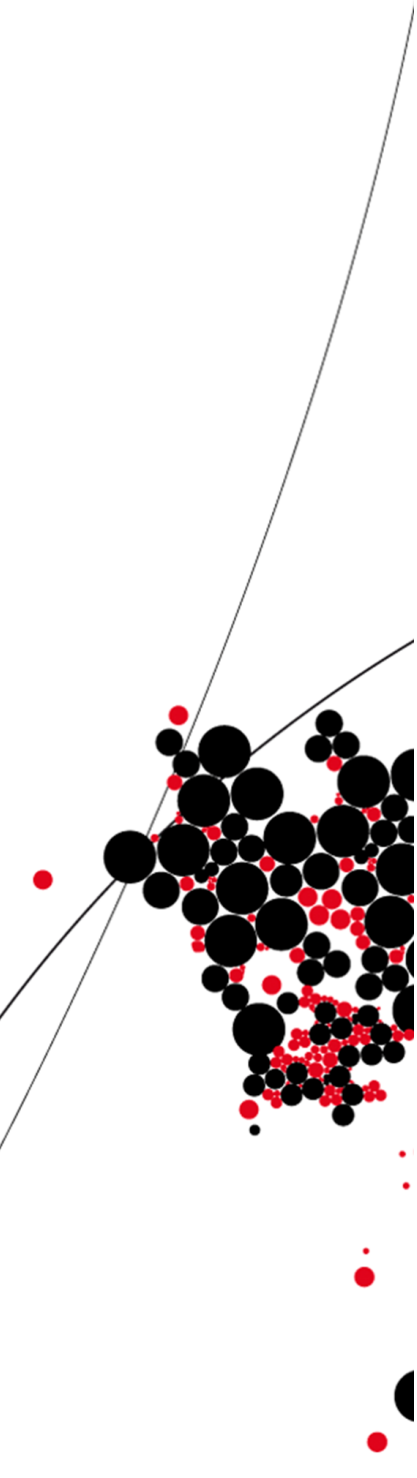
UNIVERSITY OF TWENTE.

Faculty of Electrical Engineering,
Mathematics & Computer Science

Emulation of Analog Mixed-Signal Circuits on an FPGA

N. Sulzer

MSc. Thesis
December 2022



Supervisors
prof. dr. ir. B. Nauta
dr.ir. M.S. Oude Alink
dr.ir. S.H. Gerez

Chair of Integrated Circuit Design
Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Abstract

Verifying *analog mixed-signal* (AMS) designs with conventional methods has the downsides of being expensive – in the case of *hardware-in-the-loop* (HIL) verification – or tedious – in the case of analog and digital co-simulation. Another verification technique is explored in this thesis: emulation using *field-programmable gate arrays* (FPGAs). Analog designs are modelled in synthesizable *register transfer level* (RTL) which can be run on an FPGA, sacrificing accuracy for faster run times. Based on the tools **svreal**, **msdsl** and **anasymod** developed at Stanford, analog designs are modelled in Python using various techniques, and *hardware description language* (HDL) descriptions are generated. This process is applied to a first-order and second-order audio *continuous-time sigma-delta* (CTΣΔ) converter. Emulation shows near identical performance to Simulink models. The existing tool is also extended to allow sweeping analog model parameters during emulation, opening the use of emulation for analog design-space exploration. A standalone FPGA demo is built on a Xilinx XC7Z020 platform, with an emulated CTΣΔ *analog to digital converter* (ADC) with the ability to change model parameters, in a real-time audio application. The design takes less than 6% of the available FPGA resources. Analog emulation is shown to be a feasible technique for this use-case, given that longer emulation times can be achieved in the future.

Contents

List of Figures

List of Tables

List of Listings

List of Abbreviations

ΣΔ	Sigma Delta
ADC	Analog To Digital Converter
AMS	Analog Mixed-signal
BRAM	Block RAM
CIC	Cascaded Integrator-comb
CTΣΔ	Continuous-time Sigma-delta
DAC	Digital To Analog Converter
DE	Differential Equation
DSP	Digital Signal Processing
DUT	Design Under Test
FF	Flip-flop
FIR	Finite Impulse Response
FPGA	Field-programmable Gate Array
HDL	Hardware Description Language
HIL	Hardware-in-the-loop
I2S	Inter-IC Sound
ILA	Xilinx Integrated Logic Analyzer
LUT	Look-up Table
NTF	Noise Transfer Function
OSR	Oversampling Ratio
PCB	Printed Circuit Board
PL	Programmable Logic
PS	Processing System
PWL	Piece-wise Linear
RTL	Register Transfer Level
SN_QR	Signal-to-quantization-noise Ratio
SNR	Signal-to-noise Ratio
SoC	System-on-chip
STF	Signal Transfer Function
TF	Transfer Function
VIO	Xilinx Virtual-IO
WDF	Wave Digital Filter

Chapter 1

Introduction

A crucial step in the design of any system is verification. This is especially true in the design of *systems-on-chip* (SoCs), where the manufacturing of the final product is time-consuming and expensive. Knowing that each part of the SoC will function separately and together is imperative. Verification is not an easy task in modern SoCs which comprise complex *analog mixed-signal* (AMS) designs where analog blocks are combined with digital control, calibration and filtering. On top of this, many digital designs that are part of an SoC run firmware that also requires verification. There are well-accepted verification techniques for separate analog and digital designs, for AMS designs there are several methods, each with their own trade-offs.

1.1 Conventional Verification

To understand the trade-offs in AMS verification, it is important to understand conventional verification of separate analog and digital designs.

For analog designs, the convention is simulation using SPICE [1] or comparable tools. SPICE will refer to all circuit-level simulators for the rest of this text. Designs are modelled with a netlist: a list of components, their properties, and connections to other components. These simulations, which calculate voltages and currents, can be done on a number of levels with varying detail; from a functional schematic level, to device level simulations of a chip layout. This is necessary, as test chips for physical testing are expensive and associated with long lead times. Sometimes it is possible to do physical testing on a scaled version of the analog design using discrete components, though this technique can be lacking in accuracy.

Digital verification takes a different approach. Digital designs are usually modelled, or described, in a *hardware description language* (HDL) such as (System)Verilog or VHDL. A subset of these HDL descriptions, so called *register transfer level* (RTL) descriptions, are synthesizable; they can be translated to hardware. The RTL description is then synthesized into gate-level description which describes the connection of standard logic gates which can be implemented physically on a chip. HDL designs (at the non-synthesisable behavioural, RTL, and gate-level) are conventionally verified using event-driven simulation in which digital signals are represented as ones and zeros. In fact, in the context of HDL descriptions, simulation is always event-driven. Event-driven simulation, however, may become prohibitively slow when it is used to develop firmware that might run on the digital chip, and test

chips, come with the same cost and lead-time issues as their analog counterparts. The solution to slow simulation is prototyping digital designs on a *field-programmable gate array* (FPGA). The logic gates and *look-up tables* (LUTs) on these devices are configured to the desired RTL behaviour. The FPGA allows for verification and firmware development as if it was the final hardware, and is easily updated as the RTL design changes.

1.2 AMS Verification

In the move to AMS verification, the techniques above are essentially combined and adapted. A hybrid approach combines the physical verification of analog designs with digital verification on an FPGA. Analog test chips are interfaced with the digital verification platform, also called *hardware-in-the-loop* (HIL). However, these setups are expensive and impractical. Because of the lead time associated with analog chips it is not possible to effectively verify digital designs alongside the analog design process, since a change in the analog design is not immediately available for digital verification. Therefore, this method is only practical when the analog design has been finalised.

The other conventional solution for AMS verification is analog and digital co-simulation. This combines SPICE simulation with digital event-driven simulation, bringing the digital design into an analog verification environment. This type of simulation requires converting between the representations used by the simulation engines; voltages and currents, and digital signals respectively. The big drawback of this method is the speed of SPICE simulation, especially in comparison to digital-only simulation, which is a bottleneck for efficient digital verification, especially when verifying firmware. Additionally, the level of detail of the analog design is often higher than what is required for digital verification or firmware development. Neither does this method allow for digital prototyping on FPGA, which is standard practice.

The final option for AMS verification is to bring the analog model fully into the digital verification environment. This involves describing analog models in behavioural (non-synthesisable) HDL and simulating them using event-driven simulation. Provided that the HDL model is synthesisable, synthesizing and running these AMS designs on an FPGA is called *emulation*. While simulating analog models using behavioural HDL such as Verilog-AMS is a well-accepted technique, most methods stop there and are not synthesizable and therefore not capable of running on an FPGA. This thesis explores the techniques for taking the step towards emulation which requires synthesisable RTL models.

1.3 The Case for Emulation

The term emulation comes from the fact that the analog behaviour running on the FPGA is not truly analog, but an imitation of the desired behaviour.

Like with the other AMS verification techniques presented above, there are trade-offs to emulation. Running analog models in HDL simulation can give a speed-up of 2-3 orders of magnitude, though at the cost of accuracy [2]. Running these designs on an FPGA can give an additional speed-up of 2 orders of magnitude [2]

at the same accuracy. Despite this lower accuracy, emulation has a clear use case in verifying AMS designs with large digital parts that could run firmware, where extremely detailed analog behaviour is not required to verify the digital design that relies on the analog behaviour. Additionally, the digital design is directly verified in hardware.

1.3.1 Beyond Emulation

Even though emulation in this context has been developed for AMS design, there is no reason why the techniques could not be used for analog design alone. While the accuracy afforded by emulation may not match conventional SPICE simulation, there is something to be said for the speed-ups afforded by the use of FPGAs. One possible use case is in design-space exploration. If an analog circuit can be emulated with the speed-ups possible through emulation with the ability to change parameters, it can become a tool for rapid exploration. The limiting factor becomes the detail that can be emulated at on the FPGA, rather than the speed of simulation. This capability adds another tool to the toolbox of an analog or AMS designer.

1.4 Research Goals

The goal of this thesis is to explore the possibilities of emulation for AMS verification and analog design-space exploration. As a vehicle for exploration, an audio-band *continuous-time sigma-delta* (CT$\Sigma\Delta$) converter is used. Since speed-up is one of the clearest advantages of emulation, the speed target for this converter will be real-time. While not necessary for verification, or feasible for all designs (especially higher frequency designs), this target also allows the converter to be run on an FPGA as a practical demo. Thus, the overarching research question for this thesis is formulated as follows:

Can an analog mixed-signal design be emulated in real-time on an FPGA?

To help answer the research question, a number of other questions are formulated. First, an exploration of FPGA emulation is required. It is not feasible to design an entire emulation workflow, so various existing emulation tools will be compared, to find a suitable platform:

1. *What tools are available for emulation on an FPGA?*

The chosen platform or tool is used as a basis for modelling the CT$\Sigma\Delta$ converter.

The first thing to consider in emulation is whether a design is suitable. There may be trade-offs in whether analog or digital design is dominant, or limits to the analog dynamics and detail that can be modelled. Further, while FPGAs are powerful tools for verification, they are limited in their resources. Synthesized RTL designs need to fit on to the available resources, possibly constraining the complexity of AMS designs that should be emulated. The following research question encompasses these ideas:

2. *Which designs are suitable for emulation on an FPGA?*

The answer to this question depends on the chosen tool and the implementation of emulation. Most importantly, the FPGA resource usage of synthesized designs needs to be evaluated.

The next thing to consider is the way that analog behaviour is modelled. Ideally, the same detail and abstraction levels can be used as in conventional analog verification, with as little manual work as possible. This may differ for emulation, raising the question:

3. *What models are appropriate for modelling analog behaviour?*

A review of modelling methods, as well as experience implementing a design in emulation will give insight into this question.

Finally, the aim of this thesis is to suggest a use-case for emulation in analog-only design, as opposed to AMS verification. This comes down to comparing it to the state-of-the-art conventional simulation, raising the final question:

4. *Which advantages does emulation present over conventional simulation?*

Some differences have already been mentioned which suggest that this is a broader comparison than simply speed and accuracy. As a potential answer to this question, emulation is proposed as a tool for design-space exploration where conventional simulation may be less suited. This use-case will be evaluated.

The research questions are summarised below:

Can an analog mixed-signal design be emulated in real-time on an FPGA?

1. *What tools are available for emulation on an FPGA?*
2. *Which designs are suitable for emulation on an FPGA?*
3. *What models are appropriate for modelling analog behaviour?*
4. *Which advantages does emulation present over conventional simulation?*

1.5 Contributions

A large part of this thesis is built upon the set of emulation tools (**svreal**, **msdsl** and **anasymod**) developed at Stanford by Herbst et al. [3, 4]. However, several extensions are made to the tool. Because the documentation for the tools is lacking in some areas, especially in the details of the **anasymod** control infrastructure, a detailed overview of these functions is presented in this work. Further, the existing tool **msdsl** is extended to enable creating models that support sweeping parameters using an analog input. This enables design-space exploration in purely analog models, without the need for manual digital design. To aid in testing and debugging HDL models created with the tool, an extension has been added to the tool **anasymod**, enabling direct post-synthesis simulation.

All of these contributions are shown together in a standalone demo on an FPGA, showing emulation being used in a setting usually reserved for discrete analog electronics or custom HDL design.

1.6 Outline

The report begins with an exploration of the state of emulation in Chapter 2. A comprehensive overview of the capabilities of the emulation tools used in the rest of this thesis is given in Chapter 3. Chapter 4 details the extensions made to the

tool in order to allow for parameter sweeping in analog models. Next, the tools are applied to a design in Chapter 5; the CTΣΔ is introduced, and the various AMS models are described. Chapter 6 presents the verification of the models from the previous chapter, comparing them to reference models. The CTΣΔ models are developed into a standalone FPGA demo with parameter sweeping, which is detailed in Chapter 7. Finally, Chapter 8 concludes this report and presents a discussion of the findings. Recommendations for further work are also given in this chapter.

Chapter 2

Emulation Methods

The goal of this chapter is to provide an overview of the state of AMS emulation on FPGAs, with focus on the tools used to facilitate modelling and emulation. A similar overview has been published as recently as 2021 by Stanley et al. [2], which takes a broad look at AMS verification though both simulation and emulation.

This overview includes simulation, as many emulation techniques stem from event-driven simulation using behavioural HDL descriptions. It can only be called *emulation* if the HDL is synthesizable RTL, and thus capable of being implemented on an FPGA. The availability of the tools, or the source code of the tools is also considered.

The methods for AMS emulation can be roughly categorised into fixed-timestep approaches in Section 2.1 and variable-timestep approaches in Section 2.2. Each approach will be outlined, and a number of implementations of the methods will be presented. Section 2.3 provides an overview of the abstraction levels used in emulation. Finally, Section 2.4 summarises the findings with a justification for the choice of tool that will be used for further exploration.

2.1 Oversampled Methods

The simplest technique for emulating analog models on an FPGA is to discretise the dynamics of the system using a fixed timestep – using, for example, the Euler method [5] – resulting in difference equations with fixed coefficients. As long as the timestep fulfils the Nyquist criterion for the bandwidth of the analog dynamics, the dynamics can be modelled. As a result, the timestep required to represent analog dynamics is often much smaller than the clock periods used in digital control. Therefore, this approach is often called *oversampled*; the analog emulation clock is much faster than the digital clocks. These models however, can still suffer from discretisation errors such as numerical errors or instability. Weakly non-linear systems can be modelled by using the internal states to select an operating region (case). Each operating region has its own set of coefficients, which are stored in memory. Switched systems with low bandwidth, such as DCDC converters, are a common target for this technique [6, 7]. However, it has also been applied to analog-only designs, such as simulating MOS transistor circuits [8].

2.1.1 Tools

A common method for creating oversampled models that are synthesizable is by starting from block diagrams in Simulink [6] or Xilinx System Generator [7, 8], which can generate synthesizable VHDL or Verilog and automatically optimise fixed-point implementations. While these tools are available, they are limited to the blocks available in the given tool and require manual derivation of the block diagram. Aside from this, they do not implicitly support switched systems, thus requiring explicit elaboration of every case.

The tool **msdsl** described by Herbst et al. [3, 4] takes a different approach. It solves the discrete state-update equation from a state-space description which can be defined in several ways (transfer function, netlist, differential equation, direct state-update equation, etc.) using Python. The state-update equation is then used to generate an HDL description in SystemVerilog. Switched cases are supported by adding logic to select appropriate constants in the state-update equation, and are automatically elaborated. This tool is open-source software and available on GitHub [9].

Wu et al. [10] propose using *wave digital filter* (WDF) to describe the dynamics of a system. Each WDF module is implemented as a difference equation in synthesizable Verilog. They generate a network of WDF modules from a netlist, which can then be realised using the library of synthesizable WDF modules for emulation. They claim "virtually perfect" correspondence between emulation and conventional SPICE simulation. Unfortunately, none of the source code for this method is available.

A similar approach is taken by Tertel and Hedrich [11], using blocks for low-and high-pass filters, summing and non-inverting op-amp configurations, half-wave and full-wave rectifiers and sample and hold modules. These blocks are strung together to create full systems. Again, the source is not available for this emulation technique.

2.2 Variable-Timestep Methods

Unlike the piecewise-constant representation of the fixed-timestep approach, a variable-timestep approach needs a time-dependent function to represent the signal between two timesteps. The variable-timestep approach is often partially or fully event-driven, reducing the number of timesteps that need to be evaluated. Events can be changing inputs, switching operating regions, or triggers to re-evaluate signals when the error gets too large.

2.2.1 Time Resolution and Area Trade-off

Before going further into variable-timestep methods, it is interesting to consider what the advantages of it are, especially in the context of FPGA emulation. In digital hardware, synchronous events are driven by a clock, or several clocks of different frequencies. There is always a maximum clock frequency that can be achieved. Either because the hardware cannot produce a faster clock signal, or because long setup and hold times result in timing violations, where the clock "arrives" before a signal is steady.

In the oversampled designs seen so far, the timestep Δt is fixed to a certain emulation clock frequency, which in FPGA emulation is inherently limited by the

available clock frequencies from a clock generator. The maximum time resolution is thus fixed to Δt which limits the dynamics that can be emulated. This can lead to faster clocks being required for the analog emulation than are necessary for the digital systems, which is wasteful of resources.

At the expense of area on the FPGA, more complex calculations can be used to adapt the timestep, to increase and decrease the time resolution as needed, similar to conventional analog simulation tools like SPICE.

2.2.2 Piecewise Linear

The most common method is using *piece-wise linear* (PWL) functions that are stored as a value and a slope at a certain time. This has been proposed in the 1990s by Cottrell [12] and implemented in behavioural HDL description by Pichon et al. [13]. A more recent approach is given by Lim and Horowitz [14], who also add constraints to detect steady states and thus allow feedback loops without oscillations caused by constant re-evaluation. Unless an input triggers an event, the size of the timestep in the PWL model is determined by the maximum timestep that could be taken while staying below a certain error threshold. Neither of these works, however, are synthesizable.

The **msdsl** tool from Herbst [3] allows for a time-varying approach using PWL, though in a less sophisticated way. The response of a system can be stored as PWL (or higher order) values for a number of evenly spaced timesteps. When a timestep is taken, the appropriate PWL value is read from the table.

2.2.3 Other Features

Aside from PWL, other signal representations have been proposed. Jang et al. [15] present a method using summations of exponential functions of the form $c \cdot t^{m-1}e^{-at}u(t)$. This method however, is not synthesizable, rather targeting event-driven simulation using Verilog.

Herbst et al. [16] present a similar method where analog outputs are represented as summations of step responses to digital inputs. While the method presented in this work is synthesizable, it is limited to digitally-driven analog circuits, where the output is only dependent on digital inputs, eliminating analog-only events. In order to synthesize the models for an FPGA, the step-responses of various parts of the system are pre-computed at compile-time and stored in lookup tables as PWL representations.

In later work, Herbst proposes using spline interpolation between points at varying timesteps [3]. This approach projects a number of equally spaced “hidden” spline points over a certain maximum time interval. This is similar to oversampling, except that several points are calculated in parallel in one emulator step. The emulator can then take any timestep smaller than the maximum time interval and the value at that timestep can be interpolated from the “hidden” points. Then a number of “hidden” points will again be projected for the next emulator timestep. Compared to the oversampled approach, this method can be event-driven and take much bigger timesteps, at the expense of more area to calculate “hidden” timesteps in parallel and to interpolate between points.

