Universiteit Leiden
The Netherlands

Opleiding Informatica

Developing a Backend Compiler for PDE Simulation on the Anadigm AN231E04 Hybrid Computer

Niels van Schagen

Supervisors:
Dr. H. Basold & Prof.dr. R. V. van Nieuwpoort

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

01/07/2025

Abstract

Reconfigurable analog-digital hybrid computers offer new opportunities for efficient simulation of partial differential equations (PDEs). Existing compilers target problem-specific hybrid computers with specialized modules. General purpose hybrid computers such as the Anadigm AN231E04 dpASP tend to be more affordable and accessible, making them promising targets for PDE simulation. This thesis implements a backend compiler that translates a new configuration file format to the configuration data used by the target device. It builds on previous work that developed a toolchain for hybrid PDE simulation, which currently ends in an intermediate representation. To achieve the goal of backend compilation, the proprietary and undocumented configuration format of the AN231E04 has to be reverse engineered. Experimental results demonstrate that simple PDEs can be accurately modeled on the analog hardware. Due to system and scope limitations, the current compiler is only able to model small, linear PDEs. This work lays the foundation for future research into PDE simulation on the targeted hybrid computer.

Contents

1 Introduction

With an increased focus on energy efficient and economic computing, analog computers have found new interest during the last decade [Cra16, Has20, HGS+17]. One active area of research focuses on the simulation of partial differential equations (PDEs) on analog-digital hybrid computers [Ulm20, GHM+16, HGS+17]. This field has made new advancements since the advent of the analog-digital hybrid computer. The simulation of PDEs is traditionally resource intensive due to the need for discretization on digital systems, which makes hybrid computer simulation very suitable, combining the inherent parallel and continuous nature of the analog computer with the flexibility and accuracy of the digital computer.

While successful compilers have already been built [Ulm20, GHM+16], these have been tailored to purpose-built hybrid computers with dedicated components for operations such as summation or integration circuits. However, less research has focused on compilation to general purpose hybrid computers, which are typically more affordable and widely accessible. This thesis explores one such device: the Anadigm AN231E04 dynamically programmable analog signal processor.

1.1 Problem Statement & Research Questions

In two previous theses, the foundations of this compiler have been built [VDE23, Dui24]. The current toolchain ends with a compiler to an intermediate representation of the configuration. The final step of compiling to the raw configuration data used by the hardware remains unaddressed. This data format is undocumented, making direct compilation non-trivial.

The project is limited in scope to the simulation of simple, small PDEs using basic linear operations, excluding more complex functions such as multiplication or trigonometric functions. The main research question is:

> How can the digital microchip of the AN231E04 dpASP hybrid computer be programmed to reconfigure the analog blocks to simulate simple, linear PDEs?

To answer this research question, the project is divided into two parts. The first part will focus on reverse engineering the configuration data of the device, an array of bytes which dynamically configures the analog chip. Its main focus is on gaining an understanding and documenting the layout of the data configuring the modules used to simulate PDEs. This leads to the first sub-question:

> What is the format of the undocumented reconfiguration data of the AN231E04?

The second part of the research involves the development of a compiler built around the findings of the first part. It is able to compile to the configuration data format of the AN231E04. A new textual configuration file format is introduced, which is used to specify configurations for the compiler. The development of the compiler answers the second research sub-question:

> How can a compiler generate C/C++ code that configures the AN231E04 to simulate simple, linear PDEs?

1.2 Thesis overview

This bachelor's thesis aims to investigate the configuration data format of the AN231E04 and to build a compiler around the findings to compile PDE simulations to the AN231E04 format. The study was conducted at the Leiden Institute of Advanced Computer Science and was supervised by Dr. Henning Basold.

The thesis starts with background information on analog and hybrid computing and on the simulation of ODEs and PDEs, as well as an explanation of the two previous compilers in Section 2. Then, the process of reverse engineering the chip's configuration data format is discussed in Section 3. The findings were used to implement a compiler to the target format, which is described in detail in Section 4. The results of experiments involving PDE simulations on the AN231E04 are discussed in Section 5. Relevant work on the analog simulation of PDEs and general implementations of FPAAs is described in Section 6. Finally, the conclusions and possible topics for further research are discussed in Section 7.

2 Background

2.1 Analog-Digital Hybrid Computing

Analog computing differs from digital computing in that it represents values using continuous physical quantities rather than discrete symbols. In digital computing, values consist of a fixed number of digits or bits, and there is no direct relationship between the value and its physical representation. Analog computing, by contrast, represents values by continuous physical quantities. Traditionally, these quantities follow the same mathematical laws as the system that is being modeled. An example of this is modeling the flow of fluids through pipes using electronic voltages [Mac07].

For this thesis, the focus will be on electronic analog computing, where the values are represented by voltages. These can be manipulated by certain operations, which are typically abstracted to modules that a circuit designer can use and connect. Specifically, the Anadigm AN231E04 dpASP is used. This is a reprogrammable, general-purpose analog computer. Although analog computing is typically continuous, the AN231E04 uses high-frequency synchronizing clocks to implement its operations. This facilitates the reconfigurability of the device and the implementation of certain operations which require synchronization.

Examples of operations that are relevant to this thesis are inverting sum and gain stages, integrators, and sample and hold stages. These components and their implementation in the Anadigm AN231E04 are described in the next sections.

2.1.1 Modules for the AN231E04

The inverting sum component has two inputs in its default configuration. The output is the result of adding its inputs and negating the result. The input values can be scaled using gain values determined by the user. The transfer function of the component is given by Equation 1 [Ana25].

$$V_{out} = -(G_1 V_1 + G_2 V_2) \tag{1}$$

Here, V_i indicates an input voltage which is scaled by gain $G_i \in [0.01, 1]$. Optionally, a third input can be enabled, which is scaled and added like the other two inputs.

The inverting gain stage can be considered a simpler version of the inverting sum stage, consisting of only a single input which is scaled and inverted to form its output. A variation of an inverting gain stage is the gain stage with switchable inputs, which selects one of two scaled inputs using a control signal. This variant is also able to control the sign of the output.

Central to the simulation of differential equations is the integrator. Its output is equal to the change of the output voltage, as measured over a single clock period. Its transfer function is given by Equation 2 [Ana25].

$$\frac{\Delta V_{out}}{\Delta t} = \pm K V_{in} \tag{2}$$

The transfer function can be rewritten to Equation 3. This shows how the change of the output

voltage is calculated through the input voltage.

$$\Delta V_{out} = \Delta t \cdot \pm K V_{in} \tag{3}$$

The integrator operates over discrete time intervals defined by the clock, where the duration of a single clock period is given by Δt. It can be considered an application of the Euler method for solving differential equations. The output voltage V_{out} starts at zero and is updated at each time step by adding a change proportional to the current input voltage V_{in}, scaled by the constant K and the clock period Δt. The smaller the clock period, the closer the module approximates continuous integration. Over time, this process accumulates changes to approximate the integration of the input signal. The sign of the constant K can be configured, allowing for both positive and negative changes to V_{out}.

