–33, 2016.

[42] Guangxi Liu. Gaussian Noise Generator. URL: http://opencores.org/project/gng.

[43] Chick Markley, Paul Rigge, Stevo Bailey, and Angie Wang. dsptools: A Library of Chisel3 Tools for Digital Signal Processing, 2016. URL: https://github.com/ucb-bar/dsptools.

[44] G. Marsaglia. The Marsaglia Random Number CDROM, with The Diehard Battery of Tests of Randomness, 1985.

[45] MathWorks. Simulink HDL Coder. URL: https://www.mathworks.com/products/hdl-coder.html.

[46] MathWorks. Simscape Electrical User's Guide, 2021. URL: https://www.mathworks.com/help/pdf_doc/physmod/sps/sps_ug.pdf.

[47] Inc. MathWorks. `s2tf`: Convert S-parameters of 2-port network to voltage or power-wave transfer function, 2021. URL: https://www.mathworks.com/help//rf/ref/s2tf.html.

[48] Maxim Integrated Products, Inc. Application Note 5384: Understanding Noise, ENOB, and Effective Resolution in Analog-to-Digital Converters, 2012. URL: https://www.maximintegrated.com/en/design/technical-documents/app-notes/5/5384.html.

[49] B. K. Mishra, S. Save, and R. Mane. A Frame Work for Model Based Designing of Analog Circuits Using Simulink. In *Proceedings of the International Conference & Workshop on Emerging Trends in Technology*, ICWET '11, page 1225–1228, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/1980022.1980290.

[50] Frank Austin Nothaft, Luis Fernandez, Stephen Cefali, Nishant Shah, Jacob Rael, and Luke Darnell. Pragma-Based Floating-to-Fixed Point Conversion for the Emulation of Analog Behavioral Models. In *Proceedings of the 2014 IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '14, page 633–640. IEEE Press, 2014.

[51] Travis E Oliphant. *A guide to NumPy*, volume 1. Trelgol Publishing USA, 2006.

[52] A. V. Oppenheim and R. W. Schafer. *Discrete-time signal processing*. Prentice Hall, Upper Saddle River, N.J., 2010.

[53] A. V. Oppenheim, A. S. Willsky, and S. H. Nawab. *Signals & Systems*, chapter 2. Prentice Hall, Upper Saddle River, N.J., 1997.

[54] A. W. Overhauser. Analytic Definition of Curves and Surfaces by Parabolic Blending. Technical Report SL68-40, Ford Motor Company Scientific Laboratory, May 1968. URL: https://arxiv.org/abs/cs/0503054.

[55] PCI-SIG. *PCI Express Base Specification Revision 3.1a*. 2015. URL: https://pcisig.com/specifications.

[56] W. Peters, E. Gong, C. Chen, and H. Kim. Improved HVM ATCA Measurement Data, June 2005. URL: https://www.ieee802.org/3/ap/public/jun05/peters_01_0605.pdf.

[57] K. B. Petersen and M. S. Pedersen. The Matrix Cookbook, November 2012. URL: https://www2.imm.dtu.dk/pubdb/pubs/3274-full.html.

[58] J. G. Reid. *Linear system fundamentals: continuous and discrete, classic and modern*, chapter 7. McGraw-Hill, New York, 1983.

[59] G. Rutsch, S. Fontanesi, S. Herbst, S. Tan Hee Yeng, A. Possemato, G. Formato, M. Horowitz, and W. Ecker. Boosting mixed-signal design productivity with FPGA-based methods throughout the chip design process. Design and Verification Conference in Europe, 2020. URL: https://dvcon-europe.org.

[60] G. Rutsch, S. Herbst, and S. Saravanan. anasymod, 2021. URL: https://git.io/anasymod.

[61] R. M. Sanchez, B. T. Reyes, A. L. Pola, and M. R. Hueda. An FPGA-based emulation platform for evaluation of time-interleaved ADC calibration systems. In *2016 IEEE 7th Latin American Symposium on Circuits Systems (LASCAS)*, pages 187–190, 2016.

[62] Mohamed Shalan and Tim Edwards. Building OpenLANE: A 130nm Openroad-Based Tapeout-Proven Flow. In *Proceedings of the 39th International Conference on Computer-Aided Design*, ICCAD '20, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3400302.3415735.

[63] C. E. Shannon. Communication in the Presence of Noise. *Proceedings of the IRE*, 37(1):10–21, 1949. doi:10.1109/JRPROC.1949.232969.

[64] D. Stanley. fixture, 2021. URL: https://github.com/standanley/fixture.

[65] Synopsys. Synopsys ZeBu. URL: https://www.synopsys.com/verification/emulation.html.

[66] Cadence Design Systems. Application Notes on Direct Time-Domain Noise Analysis using Virtuoso Spectre. 2006.

[67] SIMetrix Technologies. Simetrix. URL: https://www.simetrix.co.uk.

[68] 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)*, pages 1–4, 2017.

[69] George C. Verghese, Malik E. Elbuluk, and John G. Kassakian. A General Approach to Sampled-Data Modeling for Power Electronic Circuits. *IEEE Transactions on Power Electronics*, PE-1(2):76–89, 1986. doi:10.1109/TPEL.1986.4766286.

[70] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, CJ Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake Vand erPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020. doi:https://doi.org/10.1038/s41592-019-0686-2.

[71] G. Wang and Y. Chiu. Fast FPGA emulation of background-calibrated SAR ADC with internal redundancy dithering. In *Proceedings of the IEEE 2013 Custom Integrated Circuits Conference*, pages 1–4, 2013.

[72] David L. Weaver. *OpenSPARC Internals*, chapter 2. Sun Microsystems, Inc., Santa Clara, CA, 2008. URL: https://www.oracle.com/technetwork/systems/opensparc/opensparc-internals-book-1500271.pdf.

[73] W. Wu, Y. Chen, Y. Ma, C. J. Liu, J. Jou, S. Pamarti, and L. He. Wave digital filter based analog circuit emulation on FPGA. In *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1286–1289, 2016.

[74] Xilinx. 7 Series DSP48E1 Slice User Guide (UG479), 2018. URL: https://www.xilinx.com/support/documentation/user_guides/ug479_7Series_DSP48E1.pdf.

[75] Xilinx. High-Level Synthesis, 2020. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_1/ug902-vivado-high-level-synthesis.pdf.

[76] Xilinx. Model-Based DSP Design using System Generator, 2020. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_1/ug897-vivado-sysgen-user.pdf.