2.2.4 Tools

Of the mentioned methods, only two have resulted in simulation or emulation tools. The first, XMODEL [17, 18], which incorporates methods from [15], is a commercial tool from Scientific Analog for circuit-level simulation using SystemVerilog. The models used by XMODEL however are not synthesizable, and thus not useful for hardware emulation.

Again, the work by Herbst [3] is the only available tool that can generate synthesizable RTL descriptions for variable-timestep emulation. This is achieved using PWL tables or spline points as mentioned earlier. In fact, a feature of variable-timestep approaches in general is the necessity to keep track of timesteps. Herbst et al. [3, 16] solve this by implementing a time-manager that handles requests for timesteps of varying duration.

2.3 Abstraction Levels

A choice of abstraction level can have a significant impact on the emulation speed depending on the accuracy required. Different tools have focussed on different abstraction levels.

2.3.1 Component Level

The WDF based tool presented by Wu et al. [10] takes a component level approach. Transistors can be modelled using the small-signal equivalent circuit with lookup-table for parameters based on operating-point voltages V_d, V_g, and V_s. Large-signal behaviour using dynamic lookup tables based on the current operating point is mentioned as future improvement. The **msdsl** tool in [3] allows for component level emulation using the netlist interface. In this case transistors can be modelled as switches with parasitics, or as PWL functions as also done in XMODEL [17].

The advantage of the transistor-level emulation is that it does not require additional modelling, and can be implemented (or indeed automatically created) from an existing netlist. This advantage is not to be underestimated, as every manual step is error-prone and requires separate verification.

2.3.2 Circuit Level

The circuit level approach groups components into clusters, as done in XMODEL [17]. Each circuit cluster is modelled by its transfer function. The difficulty with this approach is the appropriate choice of clusters and divisions.

2.3.3 Macromodels

The most common approach, especially in oversampled systems, is using macromodels. These are derived from the dynamic behaviour of the system, and do not necessarily show any resemblance to the analog circuit they are modelling. The advantage of these models is often their increased throughput, because internal nodes can be abstracted away. Bhattacharya et al. [7] split their oversampled model into several higher level systems, each of which is a macromodel of a component in the analog system.

A higher level approach can also use blocks (such as low-and high-pass filters, summing and non-inverting op-amp configurations, half-wave and full-wave rectifiers

and sample-and-hold modules), as done by Tertel and Hedrich in [11]. Each block is instantiated with its own set of parameters. In the same spirit, Lim and Horowitz [19] propose a library of blocks, or templates, with optional inputs for controlling the parameters of the block. This allows for more complex systems, and can directly enable digital control of analog parameters, as is the case in switched systems.

Herbst [3] implements a number of high-level blocks that are modelled using splines. These blocks are characterised in different ways; using poles and zeros, S-parameters, or state-space systems.

Higher levels of abstraction necessarily require more knowledge in the face of building the model, as choices have to be made regarding the effects that have to modelled, and the required accuracy.

2.4 Summary

A number of emulation methods and tools have been presented in this chapter. Of these, surprisingly few support emulation, usually because their HDL descriptions are purely behavioural, and not synthesizable. Most of the tools that support emulation only support oversampling, as opposed to variable-timestep emulation. Therefore, The set of tools described by Herbst in [3] has been chosen as a foundation for further exploration as it presents the only open-source framework that supports emulation, and is the only one that supports variable-timestep emulation.

The tool covers a range of emulation techniques described above; oversampled, and time-varying approaches using PWL or splines. Neither is the tool limited to a certain abstraction level, allowing for component-level models, as well as macromodel descriptions using fixed blocks or mathematical descriptions such as state-space descriptions or transfer functions. Most important, however, is the fact that it is the only tool that consistently supports synthesis of the different functions, a necessary requirement for making use of the acceleration afforded by using FPGAs. The open-source nature of the tool makes it customisable and extendable, allowing great freedom of exploration, and the ability to adapt it to specific needs beyond just AMS verification.

Chapter 3

The Tools: svreal, msdsl and anasymod

Based on the emulation tools presented in the previous chapter, the tool developed at Stanford by Herbst [3] will be used as a platform for modelling AMS designs. The tool consists of three separate packages[1]: **svreal** [20], **msdsl** [9] and **anasymod** [21]. These open-source packages are written in Python and implement SystemVerilog as HDL.

This chapter aims to give an overview of the functions of these packages. Each package does the following:

- **svreal** is a SystemVerilog macro library that allows the same HDL description to be synthesized as fixed-point or floating-point arithmetic.
- **msdsl** is used to generate synthesizable HDL descriptions of dynamic systems in SystemVerilog. Several modelling techniques and abstraction levels are supported, which are written in Python. It makes use of **svreal** to define types in SystemVerilog HDL descriptions.
- **anasymod** is used to control simulation and FPGA emulation of designs created with **msdsl**. It generates control infrastructure (which is synthesizable where needed) around the **msdsl** models for simulation control, and clock and timestep management. It also handles integration of other HDL designs that interface with the **msdsl** models. **anasymod** can call simulators to perform simulation, and directly call Xilinx Vivado to generate bitstreams for supported FPGAs. Finally, it can also be used to interface a PC with FPGAs to control emulation.

Together, the tools offer a complete workflow. However, the documentation for these is not always complete, and there exist several undocumented features. The explanations in this chapter aim to complement the existing documentation available on GitHub and in the work of Herbst et al. [3, 4].

The rest of the chapter will focus on the different modelling options provided by **msdsl**. First there will be a brief look in Section 3.1 at the number formats enabled

[1] The packages may also be called tools individually. It should be clear from context whether an individual tool/package, or the set of all three is meant.

by **svreal**. Then the modelling methods will be described in Section 3.2. The control infrastructure and other **anasymod** functions are presented in Section 3.3. Finally, in Section 3.4 the modelling methods will be compared and discussed, and the generated HDL descriptions and RTL synthesis results will be compared to get an idea of the hardware described by **msdsl**.

3.1 Number Formats using svreal

Before going into the AMS modelling capabilities of **msdsl**, it is useful to know how the analog values are represented during simulation and emulation. Using the macros defined by **svreal**, it is possible to simulate and synthesize models created by **msdsl** for both fixed-point and floating-point data types. Synthesis is done in fixed-point by default, but the Berkeley HardFloat library [22] can be used to synthesize floating-point arithmetic, though this is not treated further in this thesis. Simulation can be performed using the aforementioned types, as well as the non-synthesizable SystemVerilog **real** type. The latter can be used to debug numerical errors and fixed-point effects such as truncation and wrap-around. The SystemVerilog **real** should not be confused with the **svreal** "type", which actually refers to the macros defined by the **svreal** package, and for the purposes of this thesis represents a fixed-point type (though it could also represent a floating-point representation with HardFloat).

The different number formats are interchanged by defining all signal and operations as macros, and changing the definition of these macros as appropriate. For example, a signal `x` declared using the macro `` `DECL_REAL(x) `` could either be declared as a SystemVerilog **real** type – `real x` – or as a fixed-point signal which is discussed below. Note that the choice of number format does affect whether the HDL generated by **msdsl** is synthesizable or not. When the SystemVerilog **real** type is used, the generated HDL is behavioural (non-synthesizable), and can only be simulated with event-driven HDL simulation. The other choices – fixed-point and floating-point using HardFloat – result in synthesizable RTL that is suitable for emulation.

3.1.1 Fixed-Point

Fixed-point is the default mode of **svreal**. A number a is represented as a two's complement signed integer s of a word length w with an implied exponent p. The value of the number a is approximately:

$$a \approx s \cdot 2^p$$

An exponent of 0 is equivalent to a signed integer and a negative exponent can represent numbers < 1. Relating the exponent p to a fixed-point representation with integer and fractional bits, given that p is negative the number of fractional bits is $-p$, while the number of integer bits is $w - 1 + p$.

The word lengths of the fixed point numbers are tailored to the word lengths used in the *digital signal processing* (DSP) cores of FPGAs. This is often two different word lengths; one for constants and one for other signals. Therefore, signals and constants are declared explicitly. For the Xilinx FPGAs used later in this thesis, the shorter (constant) word length is 18 bits, while the longer one is 25 bits.

The choice of exponent presents a trade-off between resolution and range. Increasing one reduces the other. In **svreal**, the exponent is chosen based on the range of the signal, a higher exponent results in a larger range, while a smaller exponent provides more resolution. For a signal of range $\pm R$, the exponent is calculated as follows [3]:

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

The information about range, exponent length and word length have to be passed along with the signal through use of the appropriate macros The operations with fixed-point number in **svreal** are fairly basic. When doing operations with or conversions between signals of different word lengths and exponents, the inputs are aligned to the exponent of the resulting signal, and extended or truncated to the destination word length. Overflows of the fixed-point range are not handled either, there being no option for other behaviour such as saturation.

3.2 Models and Abstractions

In general, analog circuits can be modelled as dynamic systems, and in many cases as (piecewise) linear dynamic systems. The principle of **msdsl** is to solve the state-update equation of a linear dynamic system from its state-space equations. First, a set of equations is compiled, and transformed into the state-space equations with input $\mathbf{x}$, output $\mathbf{y}$ and state-space variables $\mathbf{v}$:

$$\dot{\mathbf{v}}(t) = \mathbf{A}\mathbf{v}(t) + \mathbf{B}\mathbf{x}(t) \tag{3.1}$$

$$\mathbf{y}(t) = \mathbf{C}\mathbf{v}(t) + \mathbf{D}\mathbf{x}(t) \tag{3.2}$$

Equation (3.1) is called the state equation, and Equation (3.2) is called the state output equation. From here, the state-update equation is solved [23]:

$$\mathbf{v}(t + \Delta t) = \tilde{\mathbf{A}}\mathbf{v}(t) + \tilde{\mathbf{B}}\mathbf{x}(t) \tag{3.3}$$

with:

$$\tilde{\mathbf{A}} = e^{\Delta t \mathbf{A}} \text{ and } \tilde{\mathbf{B}} = \mathbf{A}^{-1} \cdot \left(\tilde{\mathbf{A}} - \mathbf{I} \right) \cdot \mathbf{B}$$

The equations (3.3) and (3.2) are then implemented in SystemVerilog to calculate the output at each timestep. The coefficients $\tilde{\mathbf{A}}$, $\tilde{\mathbf{B}}$, $\mathbf{C}$ and $\mathbf{D}$ are computed at compile-time. Thus, in general, the timestep Δt must be known at compile-time as well, resulting in fixed-timestep emulation.

Each modelling method and abstraction performs similar steps in generating the HDL description, depending on the starting representation. The different modelling methods can be combined in one model to varying degrees. Using the example of an *RC* network shown in Figure 3.1, the different modelling methods will be presented. While this example uses input and output voltages, the inputs, outputs and states of a model could be voltages or currents.

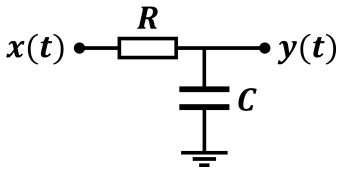


Figure 3.1: Simple RC network

3.2.1 State-Update Equation

The first modelling method is the most straightforward. For multiple inputs $\mathbf{x}$ and states $\mathbf{v}$ the single output y is directly defined by a state-update equation. Modelling this way requires manually deriving the state-update equation, which for the RC low-pass filter network is

$$y(t + \Delta t) = ay(t) + (1 - a)\, x(t) \quad \text{with} \quad a = e^{-\Delta t/RC} \tag{3.4}$$

In this case, **msdsl** skips all steps discussed above, and generates an HDL description from the given state-update equation. The full Python code to create such a system is shown in Listing 3.1. The majority of this code will be identical for every model, as it also handles passing arguments from **anasymod** (lines 19-23) and compiling the model to SystemVerilog (lines 26-33). The highlighted lines of interest are lines 7-12; inputs and outputs to the system (which could be digital of analog) are defined, and an equation for the next cycle is given.

The generated HDL description is shown in Listing 3.2. The input and output signals and ports are defined explicitly in lines 9-10 and 12-13 respectively. It then clearly shows the steps of the state-update equation; input and output are multiplied by constants (a and $(1 - a)$), then added and clocked to the output. Each of these operations uses an **svreal** macro, and can thus be synthesized to fixed- or floating-point by changing the definition of the macro.

The first two arguments of each macro are the LHS and RHS inputs, and the final argument is the resulting signal which will be created or written to. Thus, `` `MUL_CONST_REAL `` on line 16 multiplies the signal `v_out` with a constant, and assigns the result to the signal `tmp0`. On line 18, the signals `tmp_0` and `tmp_1` are added and assigned to `tmp_2` The final macro `` `DFF_INTO_REAL `` implements a D-*flip-flop* (FF) to clock the data to the output. The reset signal is given by the **msdsl** macro `` `RST_MSDSL `` like the clock which is given by `` `CLK_MSDSL ``, and the enable signal is always high. Note that this macro uses `_INTO_` to assign its result to an existing signal, as opposed to defining the output signal. That means that the signals that are not explicitly defined (such as `tmp_0`), are defined within the macros that require them.

3.2.2 Differential Equation

The next method uses symbolic systems of linear *differential equations* (DEs). For example, the RC filter can be modelled by the differential equation:

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

The relevant code is shown in Listing 3.3. Comparing to the relevant lines from Listing 3.1, the DE method is negligibly shorter, but does save the manual calculation of the state-update equation.

Listing 3.1: Modelling using a state-update equation in **msdsl**

Python

```python
from pathlib import Path
from argparse import ArgumentParser
from msdsl import MixedSignalModel, VerilogGenerator
from numpy import exp

def make_model(name, build_dir, dt=1e6, r=1e3, c=1e-9):
    m = MixedSignalModel(name, dt=dt, build_dir=build_dir)
    m.add_analog_input('v_in')
    m.add_analog_output('v_out')
    # apply dynamics
    a = exp(-dt/(r*c))
    m.set_next_cycle(m.v_out, a*m.v_out+ (1-a)*m.v_in)
    return m

def main(name='rc_eq'):
    print('Running model generator...')

    # parse command line arguments
    parser = ArgumentParser()
    parser.add_argument('-o', '--output', type=str,
    ↪  default=name+'/build/models/default/main')
    parser.add_argument('--dt', type=float, default=1e-6)
    args = parser.parse_args()
    build_dir = Path(args.output).resolve()

    r, c = 1e3, 1e-9
    m = make_model(name, build_dir, args.dt, r, c)

    # determine the output filename
    filename = build_dir / f'{m.module_name}.sv'
    print(f'Model will be written to: {filename}')

    # generate the model
    m.compile_to_file(VerilogGenerator(), filename)

if __name__ == '__main__':
    main()
```

The generated HDL description is identical to that in the previous section, as can be seen in Listing C.1 in the Appendix. This is to be expected, since the state-update equation derived from the DE description is the same as that derived manually in the previous section, as shown in the following. Deriving the state-space equations from the DE as **msdsl** would do, gives:

$$\dot{v} = \left[-\frac{1}{RC}\right] v(t) + \left[\frac{1}{RC}\right] x(t)$$

$$y(t) = [1]\, v(t) + [0]\, x(t)$$

Listing 3.2: Generated HDL description from **msdsl** state-update equation

SystemVerilog

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

Listing 3.3: Modelling using a DE **msdsl**

Python

```
m = MixedSignalModel(name, dt=dt, build_dir=build_dir)
m.add_analog_input('v_in')
m.add_analog_output('v_out')
# apply dynamics
m.add_eqn_sys([c*Deriv(m.v_out) == (m.v_in-m.v_out)/r])
```

The equivalence to Equation (3.4) becomes apparent in the coefficients of the state-update and state output equations:

$$\tilde{\mathbf{A}} = e^{-\frac{1}{RC}\Delta t}, \quad \tilde{\mathbf{B}} = 1 - e^{-\frac{1}{RC}\Delta t}, \quad \mathbf{C} = 1, \quad \mathbf{D} = 0$$

While this example is simple, **msdsl** is capable of handling systems of several linear DEs, with multiple inputs, outputs and states as long as the number of equations is equal to the number of unknowns.

3.2.3 Transfer Function

The third way to describe a dynamic system is by *transfer function* (TF). The TF of the RC low-pass filter is:

$$H(s) = \frac{1}{sRC + 1}$$

This can be specified in **msdsl** using the coefficients of the numerator and denominator, as shown in Listing 3.4. The TF is discretized and the state-update equation is derived, though unlike other methods the state-space description (a necessary step) is calculated internally in a SciPy function [24], rather than explicitly by **msdsl**. The advantage of a TF model is that the TF could be calculated from existing SPICE models, for example.

The HDL description can be found in Listing C.2 in the Appendix. While similar to the previous two, this HDL description adds an extra clock period of latency, as the inputs are clocked in as well. The extra delays generated at the input and output of the TF are generated to save an input and output history, something not

Listing 3.4: Modelling a TF in **msdsl**

```python
m = MixedSignalModel(name, dt=dt, build_dir=build_dir)
m.add_analog_input('v_in')
m.add_analog_output('v_out')
# apply dynamics
m.set_tf(m.v_in, m.v_out, [[1], [r*c, 1]])
```

Listing 3.5: Modelling from a netlist in **msdsl**

```python
m = MixedSignalModel(name, dt=dt, build_dir=build_dir)
m.add_analog_input('v_in')
m.add_analog_output('v_out')
# apply dynamics
circ = m.make_circuit()
gnd = circ.make_ground()
circ.voltage('net_x', gnd, m.v_in)
circ.resistor('net_x', 'net_y', r)
circ.capacitor('net_y', gnd, c, voltage_range=RangeOf(m.v_in))
circ.add_eqns(AnalogSignal('net_y') == m.v_out)
```

necessary for the simple *RC* model used here, but required for more complex TFs where delayed samples of input are necessary.

3.2.4 Netlist

Finally, **msdsl** has the capability to derive the state-update equation from a netlist. The code for this is shown in Listing 3.5. Each component is defined between two nodes. For storage elements (capacitors and inductors) the voltage and current range respectively has to be given, though **msdsl** supports automatic calculation of this based on existing, defined voltage or current nodes. Other supported elements are diodes and switches. These will be discussed later. The netlist is broken down into equations, compiled into a state-space description, and the state-update equation is found.

Again, the generated HDL description is almost identical to the other methods. The difference is that the state variable is explicitly present as a signal `tmp_circ_2`, which is defined from the input signal, and assigned to the output. The HDL description can be found in Listing C.3 in the Appendix.

3.2.5 Non-linear models

msdsl has support for non-linear modelling as well. This allows for both implementation of switches, non-linear functions, different types of random noise and the use of variable-timesteps.

3.2.5.A Switches

Switches can be described in a number of ways, depending on the modelling method. First, a digital signal must be defined which will control the switch:

```
m.add_digital_input('sw')
```

Then a constant can be defined with different values based on the state of the switch signal. For example, changing the resistance of the resistor in the RC network from R_1 to R_2 by changing constants in the DE example above:

```
rsw = eqn_case([1/r1, 1/r2], [m.sw])
m.add_eqn_sys([c*Deriv(m.v_out) == (m.v_in-m.v_out)*rsw])
```

Note that `rsw` is defined as the inverse of `r1` and `r2`, since division by an `eqn_case` object is not supported, thus we must multiply by the inverse.

When using a netlist, the `switch` element can be used to the same effect. This element has two values based on the state of the switch; on resistance and off resistance.

```
circ.switch('net_x', 'net_y', ctl=m.sw, r_on, r_off)
```

The diode model available in the netlist mode is similar to the switch, with the difference that it is not activated by a switching signal, but by the current through the diode. The diode will only turn on when the voltage across it (from `net_x` to `net_y`) is above a voltage threshold, and will turn off when the current is below a certain threshold. The diode has a forward voltage drop `vf` that can be defined, along with the on- and off-resistance of the diode.

```
circ.diode('net_x', 'net_y',r_on, r_off, vf)
```

From testing, the methods named above are the only switch implementations that work. That is, the switch case does not work for models described by state-update equation or TF. This is likely because these methods fully derive the state-space equations and state-update equation.