A variant of the integrator includes multiple inputs, which are summed before integrating. This is useful to model ODEs efficiently as this is a common operation.

In some cases, the output of a component is only valid during one of two clock phases. The sample and hold component can be used if another component requires a signal to be valid for a full phase. The module samples its input voltage during one of the clock phases and holds it during the other phase. This can be used in combination with the integrator or the gain stage with switchable inputs, both of which give different outputs on the two clock phases.

2.1.2 Module implementations

The AN231E04 is a Field Programmable Analog Array or FPAA. This refers to an analog chip which can be reprogrammed "in the field". In this case, the chip can be reprogrammed dynamically at runtime. The chip implements the modules as Configurable Analog Modules or CAMs. A CAM can be considered an abstraction over the raw resources used by each module and the configuration of these resources. Available resources include capacitors, op-amps, comparators, and switches to connect them. These will be explained in more detail in the remainder of this section.

As an example, the circuit configuring an integrator will be considered. Through this example, each part of the AN231E04 that is relevant to this thesis will be described. The circuit diagram is shown in Figure 1. This circuit consists of three types of components: capacitors indicated by C_{in} and C_{int}, switches indicated by Sx and one operational amplifier (*op-amp*).

The first capacitor, C_{in}, is surrounded by two switches $S1$ and $S2$. In a traditional circuit for an op-amp integrator, a resistor is used to connect the input to an input terminal of the op-amp. In this implementation, the resistor is replaced by a capacitor with two switches, forming a switched capacitor (SC). A switched capacitor is more reliable and economic to manufacture than a resistor. It can be used to simulate the continuous transfer of a real resistor using two switches on a high frequency clock. The switches are connected to the signal path on opposite clock phases. Equation 4 shows the relationship between the capacitance C_s, the switching frequency f, and the equivalent resistor $R_{equivalent}$.

$$R_{equivalent} = \frac{1}{C_s \cdot f} \tag{4}$$

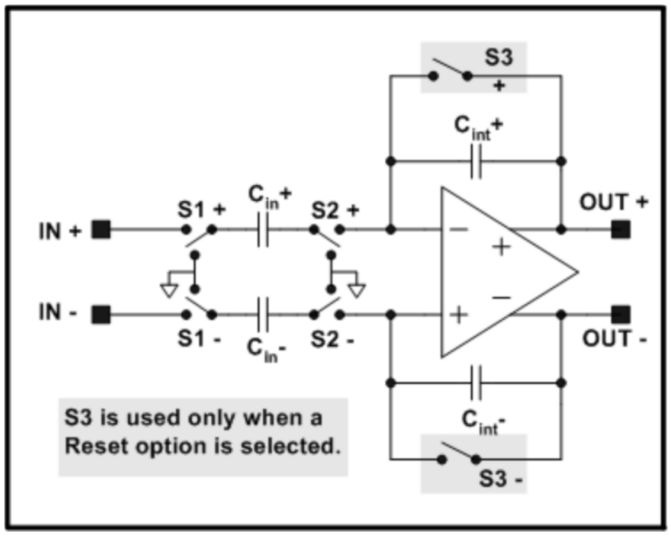

Figure 1: Circuit diagram for an Integrator [Ana25]

The switching frequency refers to the frequency of the clock to which the switches of the SC are connected. In the AN231E04, each capacitor is likely chosen from an array of capacitors, each with a different capacitance. This allows the system to dynamically configure equivalent resistance.

The switched capacitor is connected to the input terminal of the op-amp. The op-amp is the core component of most operations in analog computing. It amplifies the voltage difference between its two inputs with a very high gain. Due to this high gain and the presence of a negative feedback loop, the op-amp continuously adjusts its outputs to keep the input voltages nearly equal. This behavior is known as a virtual short. It enables the op-amp to perform operations such as integration and addition when combined with simple components like capacitors and switches in a feedback loop.

In the integrator CAM, the second capacitor, C_{int}, connects the output of the op-amp back to its inverting input, forming a negative feedback loop. This setup causes the output to change based on the input voltage over time, which means it performs integration. The circuit slowly builds up charge, so the output voltage increases or decreases smoothly depending on the input signal.

The final switch in the circuit, $S3$, is placed in parallel to the feedback capacitor C_{int} and is used to implement the optional reset functionality. When the integrator receives a reset signal, this switch closes, short-circuiting the feedback path and discharging the capacitor. Due to the op-amp's high gain in combination with the feedback loop, the output is forced to the reset value. The reset switch is only active if the reset CAM option is enabled.

The reset control signal is controlled by the comparator. This component is separate from the main circuit and not pictured in the diagram. It can be used to compare two input signals. The output signal is positive if the first input is greater than the second input and negative otherwise. The sign of the output can be switched depending on the configuration. Instead of a second input, signal ground may be used, which replaces the second input by a constant value of zero.

Each previously described part of the circuit is present twice and mirrored across the horizontal axis around the op-amp. This is because the AN231E04 uses differential signals for representing

values. The signs next to each input, output, capacitor and switch indicate one of two parts of this signal. The actual value being calculated is the difference between these two parts. The main advantage of differential signals concerns the accuracy of the computation, a general problem for analog computing. This type of signal rejects common-mode noise, which refers to noise that interferes with the circuitry [Sun05]. As both signals are affected equally and the output value is calculated by subtracting the two signals from each other, this noise is also subtracted, meaning that the interference is rejected.

The described integrator circuit is contained in a single CAM. On the chip, CAMs are implemented as part of a Configurable Analog Block (CAB). One CAB consists of the following resources:

- 8 switched capacitors
- 2 op-amps
- 1 comparator

Each CAB can consist of multiple CAMs. Consider the integrator without the reset option enabled, which does not use the comparator. The circuit diagram shows that the CAM uses two capacitors and one op-amp. This means that a single CAB can implement up to two basic integrator circuits, which leaves four capacitors and the comparator unused.

Each capacitor in the system is present twice on the physical hardware. This is necessary to account for the differential signal, requiring a capacitor on the positive and negative signal path.

The device likely uses binary weighted programmable capacitor arrays (PCAs) to implement its capacitors with a configurable capacitance. This is not mentioned in Anadigm's documentation, but it is implied by the configuration and supported by the usage in similar devices [LH98]. This would mean that every capacitor is implemented through an array of eight capacitors. Each of these capacitors has a capacitance of a unit capacitance multiplied by a power of two $(C, 2C, 4C, ...)$. By controlling which of these capacitors in connected, a large number of capacitor values can be realized.

Each CAB also has access to a Successive Approximation Register or SAR. This component can be used to discretize an analog signal to an 8-bit digital signal. As the relevant CAMs do not use the SAR, it is not used for this thesis. The digital signal can be used to index the chip's lookup table (LUT). The LUT allows for the implementation of non-linear functions and can be used to dynamically set the gain value of a specialized gain stage. These features are also not used in this thesis.