Both methods are implemented in the same manner. In each iteration the value of each state-space coefficient $\tilde{\mathbf{A}}$, $\tilde{\mathbf{B}}$, $\mathbf{C}$ and $\mathbf{D}$ is selected from a table, based on the state of the switch. The full Python code and generated HDL description for the DE with switch can be found in Listing C.4 and Listing C.6 in the Appendix. On the implementation side, the tables of coefficients are stored in LUTRAM as opposed to *block RAM* (BRAM), as the latter would introduce one clock cycle delay in fetching the coefficients, every time they are required.

3.2.5.B Piecewise Approximations

msdsl allows for approximating arbitrary functions as piecewise nth order linear approximations. The coefficients $\{a_1, \ldots a_n\}$ of the function approximation $f(t) = a_1 + a_2 t + a_3 t^2 + \ldots + a_n t^{n-1}$ for a chosen domain of t (such as a number of timesteps Δt) are computed at compile-time and stored in BRAM.

A special use of the function approximation is for variable-timestep approximations. Take Equation (3.4), for a variable-timestep the state-update equation becomes:

$$y(t + \Delta t) = a(\Delta t)y(t) + (1 - a(\Delta t))\, x(t)$$

Listing 3.6: Generating HDL description from a TF using **msdsl**

Python

```python
m = MixedSignalModel(name, dt=dt, build_dir=build_dir)
m.add_analog_input('v_in')
m.add_analog_output('v_out')
m.add_analog_input('dt')
# apply dynamics
func = lambda dt: exp(-dt/(r*c))
f = m.make_function(func, domain=[0, 10*r*c], numel=512, order=1)
a = m.set_from_sync_func('a', f, m.dt)
v_in_prev= m.cycle_delay(m.v_in, 1)
v_out_prev= m.cycle_delay(m.v_out, 1)
m.set_this_cycle(m.v_out, a*v_out_prev+ (1-a)*v_in_prev)
```

Now the timestep-dependent coefficient $a(\Delta t) = e^{-\Delta t/RC}$ can be implemented as a function and approximated. The code in Listing 3.6 would be used to generate a piecewise-linear (order 1) approximation of $a(\Delta t)$. Notable differences to those in Listing 3.3 are highlighted. The resulting HDL description is shown in Listing C.5 in the Appendix. The original two multiplications and one addition are still clearly visible (and highlighted) alongside the extra hardware generated to compute $a(\Delta t) = a_1 + a_2\Delta t$ and $1 - a(\Delta t)$ from the coefficient tables stored in memory.

While variable-timestep emulation will not be further treated, the piecewise function approximation is used for parameter sweeping in Section 4.2.

3.2.5.C *Other modelling methods*

For completeness, the other (non-linear) modelling capabilities of **msdsl** are briefly outlined, though these are not further treated in the scope of this thesis.

One is pseudorandom noise generation using PWL tables. This feature be extremely useful for modelling noise sources in analog circuits.

The final modelling option uses splines to interpolate between timesteps. The spline representation is implemented in some high-level blocks to provide some high-level abstractions in **msdsl**. Currently, this includes saturation non-linearity, a lossy channel specified from S-parameters and a continuous-time linear equalization model which is specified by pole and zero values. Each of these blocks represents its own model, which can be interfaced with other **msdsl** models using **anasymod**.

3.3 anasymod

Before moving on to HDL simulation, synthesis and emulation of the **msdsl** models, some attention has to be given to the simulation and emulation process, and especially the control of the mixed-signal **msdsl** models. HDL models generally require clock and reset signals for their synchronous elements, and variable-timestep **msdsl** models need to know the length of timesteps. Furthermore, simulation and emulation requires some way to pass data to and from the *design under test* (DUT) – the **msdsl** model and other digital designs. Therefore, it is not possible to run **msdsl** models on their own. This is where the tool **anasymod** comes in. It provides two

vital functions; a means of calling simulation tools and running interactive emulation straight from Python, and generating the control infrastructure around the DUT. When an **msdsl** model is run, it is always with the **anasymod** infrastructure wrapped around it.

3.3.1 Simulation and Emulation

anasymod provides integration with Icarus Verilog [25] for SystemVerilog simulation, and Xilinx Vivado [26] for (mixed language) simulation, synthesis and emulation. **anasymod** reads project configurations, written as `yaml` files, calls **msdsl** to generate the SystemVerilog descriptions of any models, and generates all the SystemVerilog descriptions of the control infrastructure. Finally, Icarus Verilog or Vivado are called to run the simulation or emulation. In the case of emulation, which requires programming the FPGA, this step includes configuring Vivado to synthesize the HDL, and checking the resulting RTL description. At this point, post-synthesis simulations can be run; the ability to launch these from **anasymod** has been added in the scope of this thesis[2], but is not further treated except for the implementation details given in Appendix A. Finally, a bitstream is generated that can be used to program the FPGA.

Emulation, and emulation control are also handled by **anasymod**. There are two methods for using a computer to control an emulation running on an FPGA. One is a custom firmware-in-the-middle approach that requires an SoC incorporating a *processing system* (PS) like a CPU alongside the *programmable logic* (PL) with comprises the FPGA. Firmware on the PS can directly interface with the emulation running in PL.

The other slower, but more general, method uses *Xilinx Virtual-IO* (VIO) [27] and *Xilinx Integrated Logic Analyzer* (ILA) [28] for direct control of the FPGA from a computer. **anasymod** sets up VIO for default control signals, as well as custom signals (specified in a `yaml` file) anywhere in the testbench. Reading signals through VIO is done by recording the signals to BRAM on the FPGA during emulation, stopping the emulation when BRAM is full, and then using VIO to read the signals from BRAM to a computer. This has the major disadvantage that the duration of an emulation run is limited by a combination of the size of signals, and the amount of BRAM available to the VIO. An outline of a solution for working around the problem is given in the recommendations (Section 8.3), but not further treated due to time constraints.

3.3.2 Control Infrastructure

anasymod generates a SystemVerilog control infrastructure around the DUT containing the **msdsl** model and any other digital design. The various modules of the control infrastructure are shown in Figure 3.2, and their functions are briefly outlined below. Each of the modules in the infrastructure are generated from templates, based on settings specified in various `yaml` configuration files. While this does not cover all the details of the infrastructure, it should give a sufficient overview for the purposes of this thesis. The functional details of timestep management can be found in Chapters 5.3 of [3], while a more detailed explanation of the modules and

[2]This extended fork of **anasymod** is available on GitHub (https://github.com/nsulzer/anasymod)

signals in the control infrastructure can be found in Appendix B. The latter aims to complement former on an implementation level, where documentation is lacking.

`clk_gen` is a wrapper for Xilinx IP that generates a clock with double the emulation clock frequency, and any other required clocks, from a master clock.

`time_manager` calculates the minimum of all timestep requests, and communicates this value back as `emu_dt`. It also keeps track of `emu_time`, the absolute emulation time across varying timesteps.

`osc_model` is required for designs that do not generate any analog clocks. In that case, this module creates the timestep requests (`emu_dt_req`) required for the desired emulation frequency that is eventually tied to `` `CLK_MSDSL ``.

`gen_emu_clks` generates the actual clocks as requested by modules such as the oscillator. These clocks are synchronised to the master emulation clock `emu_clk_2x`. This synchronisation limits the time-resolution of any clock to that of the emulation clock, as the rising edges are aligned.

`sim_ctrl_gen` is a wrapper for the VIO IP required for emulation control from a host PC, as will be explained in the following section. The global reset signal `emu_rst`, accessed with the macro `` `RST_MSDSL `` is also generated here. During simulation, it is a wrapper for the for user-written module `sim_ctrl` for simulation control.

`trace_port` is a wrapper for the ILA IP used for monitoring signals in the design.

`ctrl_anasymod` is responsible for emulation control such as starting, pausing, and stopping emulation by influencing the time manager. It takes a number of signals from `trace_port` to switch between these modes.

`tb` is a user-written module and acts as a testbench for any **msdsl** and custom HDL modules, essentially it is the DUT. All analog signals generated in the rest of the control infrastructure are inputs to this module.

`gen_top` is the top-level entity that instantiates all of these modules.

3.4 Comparison of Models

After presenting the various modelling techniques, a comparison of the different models is given in Table 3.1, as to which features they support and any peculiarities.

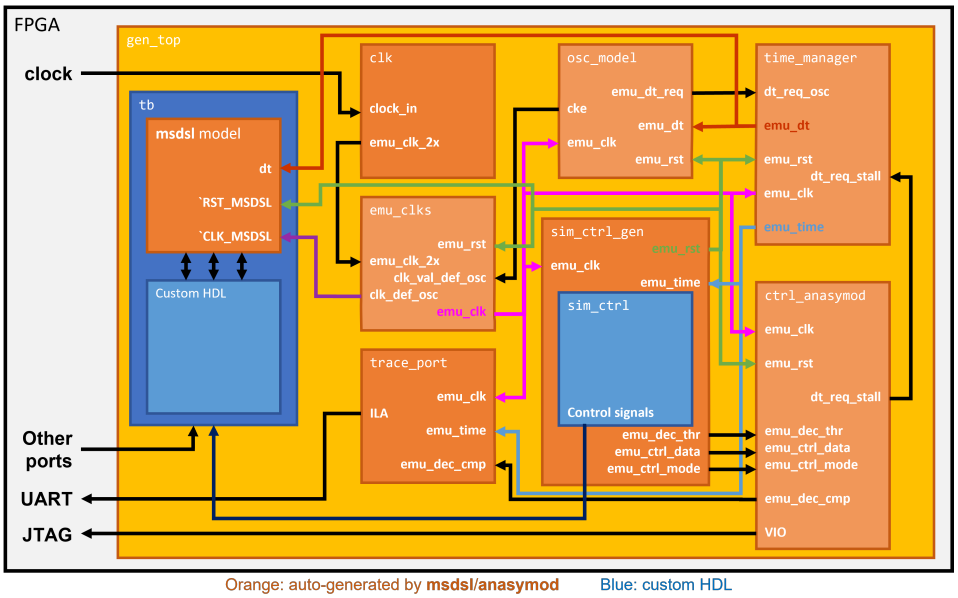


Figure 3.2: **anasymod** control infrastructure

Table 3.1: Summary comparison of different modelling capabilities

Model	Section	Requires Manual Derivation	Supports Function Approximation	Supports Switches	Notes
Update Equation	3.2.1	Yes	Yes	No	
DE	3.2.2	Yes	No	Yes	
TF	3.2.3	Yes	No	No	Additional input and output delays
Netlist	3.2.4	No	No	Yes	

Later, the generated HDL descriptions and the hardware synthesized from these will be discussed[3].

The ability to use each of the modelling techniques together in one model is crucial, since they each support different features. A model using a netlist can, for example, be augmented with TF modelling or PWL tables for (non-linear) components. This is especially necessary for the netlist, as it lacks components such as transistors, which would need to be modelled with, for example, PWL functions.

3.4.1 HDL and Synthesis

A comparison of the generated HDL descriptions is given in Table 3.2. The number of D-FFs is given on the word level, since the choice of number representation with **svreal** could still change the underlying behavioural or RTL description. Most notable is the variable-timestep model, for which extra hardware is generated to

[3] All models and simulation setups are available on GitLab (https://gitlab.utwente.nl/s1788973/anasymod-synthesis)

Table 3.2: Comparison of generated HDL descriptions of *RC* network

Model	Timestep	Python Description Listing	Generated HDL Listing	Multipli-cations	Additions	DFFs
Update Equation	Fixed	3.1	3.2	2	1	1
DE	Fixed	3.3	C.1	2	1	1
TF	Fixed	3.4	C.2	2	1	3
Netlist	Fixed	3.5	C.3	2	1	1
DE w. switch	Fixed	C.6	C.4	2	1	1
Equation	Variable	3.6	C.5	3	4	3

Table 3.3: Comparison of gate-level synthesis results of *RC* network descriptions

Model	Timestep	LUT Logic	LUTRAM	FF	BRAM	DSP
Update Equation	Fixed	25	0	25	0	2
DE	Fixed	25	0	25	0	2
TF	Fixed	25	0	25	0	2
Netlist	Fixed	25	0	25	0	2
DE w. switch	Fixed	26	0	25	0	2
Equation	Variable	67	0	1	0.5	5
anasymod infrastructure		2124	287	3921	13	0

fetch coefficients from BRAM, and the extra D-FFs generated at the input and output of the TF model.

After synthesis however, all fixed-timestep models show the same generated hardware. Choosing a fixed-point type in **svreal** ensures that the HDL generated by **msdsl** is a synthesizable RTL description. This is then synthesized to gate-level. Synthesis results for the *RC* filter models are summarized in Table 3.3, this time on a bit level. The number of instantiated FFs confirms a word length of 25 bits for analog signals. It is also clear that multiplications are mapped to DSP slices, while addition is implemented with LUTs. The additional memory described for the TF model is not explicitly instantiated, but incorporated into registers in the DSP blocks. Adding a switch to the design adds only one extra LUT (an inverter), as the switch signal itself is used to generate the appropriate bit pattern for either of the two coefficients.

The variable-timestep implementation is the only one that is notably different. Here three extra DSPs are implemented; two in calculating the address of the PWL coefficients, and one multiplying the PWL coefficient with the timestep. Additionally and it takes BRAM blocks to store the coefficient tables.

Much larger than the models themselves, is the overhead added by **anasymod**, as seen in the last row of Table 3.3. The 13 BRAM blocks are reserved by the ILA for storing monitored signals.

Chapter 4

Design Space Exploration

So far, attention had been paid to the original use-case for emulation, namely AMS verification. However, this workflow surprisingly may lend itself to analog design as well. The big advantage of emulation is the much higher speed at which models can be run, something that cannot be paralleled with conventional simulation. This property can be exploited for design-space exploration. While not matching the component-level accuracy of simulation, it may be advantageous to emulate models with the ability to tune parameters to find, for example, stable operating regions of complex designs.

The goal then is to implement the ability to tune model parameters during emulation.

4.1 Coefficient Calculation

The challenge for tuning parameters is that changing one parameter in the state-space equation has an effect on multiple coefficients of the state-update equation, requiring a change in several model coefficients.

Consider the example of the RC network from the previous chapter (Figure 3.1). The state space equations are as follows:

$$\dot{v} = \left[-\frac{1}{RC}\right] v(t) + \left[\frac{1}{RC}\right] x(t)$$

$$y(t) = [1]\, v(t) + [0]\, x(t)$$

with the solution:

$$\tilde{\mathbf{A}} = e^{-\frac{1}{RC}\Delta t}, \quad \tilde{\mathbf{B}} = 1 - e^{-\frac{1}{RC}\Delta t}, \quad \mathbf{C} = 1, \quad \mathbf{D} = 0$$

Sweeping the *paramter* R, for example, requires sweeping both the *coefficients* $\tilde{\mathbf{A}}$ and $\tilde{\mathbf{B}}$. All coefficients affected by one parameter are called the *coefficient set* – in this case $\left\{\tilde{\mathbf{A}}, \tilde{\mathbf{B}}\right\}$. In a more complex state-space equation, all coefficients could potentially be affected. Further, each coefficient is potentially a matrix of coefficient, if the number of inputs, outputs or states is larger than one.

There are two methods to deal with these changing coefficients: compile-time or runtime calculation.

4.1.1 Runtime calculation

Computing the coefficients at run-time gives the greatest flexibility. Any parameter could be arbitrarily tuned. However, the calculation of coefficients at runtime is not feasible for an arbitrarily sized design. As discussed in Section 3.2, calculating the update equation requires matrix inversion and exponentiation, since:

$$\tilde{\mathbf{A}} = e^{\Delta t \mathbf{A}} \text{ and } \tilde{\mathbf{B}} = \mathbf{A}^{-1} \cdot \left(\tilde{\mathbf{A}} - \mathbf{I}\right) \cdot \mathbf{B}$$

Performing these calculations on an FPGA is area expensive – especially considering the number of coefficients that may be affected by a single parameter – taking resources away from models themselves.

4.1.2 Pre-computation

To eliminate the issue of expensive computations, the coefficients can be pre-computed at compile-time and stored in memory. Of course, this limits the range and resolution of the sweep to a range and number of steps that is defined at compile-time. Each coefficient in a coefficient set is calculated for every possible value of the parameter. The resulting precalculated coefficients are called a *coefficient array*; a table of coefficients for different parameter values. In the *RC* network example, sweeping the parameter R over 512 value, results in the coefficient arrays

$$\left\{\tilde{\mathbf{A}}_0, \ \ldots \ , \tilde{\mathbf{A}}_{511}\right\} \quad \text{and} \quad \left\{\tilde{\mathbf{B}}_0, \ \ldots \ , \tilde{\mathbf{B}}_{511}\right\}$$

The larger obstacle to this method however, is the storage required for all coefficient arrays. Arrays can be stored in LUTRAM (LUTs used as memory), or in BRAM. The former impacts the size of designs, as LUTs are required to implement designs in PL, while the latter requires an extra clock cycle to fetch the coefficients from BRAM. Take, for example, the ZedBoard which has 3.4Mb of LUTRAM. If half of the LUTRAM is used for storing coefficients – leaving the other half for design – that corresponds to about 70800 24-bit coefficients. Assuming that sweeping 512 steps would be sufficient, that is 138 coefficient arrays of 512 coefficients each. Then, the number of parameters that can be swept depends on the size of the coefficient set of each parameter.

Since **msdsl** already has built in support for creating lookup tables in BRAM, that method will be used to add parameter sweeping functionality to the tool.

4.2 Lookup Tables in msdsl

msdsl has native support for creating lookup tables. These are originally used for PWL approximation for custom functions or variable-timestep emulation, as introduced in Section 3.2.5.B. For parameter sweeping, these tables are used to store a range of coefficients. Figure 4.1 shows the hardware generated my **msdsl** for retrieving a coefficient from BRAM based on a control input `c`. The control input is converted to an address in BRAM, by mapping the full range of control inputs on the full range of 2^n coefficients, and the coefficient at that address is returned as data. Aside from BRAM, the structure requires one DSP for multiplication, and a number of LUTs to implement the addition.

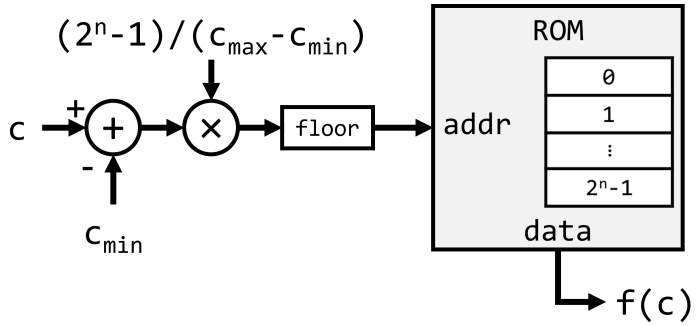


Figure 4.1: Coefficient lookup implemented by **msdsl** (adapted from [3])

Listing 4.1: Use of `make_coef_sweep`

Python

```
m = MixedSignalModel('model')                    # make a model
control = m.add_analog_input('control')          # control input
param = m.make_coef_sweep('param', control, 'lin', [0, 1], 512)
m.add_analog_output('out')                       # some function using param
m.set_this_cycle(m.out, param)                   # some function using param
```

4.3 Implementation

Using the existing **msdsl** function for table creation, **msdsl** is extended[1] with the function `make_coef_sweep`. So far, the function is only a limited implementation of what is described above. Only parameters can be swept that affect a single coefficient, and it has only been tested with update equation modelling in **msdsl**.

Use of the function is shown in Listing 4.1. An analog signal `param` is created, which is swept using the control signal `control`, which can be defined explicitly as above or implicitly by passing a string. The sweep then consists of 512 linearly spaced elements in the range $[0, 1]$. Aside from linear sweeps, the function currently supports logarithmic sweeps, and can be easily extended. Since the **msdsl** tables require the domain of a signal, the function automatically calculates the required domain from the given range. The function then builds the lookup table using `make_function` and applies it to the parameter using `set_from_sync_func`. The full function can be found in Listing C.9 in the Appendix.

[1] This extended fork of **msdsl** is available on GitHub (https://github.com/nsulzer/msdsl)

Chapter 5

Sigma-Delta Models

While the *RC* circuit presented in Chapter 3 is useful for verifying the emulation capabilities of a tool, it does not present a mixed-signal model. For this work, the *sigma delta* (ΣΔ) *analog to digital converter* (ADC) was chosen as a vehicle to further explore AMS emulation. The converter, explained in more detail in Section 5.1 consists of a *continuous-time sigma-delta* (CTΣΔ) modulator which generates a bitstream, followed by digital decimation to convert the bitstream to a digital signal. CTΣΔs present an interesting case for two reasons. The complexity can easily be increased by increasing the order of the modulator, the detail of the integrators, or the number of bits used in the converter. Second, in practical development of the digital decimation filtering that is required after the modulator, a model of the analog modulator is required for testing. Thus emulating the mixed-signal ΣΔ modulator is a clear use-case where emulation can aid in the design of digital blocks.

This chapter begins with an introduction to ΣΔ modulators in Section 5.1, followed descriptions of the ΣΔ designs and the models made with **msdsl** in Section 5.2.

5.1 Sigma Delta ADC

As the case study for emulation, ΣΔ converters, which operate on the principle of oversampling and noise shaping, warrant some explanation. The converter owes its name to the ΣΔ modulator, which constitutes one part of the converter, the other part being digital decimation. Decimation is required to downsample the high frequency bitstream produced by the modulator into a usable digital signal.

5.1.1 Modulator

The modulator is essentially a feedback loop as shown in Figure 5.1. Before each part of the loop is explained, it is important to have an overview of oversampling and noise shaping.

Non-oversampled (Nyquist) ADCs, have a bandwidth f_b, and a sampling rate $f_{s,Nq} = 2 * f_b$ whereby the total quantisation noise power is spread over the bandwidth f_b. Oversampling in ADCs reduces the power density of the quantisation error, by spreading the total quantisation noise $N_Q(f)$ over a larger frequency range by increasing the sample rate. The *oversampling ratio* (OSR) is then defined as $OSR = \frac{f_s}{2f_b}$.

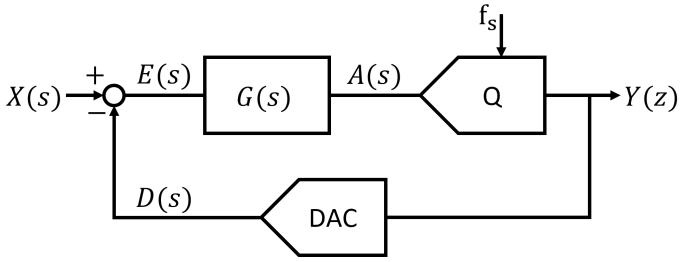


Figure 5.1: Block diagram of CTΣΔ modulator

Due to the oversampled nature of the modulator, it can employ noise shaping to filter quantisation noise of a simple quantizer to high frequencies, outside the passband f_b. The modulator comprises a loop consisting of a filter $G(s)$, quantizer Q, and *digital to analog converter* (DAC). The digital output of the quantizer is fed back to the input, and subtracted from it.

Noise shaping is achieved by appropriate choice of the loop filter function $G(s)$, which results in the *noise transfer function* (NTF) and *signal transfer function* (STF) that represent how the signal and noise are shaped. They are defined as follows, where c is the gain of the quantizer:

$$NTF(s) = \frac{1}{1 + cG(s)}$$

$$STF(s) = \frac{cG(s)}{1 + cG(s)}$$

Essentially, the signal is passed in the passband f_b, while the noise is attenuated. A transfer function $G(s) = \frac{1}{s}$ – an integrator – achieves first-order noise shaping. This is also called a first-order modulator. The NTFs of the first and second-order modulator can be seen in Figure 5.2. Higher orders $\frac{1}{s^n}$ are also possible, and each increase the *signal-to-quantization-noise ratio* ($\mathrm{SN_QR}$) in the passband by shaping more noise to higher frequencies. The theoretical $\mathrm{SN_QR}$ of a modulator with order n is given in [29] as

$$SN_{\mathrm{Q}}R = 20\log_{10}\left(\frac{\mathrm{OSR}^{n+0.5}\sqrt{2n+1}}{\pi^n}\right) \tag{5.1}$$

5.1.2 Decimation

The other vital part of a ΣΔ converter is the decimation stage that downsamples the bitstream and filters out the out-of-band noise, as shown in Figure 5.2. The result of this process is a multi-bit signal at a lower sample rate. An ideal decimation filter removes everything outside the passband without introducing ripple or aliasing. Unfortunately such an ideal filter is not realizable. Instead, *cascaded integrator-comb* (CIC) filters [30] are commonly used structures that both filter and downsample. These structures filter by means of a number of integrator stages, followed by decimation and then comb stages for further filtering at a lower sample rate. Because these structures can introduce droop in the passband, *finite impulse response* (FIR) compensation filters are often used to correct these effects, but only at the end; at the lowest sample rate.

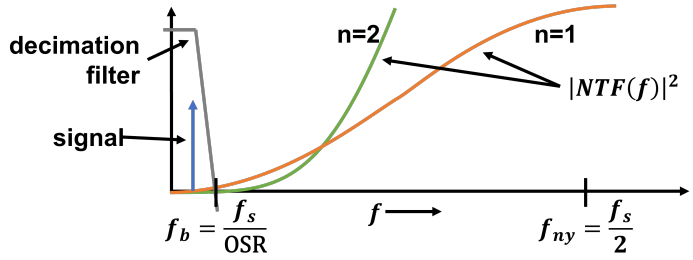


Figure 5.2: Decimation filters out the noise which has been shaped out of the pass-band

The details – topology and implementation – of the decimator used in this design are discussed in more detail in the scope of the implementation of the demo of the full converter, in Section 7.1. For now, the focus will be on the AMS models of the modulator.

5.2 Modulator Models

Since the complete design of a CTΣΔ falls outside the scope of this thesis, an existing second-order ΣΔ design was used [31]. This design has already been calculated, and has an existing HDL description of the decimator. The specifications of the converter are taken from this earlier work designed to achieve an SN_QR of 80dB for audio-band signals (up to 20kHz). The audio signals are sampled at 48.8kHz, with an OSR of 128 leading to a ΣΔ sampling frequency f_s of 6.25MHz. The advantage of these sample rates, is that they are low enough to make real-time emulation possible, with the bandwidth of the analog (audio) signals being far below the achievable clock frequencies in common FPGAs. The emulation clock frequency used in these models is 100MHz; a timestep of 10ns.

In addition to the second-order model, a first-order model was implemented as a more basic test of the emulation techniques. The first-order modulator is always stable, and can act as a control, which may help in debugging. The specifications of the first-order model were kept similar to the second-order model where possible, especially in order to allow re-use of the decimation structure. That means that the number of bits, and OSR and resulting sample rates in the modulator and decimation are kept the same. As a reference for the **msdsl** models, Simulink models are used. These models were used in previous work [31] to generate a bitstream that will be used to test the decimation in HDL; a function that should now fall to the **msdsl** model instead. These two models, and their **msdsl** implementations will be described in more detail in the following.

5.2.1 First-Order Model

The first-order CTΣΔ model represents the most basic ΣΔ modulator. It has an order of one, and 1-bit bitstream output. Therefore, the loop filter has the TF $G(s) = \frac{1}{s}$, the quantizer is a simple comparator which detects whether the signal is greater than or smaller than 0, and the DAC converts the bitstream to ± 1. Only the input gain has been adjusted to keep within the dynamic range of the modulator. Its reference Simulink block diagram is shown in Figure 5.3.

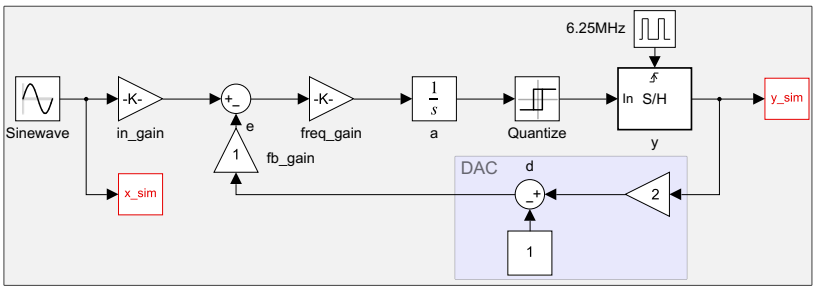


Figure 5.3: Simulink model of fist-order modulator

Listing 5.1: **msdsl** description of first-order ΣΔ model

Python

```
6  m = MixedSignalModel('sd_model', dt=args.dt)
7
8  in_gain, fb_gain, freq_gain = 0.45, 1, 6.25e6
9  G = [[freq_gain],[1, 0]]                        # TF of loop filter
10
11 fs = m.add_digital_input('fs')                  # 6.25MHz sample clock
12 x = m.add_analog_input('x')                     # analog input
13 y = m.add_digital_output('y')                   # digital output
14 e = m.add_analog_state('e', 4, init=1)         # node after summation
15 a = m.add_analog_state('a', 4, init=0)         # node after integrator
16 d = m.add_analog_state('d', 1, init=-1)        # node after DAC
17
18 m.set_this_cycle(e, x*in_gain-d*fb_gain)      # adder
19 m.set_tf(e, a, G)                             # loop filter
20 m.set_next_cycle(y, a>0, ce=fs)               # quantiser
21 m.set_next_cycle(d, y*2-1.0)                  # DAC
```

While it would be possible to describe the modulator using a state-space description, this is not done. Taking advantage of the many modelling methods in **msdsl**, the model is described in the manner of the block diagram Figure 5.1. This allows for easy understanding of the model, and direct comparison to, for example, Simulink models. The final HDL description will be largely the same, regardless of the modelling technique.

The **msdsl** description of the first-order model is given in Listing 5.1, the full Python script can be found in Listing C.7 in the Appendix. All parameters of the model are defined in lines 6-8. The TF of the loop filter is defined in line 9. Lines 11-13 define the inputs and outputs to the modulator. The generation of these input signals will be described later. Due to the choice of model description, the analog states internal to the modulator have to be explicitly defined, as done in lines 14-16. Note the second argument which is the range of the state. This influences the exponent used in the **svreal** type. A range that is too small may lead to saturation in a real analog circuit, but in this digital environment it leads to wrap-around – a very different effect. This is a limitation of **svreal** which does not support saturation in fixed-point operations. With all parameters, inputs, outputs and states defined, the model can be described. The input summation node is described in line 18 with

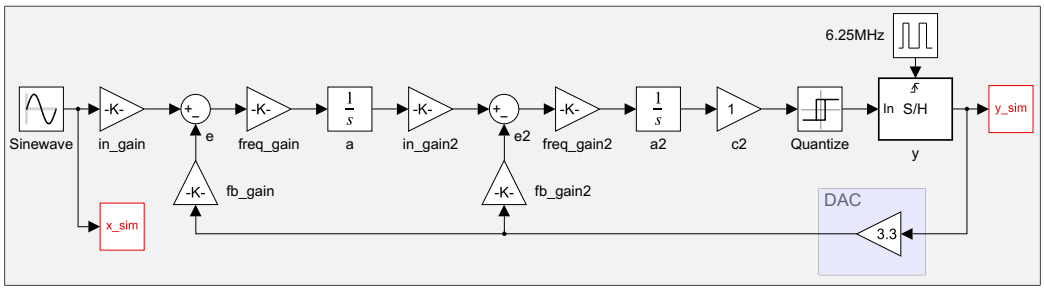


Figure 5.4: Simulink model of second-order modulator

an equation. The loop filter is set from a TF in line 19. The comparator is described in line 20, as an equation whose output is clocked with the sample clock f_s. In HDL this implements a D-FF with a clock-enable signal given by f_s. Finally, the DAC is implemented with the expression $d = 2y - 1$, converting the bitstream with values $\{0, 1\}$ to an analog signal with values $\{-1, 1\}$.

5.2.2 Second-Order Model

The second-order model is a more practical design. This model was designed in previous work and implemented on a *Printed circuit board* (PCB) using discrete components [31] and is adapted for this work. The decimation was done on an FPGA with input voltages of 3.3V. Therefore, as opposed to the first-order model, this model has a range of 0-3.3V, biased at 1.65V. The coefficients, calculated using the MATLAB [32] Delta Sigma Toolbox [33], are scaled to handle a 2.1V input without saturating the 0-3.3V range. This also means that the full-scale input is scaled to fit the range. Again, the reference Simulink model is shown in Figure 5.4.

Similarly to the first-order modulator, the second-order modulator is described in a readable way, with each integrator described by its own TF. Listing 5.2 gives the model of the second-order modulator, while the full script can be found in Listing C.8 in the Appendix.

5.2.2.A Parameter Sweeping

Parameter sweeping, described in Chapter 4, is implemented on the second-order model for the parameters `in_gain2`, `fb_gain` and `fb_gain2`. The **msdsl** model for this is given in Listing 5.3. Each parameter is swept linearly in the range $[0, 1]$. The names of the input signals are a result of the demo implementation which is discussed later.

Listing 5.2: **msdsl** description of second-order ΣΔ model

Python

```
m = MixedSignalModel('sd_model', dt=args.dt)

v_of = 3.3/2                                  # bias voltage
in_gain  , fb_gain  , freq_gain  = 0.2826 , 0.2824 , 6.25e6
in_gain2 , fb_gain2 , freq_gain2 = 0.6520 , 0.5677 , 6.25e6
G  = [[freq_gain ],[1, 0]]                    # TF of integrator 1
G2 = [[freq_gain2],[1, 0]]                    # TF of integrator 2

fs = m.add_digital_input('fs')                # 6.25MHz sample clock
x  = m.add_analog_input('x')                  # analog input
y  = m.add_digital_output('y')                # digital output
xs = m.add_analog_state('xs', 4, init=0)      # scaled input
e  = m.add_analog_state('e' , 4, init=1)      # node after summation 1
a  = m.add_analog_state('a' , 4, init=0)      # node after integrator 1
e2 = m.add_analog_state('e2', 4, init=1)      # node after summation 2
a2 = m.add_analog_state('a2', 4, init=0)      # node after integrator 2
d  = m.add_analog_state('d' , 4, init=-1)     # node after DAC

m.set_this_cycle(xs, 0.65*v_of*x + v_of)      # scale input
m.set_this_cycle(e, xs*in_gain-d*fb_gain)     # summation node 1
m.set_tf(e, a, G)                             # integrator 1
m.set_this_cycle(e2 , a*in_gain2-d*fb_gain2) # summation node 2
m.set_tf(e2, a2, G2)                          # integrator 2
m.set_next_cycle(y, a2>v_of, ce=fs)           # quantiser
m.set_next_cycle(d, (y*3.3))                  # DAC
```

Listing 5.3: **msdsl** description of second-order ΣΔ model with parameter sweeping

Python

```
m = MixedSignalModel('sd_model', dt=args.dt)

v_of = 3.3/2                                  # bias voltage
in_gain    = 0.2826
freq_gain  = 6.25e6
freq_gain2 = 6.25e6
G  = [[freq_gain ],[1, 0]]                    # TF of integrator 1
G2 = [[freq_gain2],[1, 0]]                    # TF of integrator 2

fs = m.add_digital_input('fs')                # 6.25MHz sample clock
x  = m.add_analog_input('x')                  # analog input
y  = m.add_digital_output('y')                # digital output
xs = m.add_analog_state('xs', 4, init=1)      # scaled input
e  = m.add_analog_state('e' , 4, init=1)      # node after input summation
a  = m.add_analog_state('a' , 4, init=0)      # node after integrator 1
e2 = m.add_analog_state('e2', 4, init=1)      # node after second
                                               summation
a2 = m.add_analog_state('a2', 4, init=0)      # node after integrator 2
d  = m.add_analog_state('d' , 4, init=-1)     # node after DAC

in_gain2 = m.make_coef_sweep('in_gain2','n',    'lin',[0, 1],512,order=0)
fb_gain  = m.make_coef_sweep('fb_gain' ,'vaux0','lin',[0, 2],512,order=0)
fb_gain2 = m.make_coef_sweep('fb_gain2','vaux8','lin',[0, 2],512,order=0)

m.set_this_cycle(xs, 0.65*v_of*x + v_of)      # scale input
m.set_this_cycle(e, xs*in_gain-d*fb_gain)     # summation node 1
m.set_tf(e, a, G)                             # integrator 1
m.set_this_cycle(e2 , a*in_gain2-d*fb_gain2) # summation node 2
m.set_tf(e2, a2, G2)                          # integrator 2
m.set_next_cycle(y, a2>v_of, ce=fs)           # quantiser
m.set_next_cycle(d, (y*3.3))                  # DAC
```

Chapter 6

Model and Emulation Verification

After introducing the mixed-signal CTΣΔ models in the previous chapter, they are validated in this chapter. The aim is to confirm the correctness of the **msdsl** modulator models against the ideal Simulink models. The main tool for comparison is *signal-to-noise ratio* (SNR). Additionally, the models are compared to the shape of the ideal NTF to confirm that noise shaping takes place as intended. The **msdsl** models are both verified in a number of different ways:

1. HDL simulation of a behavioural HDL model using SystemVerilog **real** type. Call this run *Sim. Real*.

2. HDL simulation of a synthesizable RTL model using the fixed-point **svreal** type. Call this run *Simulation*.

3. Emulation of the synthesized RTL model on an FPGA. Call this run *Emulation*

The former acts as a direct comparison to the floating-point Simulink models. The next acts as a general verification of the generated RTL description, while the latter ensures that the design is functional even after synthesis.

Before running any simulation or emulation, the synthesis results for each design are presented. This is necessary to verify that the design fits onto the available FPGA resources, and identify any potential timing related issues. The amount of free FPGA resources also give an indication for the limits to larger potential designs.

First, the verification setups are detailed in Section 6.1, followed by synthesis results in Section 6.2 and finally the results of the various model runs listed above are presented in Sections 6.3 and 6.4 for the first- and second-order models respectively. Section 6.5 concludes this chapter with a discussion of the results.

6.1 Verification Setup

Each modulator model is run for 20 periods of a 1kHz sine wave, or 20ms (only 10 periods, 10ms, are used for emulation). In order to remove modulator startup effects, the first period is discarded. The resulting signal is windowed with a Hanning window and the power spectrum is calculated. Unless specified otherwise, a smoothed

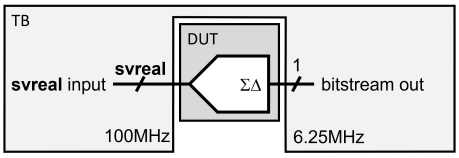


Figure 6.1: Block diagram of simulation setup for standalone modulator

spectrum is presented. The frequency responses of different runs are compared, rather than the time-domain signals in order to be less susceptible to small numeric differences that could arise from the different number formats used in the Simulink models (floating-point) and the emulated models (fixed-point). The ideal NTF of each model is calculated using the MATLAB [32] Delta-Sigma toolbox [33], and is overlaid on the frequency response. The expectation is that the frequency responses show the 20dB/decade and 40dB/decade behaviour of the first- and second-order response respectively.

For each run, the SNR will be calculated in the conventional 20kHz audio bandwidth f_b. This can be compared to the SN_QR calculated using Equation (5.1). The SNR calculation is performed by considering the signal power as the power in 3 bins around 1000Hz of the main harmonic. SNR is then calculated as:

$$SNR = 10\log_{10}\left(\frac{P_{signal}}{P_{total} - P_{signal}}\right)$$

In the following, the details of the simulation and emulation are discussed.

6.1.1 HDL Simulation

The block diagram of the simulation setup for the standalone modulator is shown in Figure 6.1. The input is generated at 100MHz as a real number and converted to an **svreal** type in the testbench. Due to a limitation of the testbench and **anasymod** framework, the outputs are all (over)sampled at 100MHz, though the block diagrams show the underlying sample rates of the data.

6.1.2 Emulation

As discussed previously in Section 3.3.1, the BRAM limitation prohibits longer emulation runs. This is especially true for the slow audio sample rates in the ΣΔ modulator.

Consider that one period of a 1kHz sine wave sampled at 100MHz consists of 100,000 samples. Monitoring the input and output of the modulator requires 26 bits (25 for the **svreal** input, and 1 for the digital output) so a total 2.6Mb of data, This amount of data is more than half of the available BRAM on the FPGA available. Even if no other parts of the design require BRAM, only about 1.5 periods of data could be collected.