The chip can interact with external signals through IO cells. It has multiple types of specific IO cells, but for this thesis, only the first four IO cells will be considered in the bypass input and bypass output modes. This means that the input or output is unbuffered, no filtering is applied and the chip has direct access to the given signal. The user is responsible for keeping the signal in the limits of the device. The differential voltage should be kept between 0 volt and 3 volt and centered about the Voltage Mid-Rail (VMR) of 1.5 V [Oki25b].

The chip's dynamic reconfigurability is facilitated through the use of its volatile configuration memory. The chip contains three regions of volatile static random-access memory (SRAM). When a

new configuration is send to the device, it is initially written to its first region, the Shadow SRAM. This means that the actual configuration in its second region, the Configuration SRAM, is left unaltered during the transfer. Once the transfer is complete, the Configuration SRAM can be updated in a single clock tick. The third region refers to the SRAM region used to implement the lookup table.

2.1.3 AnadigmDesigner2

Anadigm supplies the AnadigmDesigner2 tool to configure the analog chip [Ana25]. This electronic computer-aided design (ECAD) software allows the developer to drag and drop modules in a chip interface and to configure various options per module, such as the gains of the inputs.

There are two important methods for loading the design to the chip. One method directly downloads the design to the chip, which then begins to simulate the design. The other method builds a Visual Studio project. This project can be used to reconfigure the modules dynamically through a number of standard functions defined per module type. These functions are only able to edit the basic attributes of the module, such as the gain. More importantly, however, this project also contains a file with the complete initial, primary configuration of the chip, represented as an array of byte values. The configuration array is the target of the compiler introfuced in this thesis, and this feature will be utilized to reverse engineer the configuration data by comparing the output of different designs.

2.2 PDE and ODE Simulation

The main focus of this thesis is to simulate partial differential equations using the AN231E04. A partial differential equation or PDE is a type of differential equation that depends on multiple variables [BD01]. This in contrast to an ODE or ordinary differential equation, which depends on a single variable. A differential equation relates an unknown function to its derivatives. These equations are relevant when modeling dynamic systems in physics, biology, and engineering. Examples include temperature diffusion, chemical reactions, electrical circuits, and population dynamics [Zil18].

Following is a simple example of a PDE and its conversion to a system of ODEs. It has been taken from the thesis of Van Der Ende [VDE23], whose contributions to the compiler will be described in Section 2.3.

The one-dimensional heat equation is as follows:

$$\delta_t u = \delta_{xx} u \tag{5}$$

In this equation, the function $u(x, t)$ represents the temperature at position x and time t. The partial derivative of u with respect to time is denoted by $\delta_t u$, and $\delta_{xx} u$ is the second partial derivative of u with respect to space. The equation models the diffusion of heat along a rod over time.

To simulate this PDE using an analog circuit, the spatial variable in the equation needs to be discretized, keeping the time variable continuous. This segments the continuous space into a number of grid points. The derivatives are then approximated using the values at neighboring points. Consider the transformation to a system of ODEs with five grid points, where the outer two points

are constant boundary values set to 0. This transforms the PDE into a system of ODEs:

$$\dot{u}_1 = \frac{1}{h^2}(0 - 2u_1 + u_2)$$
$$\dot{u}_2 = \frac{1}{h^2}(u_1 - 2u_2 + u_3)$$
$$\dot{u}_3 = \frac{1}{h^2}(u_2 - 2u_3 + 0)$$

The resulting system can be simulated on an analog computer, as modules for integration and the other operations are available on the system. This particular system can be built using three summing integrators. Each integrator combines the output of the two neighboring points, represented by the other integrators, with its own output prior to the integration. For the first and last point, one of the neighboring points is the boundary value. This can be represented by a connection to ground or optionally through an input node, allowing for the simulation of external influences on the modeled system. Finally, the output of each point $\dot{u}_i$ can be observed through a connection to an output IO cell.

2.3 PDE to ODE Compiler

The mathematical foundations of this thesis were developed by Van Den Ende [VDE23]. In his thesis, various theoretical parts of simulating PDEs on analog computers are studied. The main contribution of his thesis is the conversion from a PDE to a system of ODEs. This is achieved by first reducing a system of PDEs to a system of PDEs in standard form, as in equation 6. A broad class of PDEs was defined that can be reduced to this standard form. While not all PDEs are reducible, most PDEs used in practice can be reduced. The system of PDEs is then converted to a system of ODEs.

$$\frac{\delta u_i}{\delta t} = F_i(t, x, \frac{\delta^{\alpha_1} u_{j_1}}{\delta x^{\alpha_1}}, ..., \frac{\delta^{\alpha_n} u_{j_n}}{\delta x^{\alpha_n}}) \tag{6}$$

The resulting system of ODEs can be very large. This is a problem, as analog computers are resource constrained and only have a limited set of components which can be used at a time. This means that the system of ODEs needs to be split up into different groups of grid cells. This introduces a new problem, as dependencies between equations exist, as was shown through the heat equation in Section 2.2. To solve this, an iterative method of solving the system was introduced. The groups are simulated on the analog computer one at a time, and values from other iterations are loaded from memory.

Another contribution of his thesis is an algorithm to optimize the division of the system of ODEs into groups of grid cells. Loading and storing values to the digital microchip of the hybrid computer takes time and storage, and some accuracy is lost during the conversion from analog to digital. This means that the number of values that need to be saved between iterations needs to be minimized. For this, several heuristics have been considered, including a randomized algorithm, a spread heuristic, and an optimal hyperrectangle heuristic. The last of these heuristics was found to yield the best results and has been implemented in the PTOC tool.

The PTOC tool developed for his thesis is capable of converting a system of PDEs in standard form to a system of ODEs which have been grouped using the hyperrectangle heuristic. The input system is described using various options, including the names of the dimensions, the PDE domain, the equations themselves, the boundary values, and the initial conditions of the system. In particular, an interval for each variable is also given, which is necessary for an analog computer to determine the scaling of the variables. The output of the tool is an abstract description of the system of ODEs, including equations, intervals, and definitions of which variables are "emitted", referring to variables that can be used by other systems.

2.4 ODE Compiler

The thesis by Duivenvoorde [Dui24] built on top of the thesis by Van Den Ende and introduces a compiler that reads the output from the PTOC tool as its input and generates an intermediate representation describing the FPAA configuration [Dui24]. The choice was made to opt for an intermediate representation rather than compiling to a specific architecture because the format of the reconfiguration data of the AN231E04 was undocumented and, therefore, infeasible to compile to.

The ODE-Compiler tool has two main contributions compared to the previous ODE format. The tool performs the necessary scaling to realize the computation on the analog computer while keeping the variables in the voltage range of the target computer. The thesis includes proofs of correctness of these scaling equations. Another aspect that was considered in this thesis is the reconfiguration time. As the computer requires some time to reconfigure and stabilize a new design, it is beneficial to minimize the amount of reconfiguration that is required. In other words, keeping certain operations or modules locked to a configurable block may result in a shorter reconfiguration time. This was achieved by pre-processing the configuration to minimize reconfiguration overhead.