Therefore, emulation of the modulator is not practical, because it cannot generate enough data for meaningful spectral analysis, thus excluding verification by actual emulation. However, event-driven HDL simulation of the synthesized gate-level model (post-synthesis simulation) is possible, which should show identical behaviour to the emulation running on the FPGA. While accurate, post-synthesis simulation should be reserved for verifying and debugging the emulation itself – as opposed to

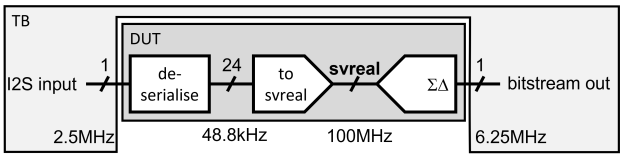


Figure 6.2: Block diagram of emulation setup for standalone modulator

verifying models – as it runs significantly slower than emulation. This is also the reason why the run is shorter than the others.

Post-synthesis simulation results will be presented and for the rest of the chapter *emulation* and *post-synthesis simulation* will be used interchangeably. A possible solution to the BRAM limitation is given in the recommendations in Section 8.3

The setup used in post-synthesis simulation is shown in Figure 6.2. The reasons for – and functions of – the I2S input and `deserialise` and `to_svreal` blocks will become apparent in Chapter 7. For now, it is sufficient to say that a sine wave input is generated at the input to the modulator, with the full range of the **svreal** floating point type.

6.2 Synthesis Results

In order to get an idea of the size of the models, the area results are presented in Tables 6.1 and 6.2 for the first and second-order models respectively. Percentages less than 0.01% are rounded up to 0.01%. The FPGA used is a Xilinx XC7Z020 with 53,200 LUTs, 106,400 FFs, 140 BRAM blocks and 220 DSP slices [34]. The total number of LUTs is split into LUTs used for logic and LUTs used for memory (LUTRAM). BRAM is given in 36Kb blocks for a total of 4.9Mb.

It is evident that the **anasymod** control infrastructure takes the majority of the resources. However, that comes down to the small size of the ΣΔ models. The large BRAM usage of the `traceport` module is due to BRAM being reserved for monitoring signals during emulation.

6.2.1 Timing Violations

Important to note are timing violations encountered during simulation. The violations are setup violations in the **anasymod** control infrastructure from the VIO control inputs (`sim_ctrl_gen`), through the control infrastructure to the time manager and further to the emulation clock generator (`emu_clks`). For reference of these blocks and signals, refer back to Section 3.3 for Figure 3.2 and the detailed description of the control infrastructure in Appendix B. The signal associated with the violated path is `emu_ctrl_data` which is used to pass timing data to the time manager in order to pause the emulation or force a certain timestep. There are several setup violations along this path:

- From VIO control inputs (`sim_ctrl_gen`) to the time manager (`time_manager`) with a slack of -2.8ns.
- From VIO control inputs (`sim_ctrl_gen`) to the oscillator (`osc_model`) with a slack of -2.7ns.

Table 6.1: Area report of first-order modulator model

	LUT Logic		LUTRAM		FF		BRAM		DSP	
	n	%	n	%	n	%	n	%	n	%
ΣΔ model	52	0.10	0	0	50	0.05	0	0	2	0.91
anasymod	1981	3.72	193	1.11	3307	3.10	29.5	21.07	0	0
↳ `clk`	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*
↳ `emu_clks`	*2*	*0.01*	*0*	*0*	*2*	*0.01*	*0*	*0*	*0*	*0*
↳ `osc_model`	*83*	*0.16*	*0*	*0*	*64*	*0.06*	*0*	*0*	*0*	*0*
↳ `time_manager`	*80*	*0.15*	*0*	*0*	*64*	*0.06*	*0*	*0*	*0*	*0*
↳ `ctrl_anasymod`	*258*	*0.48*	*0*	*0*	*24*	*0.02*	*0*	*0*	*0*	*0*
↳ `sim_ctrl_gen`	*372*	*0.70*	*0*	*0*	*820*	*0.77*	*0*	*0*	*0*	*0*
↳ `trace_port`	*754*	*1.42*	*169*	*0.97*	*1594*	*1.50*	*29.5*	*21.07*	*0*	*0*
↳ debug	*432*	*0.81*	*24*	*0.14*	*739*	*0.69*	*0*	*0*	*0*	*0*

Table 6.2: Area report of second-order modulator model

	LUT Logic		LUTRAM		FF		BRAM		DSP	
	n	%	n	%	n	%	n	%	n	%
ΣΔ model	134	0.25	0	0	118	0.11	0	0	7	3.18
anasymod	1976	3.73	193	1.11	3307	3.11	29.5	21.07	0	0
↳ `clk`	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*
↳ `emu_clks`	*2*	*0.01*	*0*	*0*	*2*	*0.01*	*0*	*0*	*0*	*0*
↳ `osc_model`	*83*	*0.16*	*0*	*0*	*64*	*0.06*	*0*	*0*	*0*	*0*
↳ `time_manager`	*80*	*0.15*	*0*	*0*	*64*	*0.06*	*0*	*0*	*0*	*0*
↳ `ctrl_anasymod`	*254*	*0.48*	*0*	*0*	*24*	*0.02*	*0*	*0*	*0*	*0*
↳ `sim_ctrl_gen`	*372*	*0.70*	*0*	*0*	*820*	*0.77*	*0*	*0*	*0*	*0*
↳ `trace_port`	*753*	*1.42*	*169*	*0.97*	*1594*	*1.50*	*29.5*	*21.07*	*0*	*0*
↳ debug	*432*	*0.81*	*24*	*0.14*	*739*	*0.69*	*0*	*0*	*0*	*0*

- From VIO control inputs (`sim_ctrl_gen`) to the time manager (`emu_clks`) with a slack of -10.8n.

Fortunately, the timing violations are not relevant for the testing methodology used, as the BRAM limitations prevent use of the **anasymod** control infrastructure for emulation anyway. Brief testing for short emulation durations has shown that emulation still seems to work, likely because the fixed-timesep approach is less reliant on the control infrastructure. Testing was not comprehensive enough to provide more insight.

The above should not take away from the severity of the error however, which definitely should be solved. While a simple solution would be to decrease the clock frequency of the control infrastructure, this also limits the emulation clock, decreasing time resolution.

In this case, the total setup time for the last violation is 20.8ns ($10.8\text{ns} + \frac{1}{100\text{MHz}}$). Reducing the emulation clock to fix the violation would require a clock of 48MHz; less than half of the time resolution.

6.3 First-Order Model

Testing the standalone ΣΔ modulator with the setup from Figure 6.1, produces the results shown in Figure 6.3. The fixed-point **msdsl** models (*Simulation* and *Emulation*) produce similar results, though significantly different to the floating-point simulations. All runs however, show SNR comparable to the theoretical 58dB, and show expected 20dB/decade first-order noise shaping. Further, all models show a pronounced third harmonic.

The lower SNR of the *Simulink* and floating-point **msdsl** model (*Sim. Real*) is surprising, but clearly related to the difference in response, with those responses showing extreme peaks. These responses are not smoothed, in order to show this behaviour clearly. Since the number format is what these models have in common, that is likely the cause of this behaviour.

6.4 Second-Order Model

Testing the second-order ΣΔ modulator produces the results shown in Figure 6.4. Simulink, **msdsl** simulation and emulation all produce similar results, showing the expected second-order noise shaping, but have a lower SNR than the theoretical SNR of 92dB. These results are closer to expected results than for the first-order model, with the peaks in the floating-point models no longer present. The emulated bitstream shows more harmonics than the rest.

6.5 Discussion

While both models show approximately the desired noise shaping behaviour, the results are not entirely conclusive. The expectation would be that the *Simulink* and *Sim. Real* runs have almost identical behaviour with slightly higher SNR than the *Simulation* and *Emulation* runs, which should also show near identical behaviour. Differences between Simulation and emulation can be related to the difference in run times. The floating-point effects in the first-order model remain unexplained, however. On the other hand, there is no direct evidence for the **msdsl** models being incorrectly modelled.

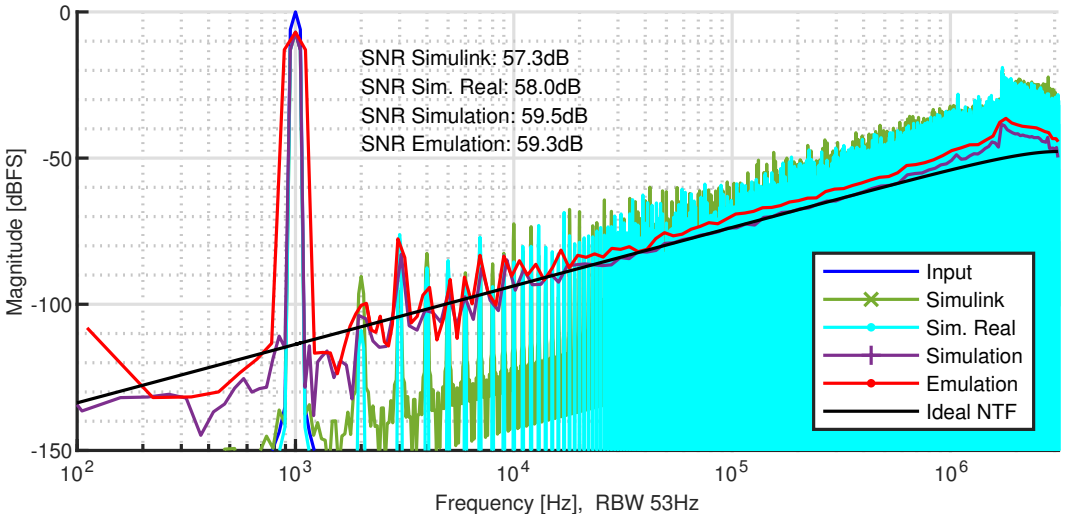


Figure 6.3: Comparison of power spectra of bitstreams of first-order modulator

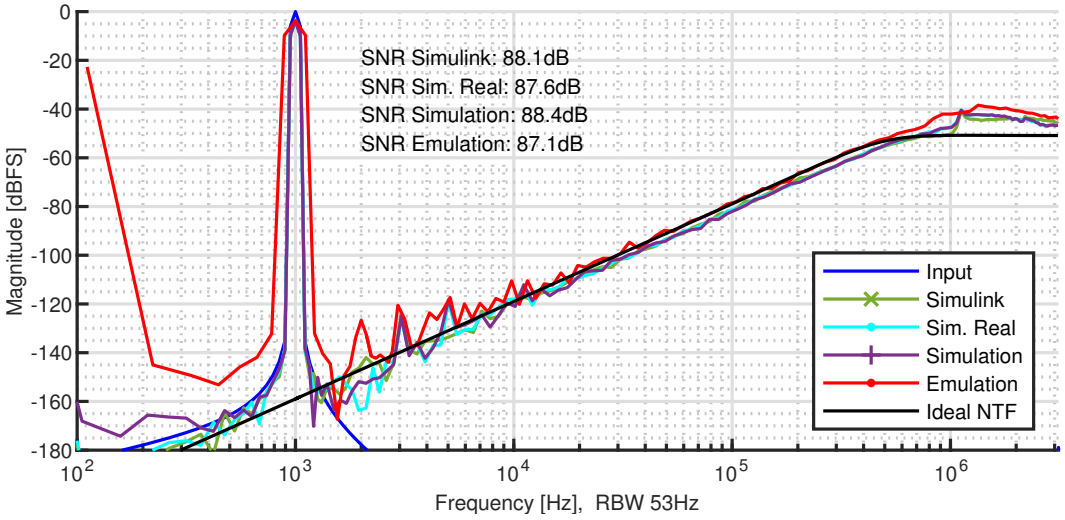


Figure 6.4: Comparison of power spectra of bitstreams of second-order modulator

Chapter 7

Demo

In order to show the capabilities of emulation and parameter sweeping in real-time, a demo has been set up and measured. The demo involves the implementation of the CTΣΔ models on an FPGA development board. The chosen board is a ZedBoard, with a Xilinx Zynq-7020 SoC. This SoC incorporates the same XC7Z020 FPGA that is targeted for synthesis in Section 6.2 (the PL part), as well as an ARM processor (PS) which is not used the scope of the demo [34]. Moreover, the board incorporates an audio codec, and ADCs which are accessible from the PL of the SoC [35].

The details of this implementation will be discussed in this chapter, starting with the decimation structure in Section 7.1. Then the interfaces with the codec and ADCs are presented in Sections 7.2 and 7.3 respectively, including the challenges encountered when interfacing **svreal** types with existing HDL descriptions. Synthesis results are presented in Section 7.4. Verification and measurement of the CTΣΔ models are described in Sections 7.5 and 7.6 respectively, while the result and presented in Section 7.7. A discussion of the demo setup and measurement results in Section 7.8 concludes this chapter.

7.1 Decimation

The output of the decimation was originally a 16-bit 48.8kHz signal in [31], but has been adapted to 24-bit to fit the audio codec on the ZedBoard. The decimation structure decimates the 1-bit 6.25MHz bitstream into a 24-bit 48.8kHz audio signal. The structure consists of three CIC filters, followed by FIR compensation filter which outputs the final 24-bit signal. The CICs are implemented in custom VHDL, while the compensation filter is implemented with the Xilinx FIR Compiler IP [36]. The original sizes of the CICs have been increased, adding more bits at each intermediary stage. Table 7.1 gives the details of the CIC decimation stages.

A separate SystemVerilog module generates the ΣΔ sample clock and the necessary clocks for decimation from the 100MHz emulation clock `CLK_MSDSL`.

7.2 CODEC

In order to implement the CTΣΔ model as a standalone demo on an FPGA, there needs to be a way of interfacing with analog signals. To this end, the Audio codec on the ZedBoard is used, an Analog Devices ADAU1761 [37]. The codec contains

Table 7.1: Parameters of CIC decimation stages

Stage	Name	Decimation Factor	Sample Rate In	Sample Rate Out	Order	Intermediary Bits	Bits Out
1	CIC0	8	6.25MHz	781kHz	2	10	8
2	CIC1	8	781kHz	97.7kHz	4	15	13
3	CIC2	2	97.7kHz	48.8kHz	16	29	16

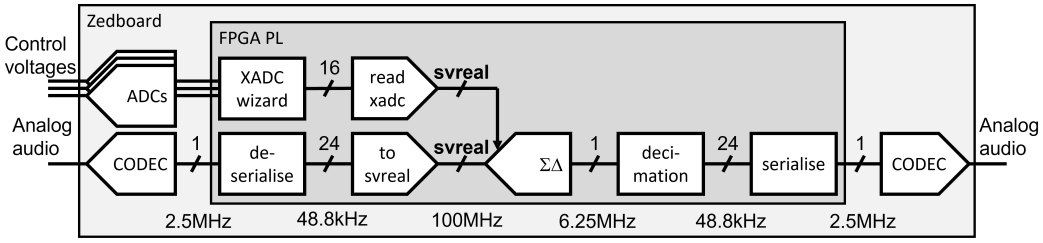


Figure 7.1: Complete signal chain of demo setup with parameter sweeping

an ADC which generates a *Inter-IC Sound* (I2S) signal to send audio data to the FPGA, and a DAC which takes an I2S signal from the FPGA. The interface to the codec is based on work by Isaac Verdu [38], who implements an SPI connection to configure the codec and blocks to serialize and deserialize the I2S signal to a 24-bit audio signal[1].

An extra DAC step is necessary to convert the deserialized I2S signal to an "analog" **svreal** type for the modulator input. This is done by the block `to_svreal`. An overview of the signal chain of the FPGA demo is shown in Figure 7.1. The more detailed RTL hierarchy is given in Figure 7.2.

7.3 XADC

In order to sweep the desired parameters, a control input is required. In emulation this input would be set using the **anasymod** control infrastructure. However, use in a full FPGA implementation such as done with the CTΣΔ models requires an external input. To this end, the ADCs on the ZedBoard – part of the Xilinx Analog-to-Digital Converter (XADC) – are used to read a voltage set using a potentiometer. Using the Xilinx XADC Wizard [39], up to 3 ADCs can be used to set parameter sweeps. Figure 7.1 shows how the XADC on the ZedBoard is integrated for parameter sweeping.

7.4 Synthesis

The synthesis results of the full demo with the second-order modulator, decimator and parameter sweeping of `in_gain2`, `fb_gain` and `fb_gain2` are shown in Table 7.2. Percentages less than 0.01% are rounded up to 0.01%. Comparing these to the area reports in the previous sections, the **anasymod** framework uses the same area. Again, the large BRAM usage of the `traceport` module is due to BRAM being

[1]This work is available on GitLab (https://gitlab.com/rtlaudiolab/fpga_audio_processor)

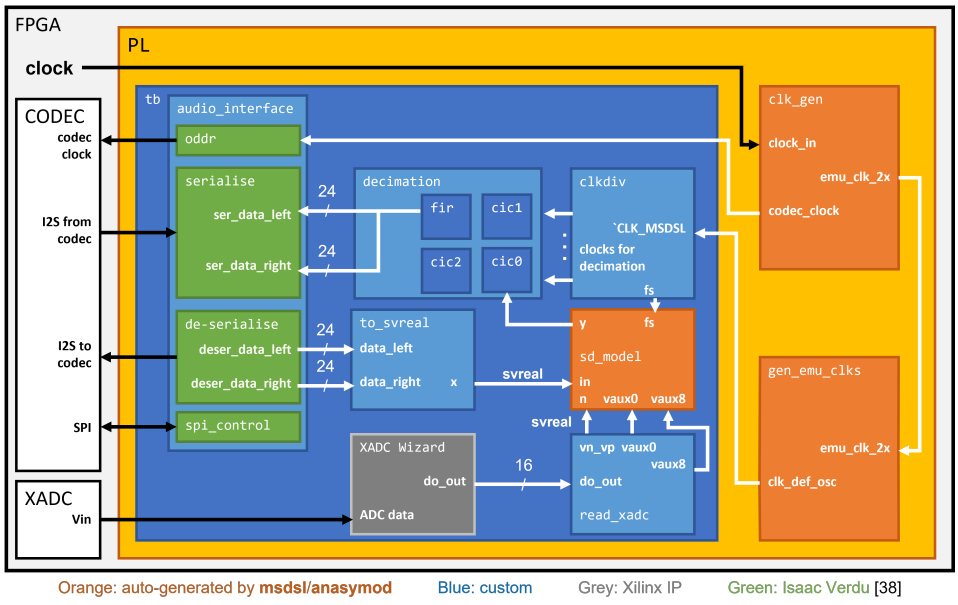


Figure 7.2: Block diagram of demo HDL description

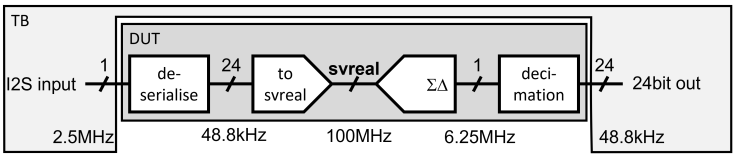


Figure 7.3: Block diagram of verification signal chain of demo

reserved for monitoring signals during emulation, although for the demo use case this could be reduced by adjusting the appropriate **anasymod** settings.

Comparing to the ΣΔ modulator from the previous chapter – which is identical except for the added parameter sweeping – shows a clear increase in BRAM usage. This is no surprise, as the implementation relies on tables stored in BRAM. Parameter sweeping also results in an increase in the number of LUTs involved in calculating addresses. Of the additional five DSPs, three are related to scaling the sweep control signals, while two are used to invert `fb_gain` and `fb_gain2`.

The area added by the digital design is dominated by the decimation, but nowhere near the limits of the FPGA, and not limiting to **anasymod** or the **msdsl** model, even with the extra overhead from parameter sweeping.

The same timing violations are encountered as described in Section 6.2.1. These do not have an impact on the demo, however, as the **anasymod** control infrastructure is not used.

7.5 Verification

In order to verify functionality of the demo setup, post-synthesis HDL simulation is done in the manner of Figure 7.3. The full system is simulated with an I2S input. The output is sampled before the deserializer, in order to skip the step of converting the final I2S signal. The input a 1V 1kHz sine wave for a duration of 10ms.

Table 7.2: Area report of demo with second-order model and parameter sweeping

	LUT Logic		LUTRAM		FF		BRAM		DSP	
	n	%	n	%	n	%	n	%	n	%
ΣΔ model	259	0.49	0	0	117	0.11	1.5	1.07	13	5.91
↳ coefficients	*0*	*0*	*0*	*0*	*0*	*0*	*1.5*	*1.07*	*0*	*0*
anasymod	1978	3.72	193	1.11	3307	3.10	29.5	21.07	0	0
↳ `clk`	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*
↳ `emu_clks`	*2*	*0.01*	*0*	*0*	*2*	*0.01*	*0*	*0*	*0*	*0*
↳ `osc_model`	*83*	*0.16*	*0*	*0*	*64*	*0.06*	*0*	*0*	*0*	*0*
↳ `time_manager`	*80*	*0.15*	*0*	*0*	*64*	*0.06*	*0*	*0*	*0*	*0*
↳ `ctrl_anasymod`	*255*	*0.48*	*0*	*0*	*24*	*0.02*	*0*	*0*	*0*	*0*
↳ `sim_ctrl_gen`	*372*	*0.70*	*0*	*0*	*820*	*0.77*	*0*	*0*	*0*	*0*
↳ `trace_port`	*754*	*1.42*	*169*	*0.97*	*1594*	*1.50*	*29.5*	*21.07*	*0*	*0*
↳ debug	*432*	*0.81*	*24*	*0.14*	*739*	*0.69*	*0*	*0*	*0*	*0*
digital	1050	1.97	47	0.27	1814	1.70	0	0	1	0.45
↳ `clkdiv`	*11*	*0.02*	*0*	*0*	*16*	*0.02*	*0*	*0*	*0*	*0*
↳ `to_svreal`	*36*	*0.07*	*0*	*0*	*23*	*0.02*	*0*	*0*	*0*	*0*
↳ `decimation`	*633*	*1.19*	*47*	*0.27*	*1376*	*1.29*	*0*	*0*	*1*	*0.45*
↳ `xadc_wiz`	*0*	*0*	*0*	*0*	*0*	*0.05*	*0*	*0*	*0*	*0*
↳ `read_xadc`	*3*	*0*	*0*	*0*	*54*	*0*	*0*	*0*	*0*	*0*
↳ `codec`	*367*	*0.69*	*0*	*0*	*345*	*0.32*	*0*	*0*	*0*	*0*

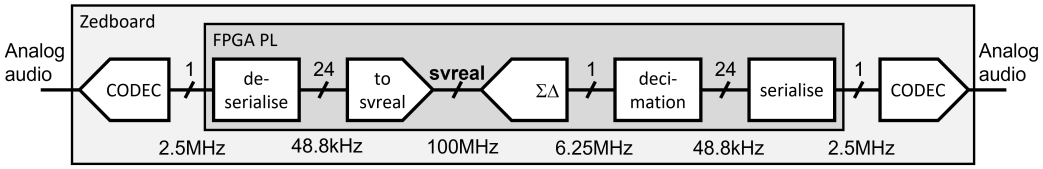


Figure 7.4: Signal chain of demo measurement setup

7.6 Measurements

After synthesis, the full ΣΔ converter is characterised on the FPGA. The block diagram in Figure 7.4 shows the full system including the audio codec. For these measurements no parameter sweeping is enabled. Measurements have been done using the NI myDAQ as function generator and oscilloscope. Like in previous simulations and emulations, the input is a 1kHz sine wave at 1V.

The codec is characterised in the same manner as the converter, by sending the data from the serializer directly to the deserializer, as shown in Figure 7.6.

7.7 Results

The post-synthesis results are shown in Figures 7.7 and 7.8 for the first-order and second-order models respectively. These results show the spectra at different points in the signal chain; after the bitstream (which is identical to the results in the previous chapter), after the last CIC decimator (CIC2), and at the audio output after the FIR filter. The droop introduced by the CICs is clearly visible, and barely compensated by the FIR filter.

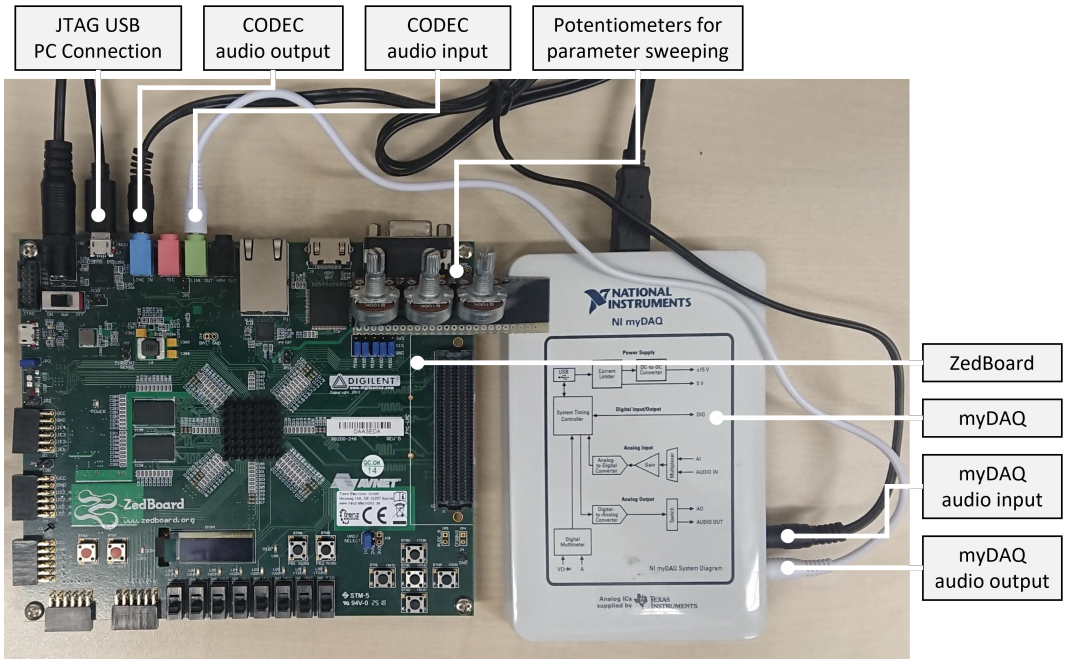


Figure 7.5: Physical demo measurement setup

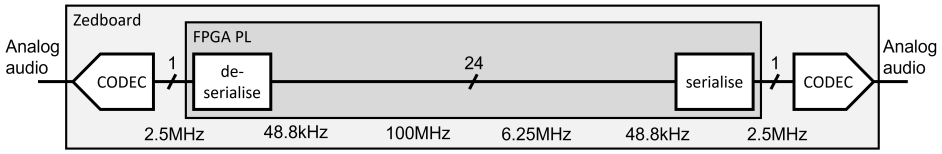


Figure 7.6: Block diagram of codec verification

The measured response, compared to the standalone codec, for the second-order CTΣΔ is given in Figure 7.9 Comparing the measured results to the simulations and emulation in the previous chapter shows clearly the adverse effects of decimation. Due to the decimation stage in between, little can be said for the effect of emulation on the results. However, given the attenuation shown in the intermediary stages in Figure 7.7, 52dB SNR is plausible. The third harmonic present post-simulation is not visible. Instead, there is a visible fourth harmonic.

For the second-order modulator, the results are shown in Figure 7.10 Compared to the first-order modulator, the SNR is better as a result of higher order noise shaping, but lower than expected. Furthermore, the noise floor is no longer flat.

7.8 Discussion

There are several noteworthy points in the post-synthesis and measurement results. The first is that the effects of the FIR compensation filter are not visible, indicating incorrect design of coefficients. The compensation filter only attenuates the output. Second is the shaped noise floor of the second-order CTΣΔ measurements.

The measurements do show that the demo works, however. While having low quality, the CTΣΔ does produce a distinguishable output. The design is probably equally hindered by poor design of the decimation, as it is by effects of emulation.

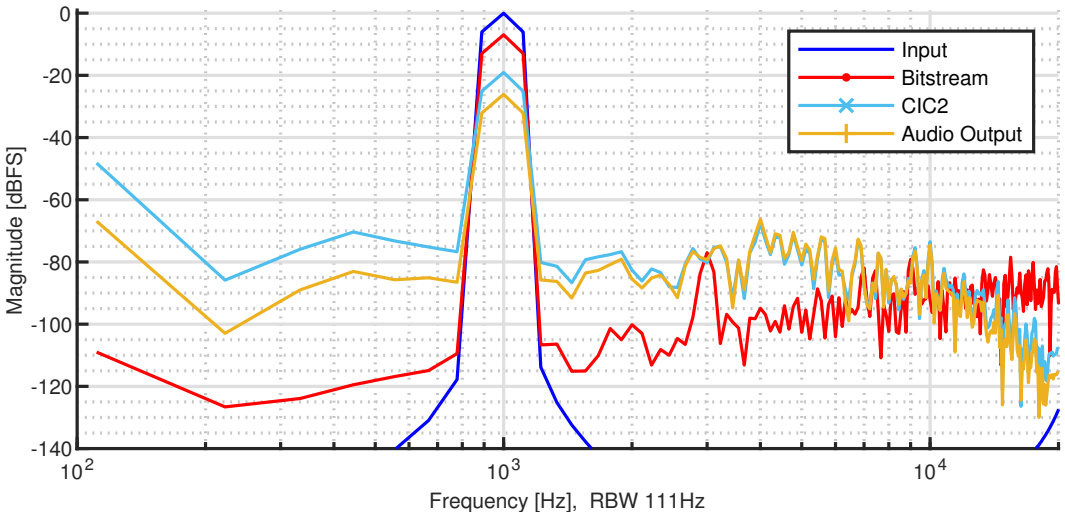


Figure 7.7: Post-synthesis simulation power spectra of bitstream and output of first-order converter

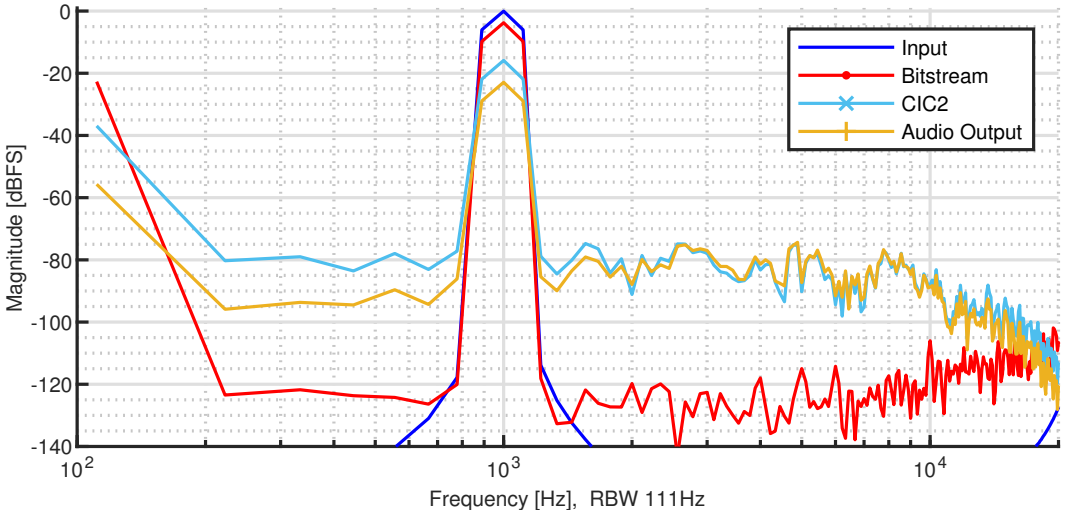


Figure 7.8: Post-synthesis simulation power spectra of bitstream and output of second-order converter

The 74dB SNR of the codec itself is also low. Though the equal noise floor of the converter and the standalone codec point to the rudimentary measurement setup as the cause.

With all the added complexity, the demo is not an effective way of accessing the correctness of an emulated model. Intermediary stages, including additional gain stages in the codec ADC and DAC all need to be taken into account before isolating the emulated design itself.

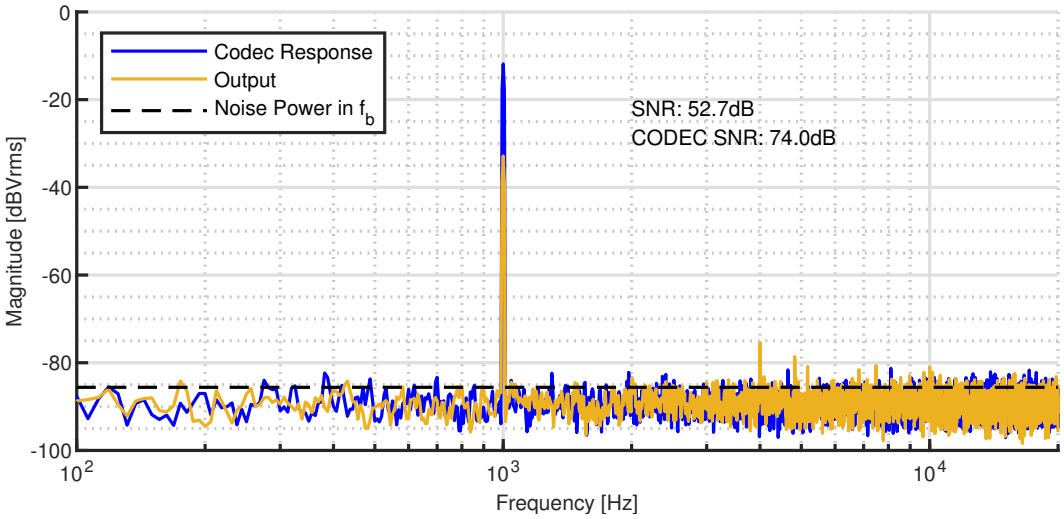


Figure 7.9: Measured power spectrum of first-order converter

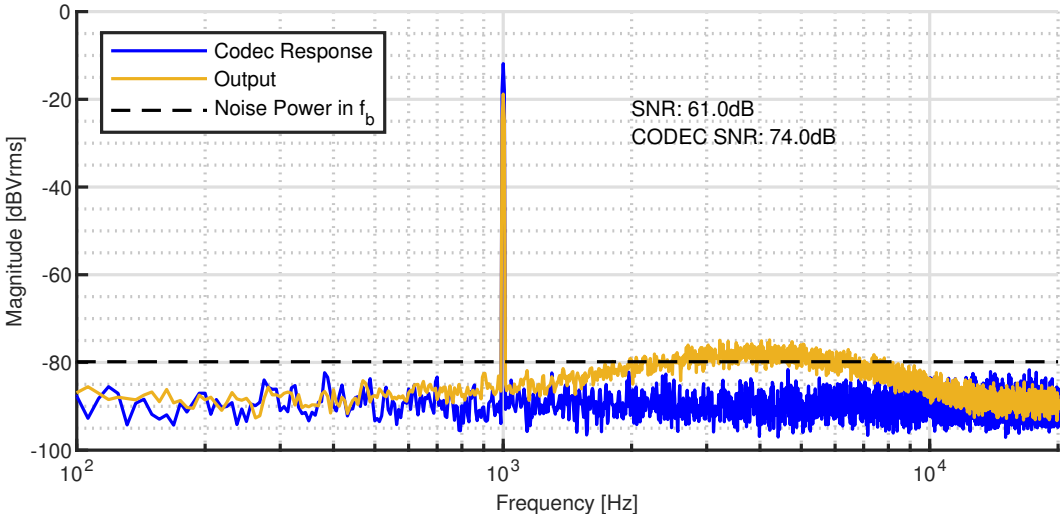


Figure 7.10: Measured power spectrum of second-order converter

Chapter 8

Conclusions and Recommendations

8.1 Conclusions

The guiding research question for this thesis was:

Can an analog mixed-signal design be emulated in real-time on an FPGA?

Before answering the overarching question, consider the sub-questions.

1. *What tools are available for emulation on an FPGA?*

Available tools were explored in Chapter 2. Only very few tools exist that support emulation, and of these only one is open-source. Therefore, the set of tools **svreal**, **msdsl** and **anasymod** were chosen as a basis for further exploration of emulation.

2. *Which designs are suitable for emulation on an FPGA?*

Whether a design is suitable for emulation comes down to the required accuracy of the design. With regard to the size of design that is suitable for emulation, Table 7.2 makes clear that emulation overhead takes only a small fraction of available FPGA resources, and the CTΣΔ model itself is also small, leaving space for much larger designs. On the other hand, the available resources can be used to improve the accuracy of the emulation, as the size of a model is related to the number of coefficients in the state space description.

3. *What models are appropriate for modelling analog behaviour?*

The modelling options are explored broadly in Section 2.3 and in detail for **msdsl** in Section 3.2. The preference for modelling methods is indeed to reduce the amount of manual derivation. Thus, many tools focus on generating models from a netlist. In **msdsl** however, this method is limited in complexity. Therefore, combining modelling methods is the most comprehensive way to model a system, as done in the CTΣΔ converter model. The designer can choose between state-update equation, differential equation, or transfer function modelling where the netlist is not sufficient. Fortunately, this is easy to do in **msdsl** as modelling methods are easy to work with, and each produce identical results, reducing the number of possible modelling mistakes due to incorrect choice of technique.

4. *Which advantages does emulation present over conventional simulation?*
As the results in Chapter 6 show, emulation comes very close to the accuracy of Simulink simulation. But, due to the use of fixed-point numbers, the accuracy of, for example SPICE simulations can not be matched. The level of detail possible with emulation, as mentioned already, comes down to the available FPGA resources. However, Chapter 4 presents a clear use case that cannot be matched by simulation. Parameter sweeping was implemented in a way that makes it easy to add to existing models, and thus use emulation for design-space exploration.

In short, the answer to the main research question:

Can an analog mixed-signal design be emulated in real-time on an FPGA?

Would be a resounding *yes* for the CTΣΔ ADC. The FPGA demo shows that reaching real-time emulation is well possible, while barely using the available FPGA resources. However, due to the BRAM limitation that prevents long emulation runs, it is hardly usable for verification using existing tools, where being able to monitor internal signals directly is a requirement. Even if the BRAM limitation had not been, however, the timing errors encountered during synthesis may have hindered emulation, especially in real-time. The target of real-time emulation may not be practical, and the emulation duration may be limited, but with the tools **svreal**, **msdsl** and **anasymod**, there is a framework for emulating AMS designs on FPGA. The limits to size and complexity are the resources of the FPGA and the necessary manual derivation required to model a design with the existing built-in techniques. If it were not for the shortcomings of the tool regarding signal monitoring and netlist modelling, emulation does present a worthwhile workflow for verifying AMS designs.

8.2 Discussion

The conclusion presents a mix of promising results and large pitfalls. This is partly due to shortcomings in the tool that were not able to be remedied in the scope of this thesis, and partly due to the method used to evaluate emulation.

Starting with the latter, while the CTΣΔ converter is a suitable design for emulation, it is not complex enough to show the true capabilities of emulation, or even make use of many of the modelling techniques. A more representative design would be a transistor level design with more states. On the other hand, this would require more manual derivation, as only few components are available to the netlist modelling method of **msdsl**, while no manual derivation was necessary for the ΣΔ models because of the ideality of the models. What further confuses the testing with the CTΣΔ converter is the immaturity of the design. In the demo it is still unclear whether some effects are due to the emulated model, or mistakes in the decimation implementation.

The extension for parameter sweeping is still in its infancy as well, limited to sweeping states in update equations. When parameter sweeping becomes especially interesting, is when not just state variables, but also the parameter from which state variables are derived, can be swept. Only more development and testing will show whether design-space exploration with emulation proves to be a useful addition to an analog designer's toolbox.

With regard to the tools, not being able to extend the emulation duration, and therefore not doing much emulation, makes the conclusion somewhat vague. The low (audio) sample rates do not lend themselves to shorter testing within the BRAM limitation encountered. Thus, the limited emulation testing that was done is not presented, as it could not give any insight into the correctness of the emulated model.

8.3 Recommendations

Based on the discussion, it is evident that there is still much work to be done on emulation; both in the exploration of existing tools, and in the development of these tools.

8.3.1 Emulation Infrastructure

Starting with the largest obstacle to emulation in this work, the emulation duration needs to be addressed. First, the ILA settings can be tuned, but the only documentation for how these settings are implemented is in the comments of the **anasymod** source itself. Another solution exists, and is a method for stopping emulation when BRAM is full, exporting the data to a host PC, and then resuming emulation, writing to the same BRAM again. This involves an appropriate sequence of VIO commands, as well as making sure that the timing of emulations is not affected. Implementation of either of these solutions was not possible in the time-frame of this thesis. Ideally, there is no limit to the duration of an emulation, or signals that can be monitored.