Additionally, the thesis mentions various ways to mitigate the accuracy loss inherent to analog computing. Different types of filters are suggested, including the amplifier filter which can be used on inputs of the AN231E04, and bilinear and biquadratic filters which are included in the chip's design software.

3 Reverse Engineering

The first step in the project is to reverse engineer the necessary parts of the configuration data of the device. This is an essential step, as the configuration data is currently not documented. In order to write a compiler with the data as its target, a detailed understanding of the format needs to be developed first.

By generating designs using the AnadigmDesigner2 tool and comparing their primary configuration arrays, it is possible to map different bytes to different components of the system, thereby gaining an understanding of how the system is configured and how this can be achieved through a tool separate from AnadigmDesigner2.

Bank Address	Configuration Purpose
00	Clock Dividers & Power
01	LUT & I/O Control
02	I/O Control
03	CAB 1 Bank A
04	CAB 1 Bank B
05	CAB 2 Bank A
06	CAB 2 Bank B
07	CAB 3 Bank A
08	CAB 3 Bank B
09	CAB 4 Bank A
0A	CAB 4 Bank B

Table 1: The Shadow SRAM banks and their contents [Oki25b]

3.1 Configuration Data

The configuration is stored on the chip in its Shadow SRAM. It consists of eleven banks of 32 bytes. Each bank is responsible for updating a specific part of the system. The purpose of each bank is described in Table 1. After the shadowed configuration is loaded to the device, the first three banks are stored in the chip's auxiliary RAM, while the other banks are stored in their respective CAB banks.

The banks are first initialized through the primary configuration array, which consists of a header and an arbitrary number of data blocks, each with their own header. Its format is documented in the user manual [Oki25a] and will only briefly be described. The reconfiguration of the chip follows almost the same format, as the array only updates the relevant parts of the chip's Shadow SRAM. When reset, every byte of the chip's Shadow SRAM is initialized to zero, meaning that the primary configuration only needs to update the non-zero bytes.

The header of the configuration is important for synchronization, device matching and loading data to the appropriate chip if multiple chips are connected together. This header can be kept constant for the purposes of this thesis, as only a single chip is used and the device ID, a 4-byte string identifying the device, is always constant for the AN231E04.

After the header, the data blocks follow. These are used to update specific sections of the Shadow SRAM, which are specified by their bank address, byte address and size. A data section may cross bank boundaries, in which case the next banks will also be configured.

Due to the fact that the data sections target specific sections in the chip's Shadow SRAM and can cross bank boundaries, this format is not suitable to compare directly for reverse engineering. For this reason, a Python tool is created which reads the array from a Visual Studio project and uses it to update a configuration matrix, in the same way the AN231E04 would update its Shadow SRAM. This matrix is then printed in a clear format. This simplifies comparison between different designs.

3.2 Reverse Engineering Strategy

The final target of the reverse engineering process is to build the backend compiler from the results, so only data necessary for compilation is considered. This is done incrementally. When the compiler requires data for a new component, this component is first reverse engineered and then added to the compiler.

During the reverse engineering, a choice has to be made to define the extent to which the configuration data is to be reverse engineered. The most basic option was to only reverse engineer the variable parts in the modules themselves. Most of a module's configuration remains static, but other parts can vary, which includes the bytes encoding the gains and routing connections of a module. In this case, the module layouts can be used as "templates" in the compiler, where variables are substituted during compilation. While this might seem like the easiest option, it presents several challenges. As the template is simply copied over with no real understanding of the behavior of the circuit, it is harder to determine in what cases exceptions may occur, which makes the compiler less robust. In addition, this complicates loading multiple modules in a single CAB. When two modules are combined in a CAB, they cannot occupy the same space and resources, so the configuration of the "secondary" module has to be adapted. In this case, many templates would be required for loading modules in various different positions, which is impractical and error-prone.

As a next step, the set up of individual components can be considered. The focus here is on the switches that control the routing between components of the module. After this process, the set up of a module will be fully understood, including which parts of the configuration data control which parts of the module's circuit. This is the approach that is chosen for the majority of this project. It solves the problem of combining modules in a single CAB, as the module can be decompiled to a configuration of connected capacitors, op-amps and comparators, where the components themselves stay effectively anonymous and can be assigned to any physical component on the hardware. This allows for a descriptive and easily extendable set up of modules in the compiler.

One problem remaining with this approach is that many of the values themselves stay abstract. For example, one discovery that is made is that the byte `0x50` controls the routing of an IO-cell to a CAB, but the significance of the value itself remains opaque. It is likely that many of the values control multiplexers or contain bitmaps. Discovering this would enhance the robustness and extensibility of the compiler, but this would require significantly more work, especially considering the limited amount of modules that are available from AnadigmDesigner2. This is considered out of scope for the purposes of this thesis.

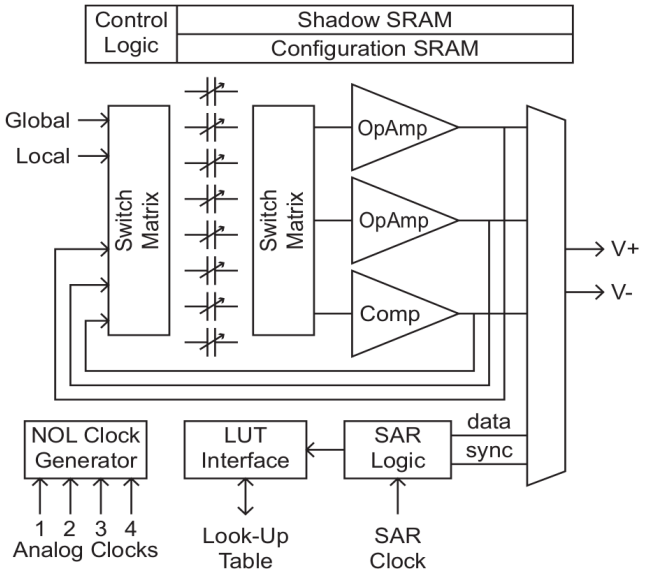

Figure 2: Layout of a Configurable Analog Block (CAB) for the AN121E04 / AN221E04 [Ana03]

3.3 Configuration Details

The diagram of a CAB as shown in Figure 2 gives an initial idea of what the configuration will have to handle. It is important to note that this diagram is from the user manual of the AN121E04 and AN221E04 and was omitted in the manual for the AN231E04, but it can be assumed that the basic design has remained the same between these variations of the product. An interesting observation is that the switch matrices, which connect different outputs, seem to suggest that each capacitor and each op-amp is equivalent, both in design and in routing. In this case, every capacitor can be connected to every global or local input, op-amp output, or op-amp input. This significantly simplifies the configuration compared to the other possibility where each capacitor can only connect to a subset of the components, which would be more complex to reverse engineer, as the configuration for each switch would likely be different.

The circuit diagrams, present in Anadigm's documentation, were also used to reverse engineer the configuration data. As an example, the circuit for the inverting sum stage or SumInv module is shown in Figure 3. This module uses six capacitors in its two-input configuration. Each capacitor appears both in the positive and negative signal path, but these two capacitors are configured by the same configuration data and will be considered as one configured capacitor. This is consistent with Anadigm's description of resources, which shows that each CAB contains eight capacitors.

As was discussed in Section 2.1, the hybrid computer uses switched capacitors, so each capacitor has two switches, one connected to its "input" port and the other to its "output" port. In the circuit for the SumInv module, the capacitor C_1 is configured such that its two switches can connect to the signal path or to signal ground. In contrast, the switches of the capacitor aC_1 are always connected to the signal path. The configuration of capacitor switches in the two respective categories are assumed to be similar. Based on this assumption, an initial mapping of the byte addresses of the capacitor switch configuration was constructed. Other strategies, such as changing the input connection of modules and observing which parts of the data change, are also applied at this stage.

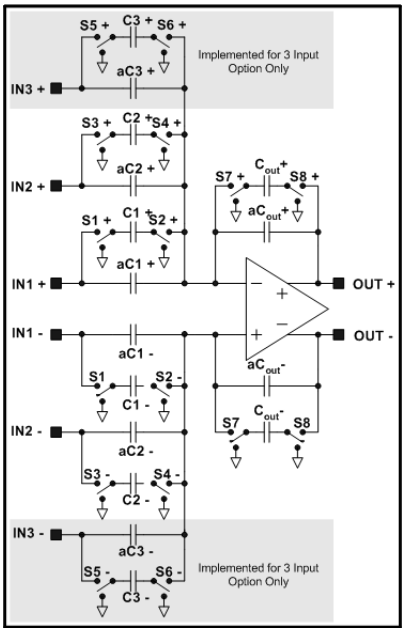

Figure 3: Circuit diagram for an Inverting Sum Stage [Ana25]

Byte Address	00	01	02	03	04	05	06	07
Capacitor Value	8	7	6	5	4	3	2	1

Table 2: Capacitor values in CAB Bank A

Each capacitor is also loaded with an initial value. For most modules, these values persist throughout the lifetime of the configuration. The locations of the values are described in Table 2, where the capacitors are referred to by numerical identifiers. Capacitor IDs are assigned in descending order from left to right in the configuration data. This assignment of IDs mirrors the AnadigmDesigner2 tool, which always uses the capacitor with the highest address that is still available when components are allocated for a module.

The values are represented as byte values from 1 to 255 for active capacitors, and 0 for inactive ones. For most CAMs, the values are obtained by optimizing an equation to realize a certain result, often a gain. An example of this is the computation of the three values necessary for the Inverting Sum Stage, shown in Equation 7.

$$G_1 = \frac{C_1}{C_{out}}, G_2 = \frac{C_2}{C_{out}} \tag{7}$$

The values for G_1 and G_2 represent the gains of each input, a simple scalar multiplication, and are given by the user. The values for C_1, C_2 and C_{out} are then optimized to form the realized gains. For this, the largest values resulting in the closest match are used. For a gain of 2.0, for example, the values $C_1 = 254$ and $C_{out} = 127$ will be computed.

To configure a module such as the inverting sum stage, the capacitors and op-amps need to be connected to each other and to other modules. For this, the configuration of the switches is used.

Each capacitor switch is configured through four bytes. The first two bytes control the input switch and the next two bytes control the output switch. For both switches, various options are possible. A switch can connect to the signal path directly, as is the case for the switches with capacitor aC_1, or it can connect to both the signal path and to signal ground, as can be seen with capacitor C_1. In the latter case, the phase in which the switch is connected to signal ground can change depending on the module.

The CAB layout in Figure 2 shows that the first switch matrix can connect to the outputs of either of the two op-amps, to the comparator or to a global or local channel. The second matrix connects a capacitor output to one of the op-amp inputs or to the comparator. Note that this means that the routing of signals between CAMs is handled through the input port. In contrast, the output port is fixed and does not require configuration.

The global input of the switch matrix is used to connect the CAM to the output of other CABs or to the chip's IO cells. The local input is used for a similar purpose. However, instead of connecting directly to the channel, another part of the CAB's configuration first connects the local channel to the global channel which can then be used by the CAMs through the local channel. This likely accounts for limitations in hardware design. Interestingly, neither the global nor local input is used for connecting to another CAM in the same CAB. Instead, the module connects directly to the op-amp output of the other CAM. This eliminates the strict separation between distinct CAMs, as their functions are no longer clearly separated. Instead of behaving like isolated modules, they operate as an abstract network of analog components.

This method of interconnecting CAMs contained in a single CAB is used by all CABs, except by CAB 3. Instead, CAB 3 routes the op-amp output to a global channel, which is then internally routed back via a local channel to reach the second CAM. This behavior could be caused by hardware limitations. This is surprising, as each CAB is designed to be equivalent. Additionally, CAB 3 is able to use the regular method of connecting to the op-amp output when configuring connections contained in a CAM, but does not do this when connecting two CAMs together. A test forcing the direct connection between two CAMs in CAB 3 using a custom configuration confirms that this does not work. Therefore, this needs to be accounted for in the custom compiler.

The other components that switches connect to are also controlled through the configuration. The op-amp is central to most CAMs. Both of the op-amps on a CAB have an isolated switch in the feedback loop. This switch is not used in the inverting sum stage, as in this case, the feedback loop only consists of the capacitors with their own switches. However, it is used in the integrator shown in Figure 1. Here, the reset switch is configured if the reset option is enabled. In this case, the switch is controlled by the comparator's output, but the switch can also be controlled by a clock.

The comparator's configuration consists of the selection of its inputs, which can be either a local or global input. In addition, the operation of the comparator is controlled by a bitmap. This controls various options, including the clock phase during which the comparator samples its input, whether the output will be inverting or non-inverting and whether the comparison will be between one input signal and signal ground, or between two signals (dual-input mode). The comparator can control the state of the capacitor and op-amp switches, but this is configured by the switch and not by the comparator. The comparator can directly open and close a switch based on its output, as is the case for the integrator circuit. The state can also be changed in other ways. One example of this is the

control of the clock phase during which a switch connects to ground. For instance, a switch might be set up to alternate between connecting to the signal path and to signal ground during clock phases 1 and 2, respectively. By using a comparator signal, this behavior can be inverted, causing the switch to instead connect to signal ground during phase 1 and the signal path during phase 2. This mechanism is used in certain CAMs to dynamically control the sign of the output signal.

The CAB also has access to a Succesive Approximation Register (SAR) and the Look-Up Table (LUT). The SAR is used in combination with the comparator to convert analog signals to digital signals, which can be used to index the LUT. The LUT can then be used to implement functions like multiplication or automatic gain control. The configuration of these components has not been explored, as they are only used in a small number of modules and are thus hard to reverse engineer. They are also not used in the modules that are of interest to this project.