Further, the effects of the timing errors need to be explored. Since it is the limiting factor for emulation speed, an effort should be made to reduce the critical paths in the **anasymod** control infrastructure, if indeed the present errors impact emulation. If not, these parts of the infrastructure may be removed for fixed-timestep emulation, simplifying the control infrastructure.

8.3.2 Model Verification

While this thesis makes a start at verifying the accuracy of the models produced by **msdsl**, it could be more thorough. More testing is required with more mature designs. This could also lead into more detailed comparison to conventional verification techniques, comparing accuracy, detail and speed in both the time and frequency domain. Ideally, emulation should be implemented on a design that has already been tested by an industry standard AMS verification method, and then compared.

8.3.3 Modelling

As mentioned in the discussion, future testing should use more mature, and more complex designs also to push the limits of synthesizable RTL modelling. In order to avoid the manual derivation that comes with modelling such designs, a beneficial extension to **msdsl** could be the building of a library of non-ideal components with definable parameters, such as op-amps and comparators or, on a lower level, transistors. This would make it easier to model real circuits and pave the way for automatic model generation from, for example, an existing SPICE netlist. Entirely removing manual modelling should be the goal to make emulation a useful tool.

Another shortcoming of the models that should be addressed, is behaviour related to the **svreal** fixed-point number format. Support for saturation as an alternative to wrap-around can prevent fixed-point-range-related modelling mistakes.

Variable-timestep modelling is almost entirely neglected in this work, despite being the main reason for many of the functions of the **anasymod** control infrastructure. Because of how close it comes to conventional event-driven simulation, it certainly warrants its own exploration, especially as a way of improving time-resolution in emulation.

Finally, the parameter sweeping extension to **msdsl** described in Chapter 4 is still only a basic implementation. To be truly useful, it should be extended to sweep parameters in several modelling methods, beginning with the ability to sweep a parameter that affects multiple state-space coefficients.

Bibliography

[1] L. W. Nagel and D. Pederson, "SPICE (simulation program with integrated circuit emphasis)," EECS Department, University of California, Berkeley, Tech. Rep. UCB/ERL M382, Apr. 1973. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/1973/22871.html

[2] D. Stanley, C. Wang, S.-J. Kim, S. Herbst, J. Kim, and M. Horowitz, "Fast validation of mixed-signal SoCs," *IEEE Open Journal of the Solid-State Circuits Society*, vol. 1, pp. 184–195, 2021. doi: 10.1109/OJSSCS.2021.3122397

[3] S. G. Herbst, "An open-source framework for FPGA emulation of analog/mixed-signal integrated circuit designs," Ph.D. dissertation, Stanford University, Jun. 2021. [Online]. Available: http://purl.stanford.edu/gj828vr5382

[4] S. Herbst, G. Rutsch, W. Ecker, and M. Horowitz, "An open-source framework for FPGA emulation of analog/mixed-signal integrated circuit designs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 7, pp. 2223–2236, Jul. 2022. doi: 10.1109/TCAD.2021.3102516

[5] M. A. Zidan, A. G. Radwan, and K. N. Salama, "The effect of numerical techniques on differential equation based chaotic generators," in *ICM 2011 Proceeding*, 2011, pp. 1–4. doi: 10.1109/ICM.2011.6177395

[6] Y. D. Staal, "FPGA emulation of analog subsystems for mixed-signal integrated circuit prototyping," Bachelor's thesis, University of Twente, 2021. [Online]. Available: http://essay.utwente.nl/87535/

[7] 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*, vol. 45, no. 8, pp. 1997–2020, Oct. 2012. doi: 10.1016/J.MEASUREMENT.2012.04.022

[8] B. S. Deepaksubramanyan, P. Parakh, Z. Chen, H. Diab, D. Marcy, and F. H. Schlereth, "An FPGA-based MOS circuit simulator," in *48th Midwest Symposium on Circuits and Systems*, vol. 1, 2005, pp. 655–658 Vol. 1. doi: 10.1109/MWSCAS.2005.1594186

[9] S. Herbst. (2021) msdsl. [Online]. Available: https://git.io/msdsl

[10] W. Wu, Y. L. Chen, Y. Ma, C. N. J. Liu, J. Y. 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)*, Jul. 2016, pp. 1286–1289. doi: 10.1109/ISCAS.2016.7527483

[11] 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)*, Jul. 2017, pp. 1–4. doi: 10.1109/SMACD.2017.7981562

[12] R. A. Cottrell, "Event-driven behavioural simulation of analogue transfer functions," in *Proceedings of the European Design Automation Conference, 1990., EDAC.*, 1990, pp. 240–243. doi: 10.1109/EDAC.1990.136652

[13] F. Pichon, S. Blanc, and B. Candaele, "Mixed-signal modelling in VHDL for system-on-chip applications," in *Proceedings the European Design and Test Conference. ED&TC 1995.* Association for Computing Machinery, Inc, Mar. 1995, pp. 218–222. doi: 10.1109/EDTC.1995.470400

[14] 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*, vol. 63, no. 1, pp. 23–33, Jan. 2016. doi: 10.1109/TCSI.2015.2512699

[15] J. E. Jang, M. J. 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*, 2012, pp. 1–4. doi: 10.1109/CICC.2012.6330558

[16] S. Herbst, B. C. Lim, and M. Horowitz, "Fast FPGA emulation of analog dynamics in digitally-driven systems," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov. 2018, p. 1–8. doi: 10.1145/3240765.3240808

[17] J. Kim, "Efficient simulation of analog/mixed-signal circuits in SystemVerilog with auto-generated models," presented at the 2022 IEEE Custom Integrated Circuits Conference (CICC), Apr. 2022, Presentation.

[18] XMODEL: Empower SystemVerilog with event-driven analog models. Scientific Analog. [Online]. Available: https://www.scianalog.com/xmodel/

[19] B. C. Lim and M. Horowitz, "An analog model template library: Simplifying chip-level, mixed-signal design verification," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 1, pp. 193–204, Jan. 2019. doi: 10.1109/TVLSI.2018.2873387

[20] S. Herbst. (2021) svreal. [Online]. Available: https://git.io/svreal

[21] G. Rutsch and S. Herbst. (2021) anasymod. [Online]. Available: https://git.io/anasymod

[22] J. R. Hauser. (2019) Berkeley HardFloat. [Online]. Available: http://www.jhauser.us/arithmetic/HardFloat.html

[23] Z. Gajic, *Linear Dynamic Systems and Signals*. Prentice Hall, 2002.

[24] (2022) scipy.signal.cont2discrete. The SciPy Community. [Online]. Available: https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.cont2discrete.html

[25] S. Williams. (2021) Icarus Verilog. [Online]. Available: http://iverilog.icarus.com/

[26] (2020) Vivado v2020.1. Xilinx, Inc. San Jose. [Online]. Available: https://www.xilinx.com/products/design-tools/vivado.html

[27] *Virtual Input/Output v3.0 Product Guide (PG159)*, Xilinx Inc., 2018. [Online]. Available: https://docs.xilinx.com/v/u/en-US/pg159-vio

[28] *Integrated Logic Analyzer (ILA) v6.2 Product Guide (PG172)*, Xilinx Inc., 2016. [Online]. Available: https://docs.xilinx.com/v/u/en-US/pg172-ila

[29] M. J. Pelgrom, *Analog-to-Digital Conversion*, 4th ed. Springer Cham, 2022. doi: 10.1007/978-3-030-90808-9

[30] E. B. Hogenauer, "An economical class of digital filters for decimation and interpolation," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 29, no. 2, pp. 155–162, Apr. 1981. doi: 10.1109/TASSP.1981.1163535

[31] J. Huiden, L. van Dijk, M. van Minnen, and N. Sulzer, "System-on-chip design: Final project," Final project report, SoC Design course, University of Twente, Jan. 2021, unpublished.

[32] (2022) MATLAB R2022a. The Mathworks Inc. Natick, MA, USA. [Online]. Available: https://nl.mathworks.com/products/matlab.html

[33] R. Schreier, S. Pavan, and G. C. Temes, *Understanding Delta-Sigma Data Converters*, 2nd ed. John Wiley & Sons, Inc., Apr. 2017. doi: 10.1002/9781119258308

[34] *Zynq-7000 SoC Data Sheet: Overview (DS190)*, Xilinx, Inc., 2018. [Online]. Available: https://docs.xilinx.com/v/u/en-US/ds190-Zynq-7000-Overview

[35] *ZedBoard (Zynq Evaluation and Development) Hardware User's Guide*, Avnet, Inc., Jan. 2014, v2.2. [Online]. Available: https://digilent.com/reference/_media/zedboard:zedboard_ug.pdf

[36] *FIR Compiler v7.2 Product Guide (PG149)*, Xilinx Inc., 2021. [Online]. Available: https://docs.xilinx.com/v/u/en-US/pg149-fir-compiler

[37] "SigmaDSP Stereo, Low Power, 96 kHz, 24-Bit Audio Codec with Integrated PLL," Analog Devices, Inc., Oct. 2018. [Online]. Available: https://www.analog.com/media/en/technical-documentation/data-sheets/ADAU1761.pdf

[38] I. Verdu. (2021, Jul.) Zedboard audio processor. [Online]. Available: https://rtlaudiolab.com/blog/001-zedboard-audio-processor

[39] *XADC Wizard v3.3 Product Guide (PG091)*, Xilinx Inc., 2016. [Online]. Available: https://docs.xilinx.com/v/u/en-US/pg091-xadc-wiz

Appendix A

Post-Synthesis Simulation with anasymod

This appendix documents the extension to **anasymod**[1] enabling post-synthesis event-driven simulation with Xilinx Vivado directly from **anasymod**. The extension[2] is not meant to be final code, but a preliminary implementation.

A.1 Infrastructure

Interaction with Vivado is done by calling Vivado in headless mode with a TCL script containing commands to be run by Vivado. These TCL scripts are generated by **anasymod** based on templates, before being passed to Vivado. Enabling post-synthesis simulation thus requires sending the appropriate TCL commands to Vivado. The post-synthesis simulation workflow is as follows:

1. Create a Vivado project
2. Add necessary files to the project fileset
3. Run synthesis
   (Open existing project)
4. Create a post-synthesis simulation fileset
5. Open post-synthesis simulation
6. Configure and run post-synthesis simulation

The first three steps are identical to those of bitstream generation. Since bitstream generation does not take much time compared to synthesis, the first three steps are not implemented for post synthesis simulation, instead, an exiting project is opened. Thus, *running bitstream generation is required before running post-synthesis simulation.*

The main **anasymod** class is `Analysis`, in `analysis.py`. From this class, simulation, emulation, and other workflows can be called, including the new post-synthesis simulation. Starting a workflow amounts to calling a function in a class – in this case `VivadoEmulation` – which generates the TCL script that is passed to Vivado. This class is a subclass of `VivadoTCLGenerator`, which is a subclass of `CodeGenerator`, responsible, as the name suggest, for generating code to a file. The

[1] **anasymod** by S. Herbst and G. Rutsch is available on GitHub (https://git.io/anasymod)

[2] The extension is available as fork of **anasymod** on GitHub (https://github.com/nsulzer/anasymod)

generated TCL scripts contains commands for opening an existing project, creating the simulation fileset (step 4), open the simulation (step 5), and configuring and running the simulation (step 6).

Step 6 is executed by passing another TCL script to Vivado to configure the wave window, and run the simulation. This script can either be user-generated or automatically generated from **anasymod**. If the script is user-generated, it is passed as an argument to the workflow post-simulation workflow function in the `Analysis` class. Else it is generated by that function.

Note that there is a known bug in Vivado, due to which signals that are not shown in the signal window will not be considered in simulation, and therefore not written to file. One solution seems to be to re-run the simulation.

A.2 Implementation

The implementation adds three functions to three respective files. These are described below:

`open_project(project_name, project_directory,full_part_name)`

This function in the file `generators/vivado.py`, of the general `VivadoTCLGenerator` class generates TCL commands to open an existing Vivado project, based on the project location and naming conventions followed by **anasymod**. The function code is shown in Listing A.1.

`post_simulate(tcl_script)`

This function in the file `emu/vivado_emu.py`, of the `VivadoEmulation` class generates a TCL and calls Vivado with said script. The TCL script contains commands for opening a project, and steps 4-6 from the list above. The function code is shown in Listing A.2.

`simulate_post(tcl_script)`

This function in the file `analysis.py`, of the main **anasymod** `Analysis` class calls the function `post_simulate`. Before doing so, results folders are created, and no `tcl_script` is given, it is generated, via the `CodeGenerator` class. The function code is shown in Listing A.3.

A.3 Listings

Listing A.1: Generating TCL commands for opening a Vivado project

Python

```
41 def open_project(self, project_name,
   ↪  project_directory,full_part_name=None):
42     # open the project
43     cmd = ['open_project']
44     cmd.append('"'+back2fwd(project_directory+'/'+project_name)+'.xpr'"')
45     if full_part_name is not None:
46         cmd.extend(['-part', full_part_name])
47     self.writeln(' '.join(cmd))
```

Listing A.2: Generating TCL commands to run post-synthesis simulation in Vivado

Python

```
229 def post_simulate(self, tcl_script):
230
231     project_root = self.target.project_root
232
233     self.open_project(
234         project_name=self.target.prj_cfg.vivado_config.project_name,
235         project_directory=project_root
236         # full_part_name=self.target.prj_cfg.board.full_part_name
237     )
238
239     self.writeln('# Post synthesis simulation')
240     self.set_property('top', '{post_tb}', '[get_filesets {sim_1}]')
241     self.add_project_defines(content=self.target.content,
        ↪ fileset='[get_filesets {sim_1}]')
...
258     if tcl_script is not None:
259         self.set_property('{xsim.simulate.custom_tcl}',
            ↪ f'{{{back2fwd(tcl_script)}}}', '[get_fileset sim_1]')
260     self.writeln('launch_simulation -mode post-implementation -type
        ↪ functional')\
261     # run post simulation
262     try:
263         self.run(filename=r"post_simulation.tcl")
264     except:
265         # remove and restore drive substitutions
266         if os.name == 'nt':
267             if self.subst:
268                 try:
269                     subprocess.call(f'subst {drive} /d', shell=True)
270                 except:
271                     print(f'WARNING: Removing mapped drive:{drive} did not
                        ↪ work.')
272                 if self.old_subst:
273                     try:
274                         subprocess.call(f'subst {drive} {self.old_subst}',
                            ↪ shell=True)
275                     except:
276                         print(f'WARNING: Mapping of drive:{drive} to
                            ↪ network path: {self.old_subst} did not work.')
277         # then re-raise the original exception
278         raise
```

Listing A.3: Calling post-synthesis simulation in **anasymod**

Python

```
345 def simulate_post(self, tcl_script=None):
346     """
347     Run post-implementation simulation
348 
349     :param tcl_script: custom TCL file for running the simulation
350     """
351 
352     if not hasattr(self, self.act_fpga_target):
353         self._setup_targets(target=self.act_fpga_target,
        ↪  gen_structures=True)
354 
355     # Check if active target is an FPGA target
356     target = getattr(self, self.act_fpga_target)
357 
358     # create sim result folders
359     if not os.path.exists(os.path.dirname(target.cfg.vcd_path)):
360         mkdir_p(os.path.dirname(target.cfg.vcd_path))
361 
362     if not os.path.exists(os.path.dirname(target.result_path_raw)):
363         mkdir_p(os.path.dirname(target.result_path_raw))
364 
365     if tcl_script is None:
366         codegen = CodeGenerator()
367         codegen.writeln(f'set curr_wave [current_wave_config]')
368         codegen.writeln(f'if {{ [string length $curr_wave] == 0 }} {{')
369         codegen.writeln(f'  if {{ [llength [get_objects]] > 0}} {{')
370         codegen.writeln(f'    add_wave /')
371         codegen.writeln(f'    set_property needs_save false
        ↪  [current_wave_config]')
372         codegen.writeln(f'  }} else {{')
373         codegen.writeln(f'    send_msg_id Add_Wave-1 WARNING "No top level
        ↪  signals found. Simulator will start without a wave window."')
374         codegen.writeln(f'  }}')
375         codegen.writeln(f'}}')
376         codegen.writeln(f'run all')
377         tcl_script = os.path.join(target.prj_cfg.build_root,
        ↪  'post_sim.tcl')
378         codegen.write_to_file(tcl_script)
379 
380     # run the post-simulation
381     VivadoEmulation(target=target).post_simulate(tcl_script)
382 
383     statpro.statpro_update(statpro.FEATURES.anasymod_post_vivado)
```

Appendix B

anasymod Control Infrastructure

This chapter gives an overview of the control infrastructure generated by **anasymod**[1], as the documentation from the tool itself is lacking. Not all details of variable timestep management and clock generation are discussed; details of these can be found in Chapter 5.3 of [3]. Rather, this piece focuses on the implementation of the infrastructure and its generation in SystemVerilog and Python respectively. Where necessary, the **anasymod** Python library is referenced.

Appendix B.1 gives an overview of the functions of each module. Appendix B.2 gives a description of each signal, and the data it carries. The connections of modules and signals are shown in Figure B.1.

B.1 Modules

Many of the modules are generated from templates in Python, so called *structures*. This way the SystemVerilog modules can be modified depending on the config files `prj.yaml`, `clks.yaml` and `simctrl.yaml`. The function of each module is outlined below.

`clk_gen` is a wrapper for Xilinx IP that generates a clock with double the emulation clock frequency, and any other required *independent* clocks, from a master clock. Independent clocks are specified in the config file `clks.yaml`.

`gen_time_manager` calculates the minimum of all timestep requests, and communicates this value back as `emu_dt`. It also keeps track of `emu_time`, the absolute emulation time across varying timesteps.

`osc_model` is required for designs that do not generate any analog clocks. In that case, this module creates the timestep requests (`emu_dt_req`) required for the desired emulation frequency that is eventually tied to `` `CLK_MSDSL ``.

[1] **anasymod** by S. Herbst and G. Rutsch is available on GitHub (https://git.io/anasymod)

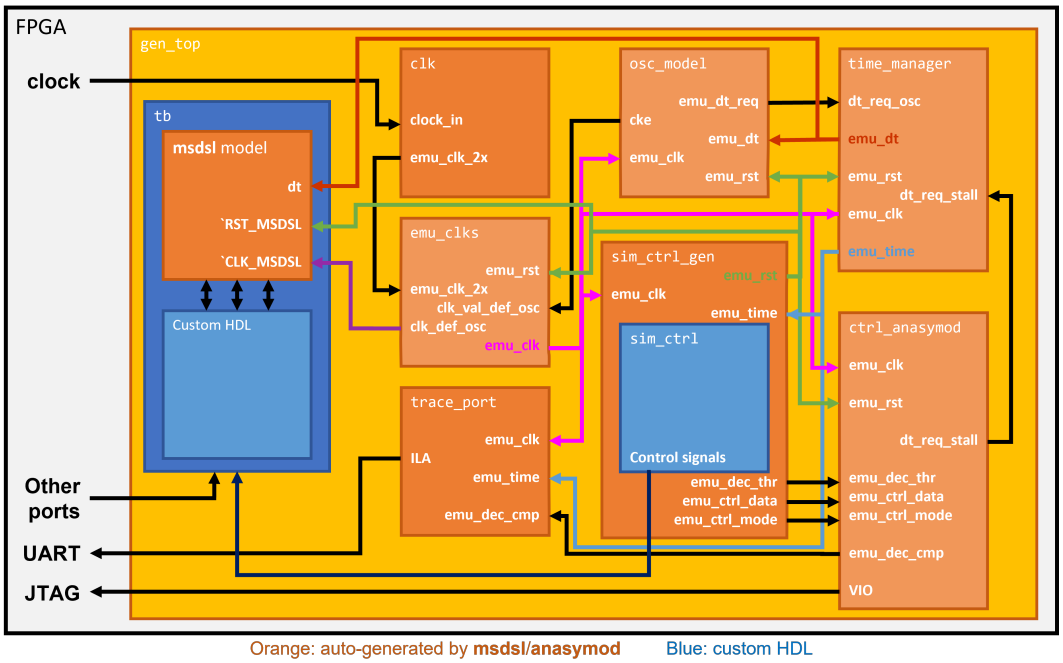


Figure B.1: **anasymod** control infrastructure

`gen_emu_clks`
: generates the actual clocks as requested by modules such as the oscillator. These clocks are synchronised to the master emulation clock `emu_clk_2x`. This synchronisation limits the time-resolution of any clock to that of the emulation clock, as the rising edges are aligned.

`sim_ctrl_gen`
: is a wrapper for the *Xilinx Virtual-IO* (VIO) IP required for emulation control from a host PC, as will be explained in the following section. The global reset signal `emu_rst`, accessed with the macro `` `RST_MSDSL `` is also generated here. During simulation, it is a wrapper for the for user-written module `sim_ctrl` for simulation control.

`trace_port_gen`
: is a wrapper for the *Xilinx Integrated Logic Analyzer* (ILA) IP used for monitoring signals in the design. Inputs to this module are signal probes specified in `simctrl.yaml`.

`ctrl_anasymod`
: is responsible for emulation control such as starting, pausing, and stopping emulation by influencing the time manager. It takes a number of signals from `trace_port` to switch between these modes.

`tb`
: is a user-written module and acts as a testbench for any **msdsl** and custom *hardware description language* (HDL) modules, essentially it is the *design under test* (DUT). All analog signals generated in the rest of the control infrastructure are inputs to this module.

`gen_top`
: is the top-level entity that instantiates all of these modules.

B.2 Signals

`emu_dt` The size of the timestep taken.

`emu_clk` The clock representing timestep changes.

`emu_time` Counts the emulation time in number of minimum timesteps.

`emu_rst` Reset signal.

`emu_clk_2x` Clock used to synchronise other clocks to. Twice the emulation clock frequency.

`clk_val_default_osc`
The un-synchronised clock generated by the oscillator.

`clk_default_osc`
The main emulation clock tied to `` `CLK_MSDSL ``, generated by the oscillator and synchronised to `emu_clk`.

`dt_req_osc` Timestep request from the oscillator.

`clock_in` Clock signal from the *field-programmable gate array* (FPGA).

`emu_dec_thr` Manage decimation ratio for capturing probe samples.

`emu_dec_cmp` Triggers sampling of the ILA based on `emu_dec_th`.

`emu_dt_stall`
Carries a signal telling the time manager to stall the emulator.

`emu_ctrl_mode`
The mode of the emulator. This signal is used to determine the state of `emu_dt_stall`:

0 Run emulator. Set `emu_dt_stall` to 0.

1 Stall emulator. Set `emu_dt_stall` to 1.

2 Stall emulator after a certain time.

3 Stall emulation when certain time is reached.

`emu_ctrl_data`
Carries data used by `emu_ctrl_mode`, such as the time.

Other ports Physical FPGA pins passed to the testbench.

control signals
Signals specified in `simctrl.yaml`, passed to the ILA.

B.3 Extensions

Two small extensions have been made to the control infrastructure. They are available in a fork of **anasymod**[2]. These extensions are not explained in further details, as they are rather simple. Both add options to `prj.yaml`, which add code in *structure* generation.

B.3.1 Write Signals to File

This extension adds code to write signals specified in `simctrl.yaml` to file. The *structure* for `gen_top` is extended with code to create and open files, and write the specified signals to file. An option `probe_to_file` is added that is configurable from `prj.yaml`.

B.3.2 Physical Ports

The second extension is for passing physical pins to and from the FPGA, through `gen_top`. The ports specified in `prj.yaml` are added as input and output ports to `gen_top` and passed to `tb`.

[2]The fork is available on GitHub https://github.com/nsulzer/anasymod

Appendix C

Code Listings

C.1 Generated SystemVerilog Descriptions

Listing C.1: Generated HDL description from **msdsl** DE

SystemVerilog

```
`timescale 1ns/1ps

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

`default_nettype none

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

`default_nettype wire
```

Listing C.2: Generated HDL description from **msdsl** TF

SystemVerilog

```
`timescale 1ns/1ps

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

`default_nettype none

module rc_tf #(
    `DECL_REAL(v_in),
    `DECL_REAL(v_out)
) (
    `INPUT_REAL(v_in),
    `OUTPUT_REAL(v_out)
);
    // Declaring internal variables.
    `MAKE_FORMAT_REAL(v_in_1, `RANGE_PARAM_REAL(v_in),
    ↪  `WIDTH_PARAM_REAL(v_in), `EXPONENT_PARAM_REAL(v_in));
    `MAKE_FORMAT_REAL(v_out_1, `RANGE_PARAM_REAL(v_out),
    ↪  `WIDTH_PARAM_REAL(v_out), `EXPONENT_PARAM_REAL(v_out));
    // Assign signal: v_in_1
    `DFF_INTO_REAL(v_in, v_in_1, `RST_MSDSL, `CLK_MSDSL, 1'b1, 0);
    // Assign signal: v_out_1
    `DFF_INTO_REAL(v_out, v_out_1, `RST_MSDSL, `CLK_MSDSL, 1'b1, 0);
    // Assign signal: v_out
    `MUL_CONST_REAL(0.09516258196404048, v_in_1, tmp0);
    `MUL_CONST_REAL(0.9048374180359597, v_out, tmp1);
    `ADD_REAL(tmp0, tmp1, tmp2);
    `DFF_INTO_REAL(tmp2, v_out, `RST_MSDSL, `CLK_MSDSL, 1'b1, 0);
endmodule

`default_nettype wire
```

Listing C.3: Generated HDL description from **msdsl** netlist

SystemVerilog

```
`timescale 1ns/1ps

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

`default_nettype none

module rc_nl #(
    `DECL_REAL(v_in),
    `DECL_REAL(v_out)
) (
    `INPUT_REAL(v_in),
    `OUTPUT_REAL(v_out)
);
    // Declaring internal variables.
    `MAKE_REAL(tmp_circ_2, `RANGE_PARAM_REAL(v_in));
    // Assign signal: tmp_circ_2
    `MUL_CONST_REAL(0.9048374180359596, tmp_circ_2, tmp0);
    `MUL_CONST_REAL(0.09516258196404037, v_in, tmp1);
    `ADD_REAL(tmp0, tmp1, tmp2);
    `DFF_INTO_REAL(tmp2, tmp_circ_2, `RST_MSDSL, `CLK_MSDSL, 1'b1, 0);
    // Assign signal: v_out
    `ASSIGN_REAL(tmp_circ_2, v_out);
endmodule

`default_nettype wire
```

Listing C.4: Generated HDL description from **msdsl** DE with switch

SystemVerilog

```
`timescale 1ns/1ps

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

`default_nettype none

module rc_de_sw #(
    `DECL_REAL(v_in),
    `DECL_REAL(v_out)
) (
    `INPUT_REAL(v_in),
    `OUTPUT_REAL(v_out),
    input wire logic sw
);
    // Assign signal: v_out
    `MAKE_SHORT_REAL(tmp1, 0.944862054881934);
    `MUL_REAL(tmp1, v_out, tmp0);
    always @(*) begin
        case (sw)
            0: tmp1 = `FROM_REAL(0.9355069850316178, tmp1);
            1: tmp1 = `FROM_REAL(0.8187307530779819, tmp1);
            default: tmp1 = 0;
        endcase
    end
    `MAKE_SHORT_REAL(tmp3, 0.18308193939123826);
    `MUL_REAL(tmp3, v_in, tmp2);
    always @(*) begin
        case (sw)
            0: tmp3 = `FROM_REAL(0.06449301496838222, tmp3);
            1: tmp3 = `FROM_REAL(0.18126924692201807, tmp3);
            default: tmp3 = 0;
        endcase
    end
    `ADD_REAL(tmp0, tmp2, tmp4);
    `DFF_INTO_REAL(tmp4, v_out, `RST_MSDSL, `CLK_MSDSL, 1'b1, 0);
endmodule

`default_nettype wire
```

Listing C.5: Generated HDL description from **msdsl** update equation with variable-timestep

SystemVerilog

```
`timescale 1ns/1ps

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

`default_nettype none

module rc_eq_pwl #(
    `DECL_REAL(v_in),
    `DECL_REAL(v_out),
    `DECL_REAL(dt)
) (
    `INPUT_REAL(v_in),
    `OUTPUT_REAL(v_out),
    `INPUT_REAL(dt)
);
    // Declaring internal variables.
    `MAKE_FORMAT_REAL(real_func_0_coeff_0_0, 1.01, 18, -16);
    `MAKE_FORMAT_REAL(real_func_0_coeff_1_0, 0.019573024820237348, 18, -22);
    `MAKE_REAL(real_func_0_prod_del_0_0, 1.01);
    `MAKE_FORMAT_REAL(v_in_1, `RANGE_PARAM_REAL(v_in), `WIDTH_PARAM_REAL(v_in), `EXPONENT_PARAM_REAL(v_in));
    `MAKE_FORMAT_REAL(v_out_1, `RANGE_PARAM_REAL(v_out), `WIDTH_PARAM_REAL(v_out), `EXPONENT_PARAM_REAL(v_out));
    // Assign signal: real_func_0_addr_real_0
    `MAKE_CONST_REAL(5.109999999999999, tmp0);
    `MAKE_REAL(real_func_0_addr_real_0, 511.0);
    `ASSIGN_REAL(tmp0, real_func_0_addr_real_0);
    // Assign signal: real_func_0_addr_sint_0
    `REAL_TO_INT(real_func_0_addr_real_0, 10, tmp1);
    logic signed tmp2;
    assign tmp2 = 0;
    logic signed [9:0] tmp3;
    assign tmp3 = ((tmp1 > tmp2) ? tmp1 : tmp2);
    logic signed [9:0] tmp4;
    assign tmp4 = 511;
    logic signed [9:0] tmp5;
    assign tmp5 = ((tmp3 < tmp4) ? tmp3 : tmp4);
    logic signed [9:0] real_func_0_addr_sint_0;
    assign real_func_0_addr_sint_0 = tmp5;
    // Assign signal: real_func_0_addr_uint_0
    logic [8:0] tmp6;
    assign tmp6 = real_func_0_addr_sint_0[8:0]; // SInt -> UInt
    logic [8:0] real_func_0_addr_uint_0;
    assign real_func_0_addr_uint_0 = tmp6;
    // Assign signal: real_func_0_addr_frac_0
    logic signed tmp8;
    assign tmp8 = -1;
    logic signed [9:0] tmp9;
```

```
48      assign tmp9 = (real_func_0_addr_sint_0*tmp8);
49      `INT_TO_REAL(tmp9, 10, tmp7);
50      `ADD_REAL(real_func_0_addr_real_0, tmp7, tmp10);
51      `MAKE_REAL(real_func_0_addr_frac_0, 1.01);
52      `ASSIGN_REAL(tmp10, real_func_0_addr_frac_0);
53      // Assign signal: real_func_0_coeff_0_0
54      `SYNC_ROM_INTO_REAL(real_func_0_addr_uint_0, real_func_0_coeff_0_0,
        ↪  `CLK_MSDSL, 1'b1, 9, 18, "path/to/real_func_0_lut_0_exp_-16.mem",
        ↪  -16);
55      // Assign signal: real_func_0_coeff_1_0
56      `SYNC_ROM_INTO_REAL(real_func_0_addr_uint_0, real_func_0_coeff_1_0,
        ↪  `CLK_MSDSL, 1'b1, 9, 18, "path/to/real_func_0_lut_1_exp_-22.mem",
        ↪  -22);
57      // Assign signal: real_func_0_prod_imm_0_0
58      `MAKE_REAL(real_func_0_prod_imm_0_0, 1.01);
59      `ASSIGN_REAL(real_func_0_addr_frac_0, real_func_0_prod_imm_0_0);
60      // Assign signal: real_func_0_prod_del_0_0
61      `DFF_INTO_REAL(real_func_0_prod_imm_0_0, real_func_0_prod_del_0_0,
        ↪  `RST_MSDSL, `CLK_MSDSL, 1'b1, 0);
62      // Assign signal: a
63      `MUL_REAL(real_func_0_coeff_1_0, real_func_0_prod_del_0_0, tmp11);
64      `ADD_REAL(real_func_0_coeff_0_0, tmp11, tmp12);
65      `MAKE_REAL(a, 1.0297687550684398);
66      `ASSIGN_REAL(tmp12, a);
67      // Assign signal: v_in_1
68      `DFF_INTO_REAL(v_in, v_in_1, `RST_MSDSL, `CLK_MSDSL, 1'b1, 0);
69      // Assign signal: v_out_1
70      `DFF_INTO_REAL(v_out, v_out_1, `RST_MSDSL, `CLK_MSDSL, 1'b1, 0);
71      // Assign signal: v_out
72      `MUL_REAL(a, v_out_1, tmp13);
73      `NEGATE_REAL(a, tmp14);
74      `MAKE_CONST_REAL(1.0, tmp15);
75      `ADD_REAL(tmp14, tmp15, tmp16);
76      `MUL_REAL(tmp16, v_in_1, tmp17);
77      `ADD_REAL(tmp13, tmp17, tmp18);
78      `ASSIGN_REAL(tmp18, v_out);
79  endmodule
80
81  `default_nettype wire
```

C.2 Python msdsl Models

Listing C.6: Modelling a DE with a switch in **msdsl**

Python

```python
from pathlib import Path
from argparse import ArgumentParser
from msdsl import MixedSignalModel, VerilogGenerator, Deriv, eqn_case


def make_model(name, build_dir, dt=1e6, r=1e3, c=1e-9):
    m = MixedSignalModel(name, dt=dt, build_dir=build_dir)
    m.add_analog_input('v_in')
    m.add_analog_output('v_out')
    m.add_digital_input('sw')

    # apply dynamics
    r1, r2 = r+0.5e3, r-0.5e3
    rsw = eqn_case([1/r1, 1/r2], [m.sw])
    m.add_eqn_sys([c*Deriv(m.v_out) == (m.v_in-m.v_out)*rsw])
    return m

def main(name='rc_de_sw'):
    print('Running model generator...')

    # parse command line arguments
    parser = ArgumentParser()
    parser.add_argument('-o', '--output', type=str,
    ↪  default=name+'/build/models/default/main')
    parser.add_argument('--dt', type=float, default=1e-6)
    args = parser.parse_args()
    build_dir = Path(args.output).resolve()

    r, c = 1e3, 1e-9
    m = make_model(name, build_dir, args.dt, r, c)

    # determine the output filename
    filename = build_dir / f'{m.module_name}.sv'
    print(f'Model will be written to: {filename}')

    # generate the model
    m.compile_to_file(VerilogGenerator(), filename)

if __name__ == '__main__':
    main()
```

Listing C.7: **msdsl** model of first-order ΣΔ modulator

Python

```python
from pathlib import Path
from argparse import ArgumentParser
from msdsl import *

def makeModel(args):
    m = MixedSignalModel('sd_model', dt=args.dt)

    in_gain, fb_gain, freq_gain = 0.45, 1, 6.25e6
    G = [[freq_gain],[1, 0]]                        # TF of loop filter

    fs = m.add_digital_input('fs')                  # 6.25MHz sample clock
    x = m.add_analog_input('x')                     # analog input
    y = m.add_digital_output('y')                   # digital output
    e = m.add_analog_state('e', 4, init=1)          # node after summation
    a = m.add_analog_state('a', 4, init=0)          # node after integrator
    d = m.add_analog_state('d', 1, init=-1)         # node after DAC

    m.set_this_cycle(e, x*in_gain-d*fb_gain)        # adder
    m.set_tf(e, a, G)                               # loop filter
    m.set_next_cycle(y, a>0, ce=fs)                 # quantiser
    m.set_next_cycle(d, y*2-1.0)                    # DAC
    return m

def main():
    print('Running model generator...')

    # parse command line arguments
    parser = ArgumentParser()
    parser.add_argument('-o', '--output', type=str,
    ↪  default='build/o1b1/models/default/main')
    parser.add_argument('--dt', type=float, default=50e-6)
    args = parser.parse_args()

    # create the model
    m = makeModel(args)

    # determine the output filename
    filename = Path(args.output).resolve() / f'{m.module_name}.sv'
    print(f'Model will be written to: {filename}')

    # generate the model
    m.compile_to_file(VerilogGenerator(), filename)

if __name__ == '__main__':
    main()
```

Listing C.8: **msdsl** model of second-order ΣΔ modulator

Python

```python
from pathlib import Path
from argparse import ArgumentParser
from msdsl import *

def makeModel(args):
    m = MixedSignalModel('sd_model', dt=args.dt)

    v_of = 3.3/2                                    # bias voltage
    in_gain  , fb_gain  , freq_gain  = 0.2826 , 0.2824 , 6.25e6
    in_gain2 , fb_gain2 , freq_gain2 = 0.6520 , 0.5677 , 6.25e6
    G  = [[freq_gain ],[1, 0]]                      # TF of integrator 1
    G2 = [[freq_gain2],[1, 0]]                      # TF of integrator 2

    fs = m.add_digital_input('fs')                  # 6.25MHz sample clock
    x  = m.add_analog_input('x')                    # analog input
    y  = m.add_digital_output('y')                  # digital output
    xs = m.add_analog_state('xs', 4, init=0)        # scaled input
    e  = m.add_analog_state('e' , 4, init=1)        # node after summation 1
    a  = m.add_analog_state('a' , 4, init=0)        # node after integrator 1
    e2 = m.add_analog_state('e2', 4, init=1)        # node after summation 2
    a2 = m.add_analog_state('a2', 4, init=0)        # node after integrator 2
    d  = m.add_analog_state('d' , 4, init=-1)       # node after DAC

    m.set_this_cycle(xs, 0.65*v_of*x + v_of)        # scale input
    m.set_this_cycle(e, xs*in_gain-d*fb_gain)       # summation node 1
    m.set_tf(e, a, G)                               # integrator 1
    m.set_this_cycle(e2 , a*in_gain2-d*fb_gain2)    # summation node 2
    m.set_tf(e2, a2, G2)                            # integrator 2
    m.set_next_cycle(y, a2>v_of, ce=fs)             # quantiser
    m.set_next_cycle(d, (y*3.3))                    # DAC
    return m

def main():
    print('Running model generator...')

    # parse command line arguments
    parser = ArgumentParser()
    parser.add_argument('-o', '--output', type=str,
    ↪  default='build/o2b1/models/default/main')
    parser.add_argument('--dt', type=float, default=50e-6)
    args = parser.parse_args()

    # create the model
    m = makeModel(args)

    # determine the output filename
    filename = Path(args.output).resolve() / f'{m.module_name}.sv'
    print(f'Model will be written to: {filename}')

    # generate the model
```

```
50     m.compile_to_file(VerilogGenerator(), filename)
51
52 if __name__ == '__main__':
53     main()
```

C.3 Extension of msdsl

Listing C.9: Function for coefficient sweeping

Python

```
1  def make_coef_sweep(self, name='param', ctrl='input', form='lin',
   ↪  range:tuple=[0, 1], numel=512, **kwargs):
2      """
3      :param name:    Name of sweeping signal
4      :param ctrl:    Name of control signal or handle to AnalogInput
5      :param form:    Sweep form. Can be 'lin', 'log10', or ...
6      :param range:   Range of the sweep
7      :param numel:   Number of values in the sweep. Must be power of 2!
8      :return:        Handle to signal whose value can be swept
9      """
10     # handle control input
11     if isinstance(ctrl,str):
12         input = self.add_analog_input(ctrl)
13     elif isinstance(ctrl, AnalogInput):
14         input = ctrl
15     else:
16         raise Exception(f'Invalid control input.')
17     # handle form
18     if form == 'lin':
19         func = lambda input: input # func is a function of input, that
           ↪  returns input. f(x) = x
20         domain = range
21     elif form == 'log10':
22         func = lambda input: np.log10(input)    # func is a function of
           ↪  input that returns log10(input). f(x) = log10(x);
23         domain = range
24     else:
25         raise Exception(f'Invalid sweep form given: {form}')
26     # make sweep signal and return
27     f = self.make_function(name=name, func=func, domain=domain,
       ↪  numel=numel, order= 0 **kwargs)
28     param = self.set_from_sync_func(signal=name, func=f, in_=input)
29
30     return param
```