The CAB configuration also controls the clocks that the components can use. Each CAB can select at most two of the six different clocks available on the chip. Each switch can use one of the two clocks. The switch then changes state on the frequency of the selected clock, switching to the first state on clock phase one and to the second state on clock phase two. Some CAMs have constraints on which clocks can be used in order to produce a valid output. For example, the integrator with the reset option enabled requires its second clock, which is used by the comparator, to be two times the frequency of its first clock. This is required to correctly sample the control signal once per phase of the clock selected for the capacitor switches.

Some parts of the analog chip need to be configured on a global level, in contrast to the local, CAB level, which has been the focus for most of this section. The global configuration is stored in the first three banks of the Shadow SRAM and is written to a dedicated part of the system when the configuration is loaded. Most of the global configuration does not change with the design, which means that only a part of it is reverse engineered. The constant sections can also be loaded as constant bytes in the custom compiler, as it can be assumed that this will not change for new designs.

The global bank that was most relevant to the reverse engineering task was the section controlling IO cell modes and IO routing. Every IO cell has various modes and options, which are encoded in this section. For this thesis, only the input bypass and output bypass options are considered, which are encoded through a simple bitmap for each cell. The routing between IO cells and CABs, however, is more complex. Due to limited resources and hardware constraints, only a small amount of channels is available, with specific channels only available to certain IO cells and CABs. These channels are also shared between IO cells. Two different types of channels are available. One routes directly to a single CAB, while the other routes to two CABs, but for which the CAB also needs to use its local routing. The exact CABs that each channel connects to depends on the IO cell. However, enough channels are available to allow every IO cell to route to every CAB in every configuration without conflicts. The bi-directional switch matrix controlling the connections is pictured in the chip layout in Figure 4.

Another section of the global configuration controls the six clocks. Each clock bases its frequency on one of two system clocks. In turn, the system clocks base their frequency on the ACLK input of the analog chip. Both the frequencies of the system clocks and of the chip clocks can be scaled depending on design requirements. The chosen multipliers are encoded in the configuration, as well

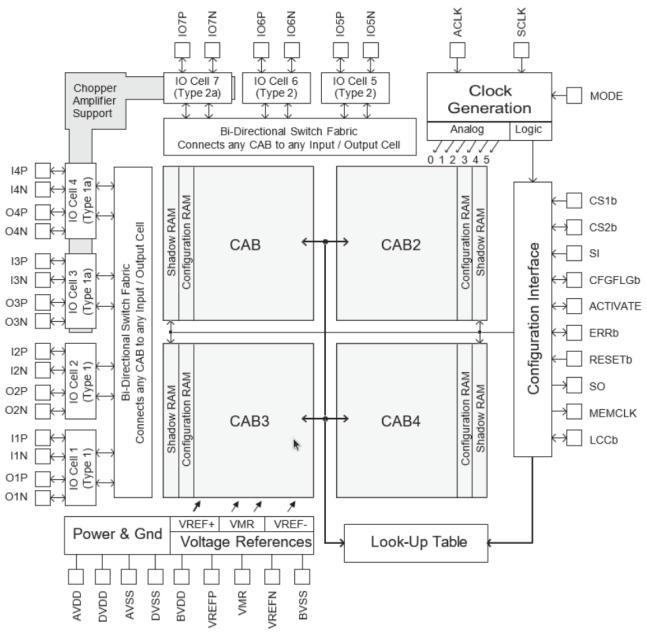

Figure 4: Layout of the AN231E04 chip [Oki25b]

as the system clock that the chip clock is based on. Additionally, the clocks that are in use by the design are encoded using a bitmap.

3.4 System Limitations

During the process of reverse engineering, several system limitations have been discovered which will need to be considered when simulating PDEs. One of these limitations is the fact that the digital microchip is not able to directly read values from or write values to the analog chip. This is an important feature to facilitate the iterative process of solving PDEs, where the values from previous iterations need to be loaded to the chip and the new approximation of a value needs to be stored for usage in future iterations. It may be possible to flash the digital microchip with new software capable of direct communication with the analog chip, but this is out of scope for this thesis. The custom compiler will therefore focus on single iteration configurations which do not require storing values and reconfiguration.

Another limitation pertains to the initial value of the integrator. For many ODEs, the integrator needs to be initialized to a starting value before the system can be simulated. The default integrator CAM, however, does not support this and can only be reset to the value of zero. Several options are considered to still achieve this functionality. Implementing this as a custom, standalone CAM is likely not feasible because of the limited routing available through the switch matrices. Another option uses a gain stage with switchable inputs to output a reset value when the integrator is disabled and outputs the integrator's output otherwise. This circuit is able to successfully load a different reset value, but it uses more resources than the default integrator and the reset value does not seem to correspond linearly to the constant input value. These disadvantages therefore make the implementation hard to use. The compiler and experiments will focus on simple PDEs that can be simulated with a reset value of zero.

4 Compiler

The second main part of the project consists of developing a compiler with the configuration data of the AN231E04 as its target. The main purpose of the compiler is to serve as a backend to the existing toolchain of PDE compilation for analog computers. It would replace the role of the AnadigmDesigner2 tool in the toolchain. This has several advantages. An input file in plain text can be more descriptive of a complex system than the ECAD configuration. It also avoids the visual complexity of CAMs and wires that arises when building a complex configuration. Additionally, some CAM options, such as the selected gains, are also hidden behind menus, which means that the system cannot be understood completely from the chip layout. These issues are avoided by a clear text file format. This file can also include numerical constants and expressions solved at compile time, which improves the modularity of a system.

The initial goal of the compiler was to use the intermediate representation as generated by the ODE-Compiler by Duivenvoorde as input [Dui24], which would then be parsed and compiled to the configuration data compatible with the hardware. However, due to problems with the generated intermediate representation, this is not feasible to accomplish within the scope of the project. The ODE-Compiler tool appears to incorrectly assign a single output variable to multiple internal variables, leading to ambiguity. Additionally, several assumptions were made that do not hold for this specific architecture. For example, the tool assigns exactly one operation to each CAB. This is inconsistent with the AN231E04's configuration, as multiple operations can fit in a single CAB.

Operations on constant variables, such as multiplying constants to form a scalar, are also not handled separately and are instead treated as regular variables with the operations applied at runtime. While this is not a problem for the tool's simulator, the actual hardware only has limited resources. Allocating these for static constants results in fewer resources being available to run the actual calculations.

As the intermediate representation could not be used, and creating a new compiler or modifying the existing tool is out of scope, a new file format providing complete control over the layout of CAMs in the chip will be introduced instead. This format is described in Section 4.2 and a complete example is given in the appendix.

4.1 Overview

The first step in building the compiler was modeling the chip's architecture in C++ code. The chip is modeled on two levels. The first level is represented by the chip itself, which owns its CABs, IO-cells, and clocks. The CABs form the second level, which own their capacitors, op-amps and comparators. The modules (CAMs) are implemented as separate classes which can be added to CABs. The classes inherit from an abstract base module class which means that the compiler can easily be extended with additional modules. The API was designed to be both readable as well as easy to use in the file format parser. An example of the API is shown in Figure 5, where a chip is set up with an integrator module.

The compilation to the configuration data happens in multiple stages. The first stage is run when a module is added to a CAB. In this stage, the module requests the resources it requires from the CAB that is assigned to implement the module. In the example with the integrator, the module will

```
AnalogChip chip;

chip.io_cell(1).set_mode(IOMode::InputBypass);
chip.io_cell(2).set_mode(IOMode::InputBypass);

chip.cab(3).setup(chip.clock(1), chip.clock(3));

auto &integ = chip.cab(3).add(new Integrator(0.42, true));

chip.io_cell(1).out(1).connect(integ.in(1));
chip.io_cell(2).out(1).connect(integ.comp().in());
```

Figure 5: Example of the usage of the ODE-Backend API

claim two capacitors, one op-amp and one comparator. If any of these resources are not available, an error is thrown. As the CAB that the module is assigned to is chosen by the user, prior knowledge about the resources of the system is required to use the API. If the compiler is extended with an automatic allocator, then the allocator is expected to handle the resource constraints.

The next stages are started when the chip's compile method is called. The compiler first sets up data structures to track the network of connections between IO-cells and CABs, and the routing of IO-cells is now compiled. The data structures are used in the next stage, where the modules are finalized. Each claimed component is set up with the required options and the structure of the module is build. Capacitor switches are connected to other components and to external ports, for which the routing data structure is referenced. From this, the components can be compiled. Note that the modules themselves do not directly write to the compiled configuration data. Instead, the components that were finalized previously write their own data to the configuration. This allows the modules to be easily extensible. During this stage, the clocks used by the system are also marked. In the next stage, the configuration describing the clocks is compiled. After this stage, the configuration, represented by a virtual Shadow SRAM array, is completed.

In the final stage, the Shadow SRAM array is converted to the byte array representation as described in Section 3.1. The array must be segmented into data sections such that the total size of the output array is minimized. Each data section is preceded by three header bytes determining the bank and byte address, the size and additional options. The section is also terminated by a synchronization byte, used to check errors. This means that a total of four bytes is added to the data section. Only non-zero configuration bytes need to be explicitly written, but short sequences of up to four zero bytes are still allowed within data sections. This is because introducing a new data section to skip a small number of zero bytes would increase the overall size due to the fixed header and synchronization byte overhead. As a result, zero bytes are only skipped when longer zero runs make it worthwhile to start a new section.

4.2 ACF Input File

The compiler accepts a new input file format, the Analog Configuration Format (ACF). It provides a readable, unambiguous and concise method of programming the AN231E04. The language is centered around objects using the following syntax:

```
{
  <identifier>: <value>
}
```

This type of declaration is similar to JSON, but the allowed structure for `<value>` depends on the context. Types of values include CAB objects, CAM objects or routing declarations. Keys can be omitted if the value is equal to the default value and can be specified in any order. The language is evaluated from top to bottom, meaning that dependencies need to be declared before they are used. Each key can only be present once per object.

At the top level of the format, constant values can be declared using the following syntax:

```
let <identifier> = <expression>
```

The declared constant can be used by other constants and in expressions controlling the value of CAM options. The chip structure is also declared on the top level. Currently, only a single chip is supported, but the language is designed to be easily extensible to multiple chips. The chip contains three keys: `io`, `cabs` and `routing`.

The `io` key sets up the chip's IO cell modes and expects a list with three possible values per entry: `input`, `output` or `-`, the latter of which is used to represent an unused IO cell.

The modules included in the chip are specified in CABs through the `cabs` key. This key expects a list of CAB declarations, which use the following syntax:

```
cab <id> with clocks <clock-a>, <clock-b> {

}
```

Here, `<id>` represents the numerical ID of the CAB and ranges from one to four. The clock values are numerical clock IDs from one to six. The value `-` is also accepted to disable a clock. The CAB structure only accepts a single key, `cams`. This specifies a list of CAMs included in the CAB. The following syntax is used to declare a CAM:

```
cam <cam-name> as <cam-key> {

}
```

CAM Name	Options
GainInv	`gain` $\in [0.1, 1]$
SumInv	`inputs` $\in \{1, 2, 3\}$ `gain1`, `gain2`, `gain3` $\in [0.1, 1]$
Integrator	`inputs` $\in \{1, 2, 3\}$ `integ_const1`, `integ_const2`, `integ_const3` $\in \mathbb{R}_{>0}$ `invert1`, `invert2`, `invert3` $\in \{true, false\}$ `reset` $\in \{true, false\}$
GainSwitch	-
SampleAndHold	-

Table 3: CAMs available in the backend compiler with available options

The value of `<cam-name>` should match one of the CAMs included in the compiler, such as `SumInv` or `GainSwitch`. Each CAM is also assigned a key, which is used to reference the module in a later section. The options available depend on the specific CAM. For example, `SumInv` accepts the options `inputs` to control the amount of input ports and `gain1`, `gain2` and `gain3` to control the gain of each input. A complete list of CAMs and the available options is given in Table 3. For the integrator, the aliases `integ_const` and `invert` can be used instead of `integ_const1` and `invert1` if it is configured in single-input mode.

Finally, the last key of the chip structure, `routing`, represents the connections between CAMs. It expects a list of routing declarations, configured through the following syntax:

```
<output-name>:<output-port> -> <input-name>:<input-port>
```

Both `<output-name>` and `<name-name>` should be one of the declared CAM keys or one of the pre-declared IO cell names, which are named `io1` through `io4`. The values for `<input-port>` and `<output-port>` specify the port that the connection should use. Both the input and outport port can refer to a numerical ID. The port value can also be omitted if no ambiguity exists. For the input port, the value `cmp` can be used to route to the comparator's input port.

A complete example of an ACF file of the heat equation is given in the appendix. This is explained in more detail in Section 5.

4.3 Testing

The compiler can be automatically tested using the `test.py` Python script. A `tests` directory is included in the project which contains various test cases, consisting of an ACF file and the expected output. For most tests, the output exactly matches the output of AnadigmDesigner2. However, in some cases, the compiler routes a design in a different but equivalent way. These outputs have been manually validated.

Current test coverage includes CAM layout verification, CAM-to-CAM and IO-to-CAM routing, as well as complete configurations such as a discretized heat equation. Each output is a list of `bank_count` $\times$ `bank_size` $= 11 \times 32$ integers. This format can be generated either from an

AnadigmDesigner2 C++ project using the `write_raw.py` Python script or by compiling with the `--raw` option. For each test, the compiler is invoked with the relevant ACF file, and the resulting output is compared to the expected file. The Python script reports any compilation errors or mismatches.

5 Experiments

To test the feasibility of simulating Partial Differential Equations on the Anadigm AN231E04, an experiment is conducted. The one-dimensional heat equation discretized over a small number of grid points is chosen for the simulations. This equation can be simulated by using single input integrators in combination with external inverting sum and inverting gain stages. The two boundary values are controlled through an input voltage, which simulates external influences on the modeled system.

The heat equation has been discretized over four grid cells. The following system of ODEs is simulated.

$$\dot{u}_1 = \alpha(u_0 - \beta u_1 + u_2)$$
$$\dot{u}_2 = \alpha(u_1 - \beta u_2 + u_3)$$

The parameters α and β can be tuned to influence the behavior of the system. For the standard heat equation discretized in space, β is equal to 2, while α is used as a scaling factor on the rate of change.

The Analog Configuration File for the simulated design is given in the appendix, while the ECAD design is given in Figure 6. The parameter α is used in both the integrator and inverting sum stages and is therefore expressed as the product $\alpha = \alpha_1 \cdot \alpha_2$, where α_1 and α_2 represent the gain value of the summation and integrator CAMs respectively..

The design contains two three-input inverting sum stages, each connected to the input of an integrator. The sum stages combine the output of the two neighboring cells with the cell's own negative feedback. To account for the sign inversion of the sum stage, the signal of the neighboring cell is inverted using an inverting gain stage for each integrator. This means that the signal of the boundary cells u_0 and u_3, expressed as inputs to the system, are inverted, but this does not affect the system as the sign of the input signals can be adjusted using the signal generator.

The first experiment simulates the system using the parameters $\alpha_1 = 0.04$, $\alpha_2 = 4$ and $\beta = 2$. The input signals consisted of pulses of 200 kHz, a rising edge of 10 ns, a pulse width of 200 μs and a high level of 0.5 V on each input. The second signal has a delay of 100 μs. The third grid cell u_3, representeed by the teal oscilloscope probe in Figure 6, has been measured for the results.

The results of the experiment are shown in Figure 7. Three phases of the pulses are visible. At $t = 0\mu s$, the first pulse from u_0 begins. This causes a convergence toward a stable heat distribution within the system. At $t = 100$ μs, the second pulse from u_3 begins. As this cell is closer to the measured cell, this pulse causes a larger change in heat before the system again converges to a new stable state. When the first pulse ends at $t = 200$ μs, the cell gradually cools. Finally, the second pulse ends at $t = 300$ μs, and the system returns to its default equilibrium state at 1.5 V or virtual ground. The active period, defined as the time between the start of the first pulse and the system's convergence back to the default state, is measured from the waveform to be approximately 306μs.

By adjusting the direct negative feedback β, different results are obtained. The resulting waveform

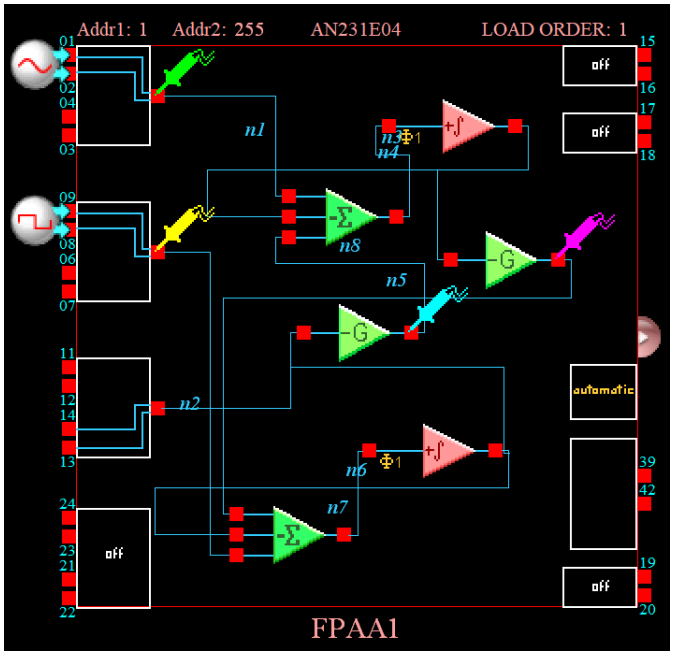

Figure 6: A circuit in AnadigmDesigner2 for simulating the discretized heat equation

of a simulation with $\beta = 1.5$ and $\beta = 2.5$ are shown in Figures 8 and 9, respectively. The first result shows that the influence of the external boundary cells increases as the negative feedback is reduced, while the opposite can be observed in the second result. This directly corresponds to the adjusted parameter, as the feedback from the cell's neighbors is not adjusted. Interestingly, the active period also changes. This is a result of the system needing more time to converge back to the equilibrium from a higher temperature.

In the next simulation, the scaling factor was reduced while keeping $\beta = 2.5$ constant. The waveform is shown in Figure 10. This change affects the rate of integration. The resulting stable temperatures at each stage are unaffected, but the convergence to the stable state is visibly shower. The active period also changes from 303μs to 311μs. The slower integration leads to more gradual changes in voltage, extending both the heating and cooling phases of the system.

The next experiments are performed with β set to 1. While this does not correspond to the heat equation, it results in distinct systems that can be used to model a different type of energy dynamics. The interpretation of the boundary values is changed. In this system, a positive input heats the system up or adds energy, while a negative input cools the system down or removes energy.

This design has been simulated with various settings for the input voltages. The first result was obtained by using a 0.6 V pulse of 30 kHz on one input with a pulse width of 9 μs and a rising edge of 10 ns. The other input was set to a constant input of -0.1 V, representing a constant cooling condition. The resulting waveform is shown in Figure 11. The results show a noticeable ripple effect on the transition edges, which is an artifact of the high input clock frequency. These clock settings were chosen to be able to match the result with AnadigmDesigner2's simulator, which has a very small simulation window. Despite this, the results match the expected waveform. The system continually removes heat, as modeled using the input of -0.1 V. Every pulse of the other

Figure 7: Resulting waveform of the heat equation with two overlapping pulses

Figure 8: Resulting waveform of the heat equation with decreased negative feedback and two overlapping pulses

Figure 9: Resulting waveform of the heat equation with increased negative feedback and two overlapping pulses

Figure 10: Resulting waveform of the heat equation with decreased negative feedback, a low integration constant and two overlapping pulses

Figure 11: Resulting waveform of periodic heating from 35 kHz pulses with constant -0.1 V cooling

input causes a sudden increase in heat, which saturates the output voltage. This explains the flat top at each period of the clock. After the pulse, the added heat is removed from the system.

Reducing the frequency of the pulse input to 30 kHz results in the waveform shown in Figure 12. In this figure, the pulses are less frequent, which allows the constant cooling condition to fully cool down the system to the negative saturation limit.

Instead of adjusting the frequency, the amount of removed heat from the constant boundary condition can also be adjusted. For the third setup, the experiment was performed by restoring the pulse input to 35 kHz, but instead changing the constant cooling input to -0.2 V. This results in the waveform pictured in Figure 13. In the absence of pulse input, the negative saturation limit is reached. This differs from the first setup, as in this case, more heat is removed. Each pulse causes an increase in heat, but this is removed faster than in the previous two setups, resulting from the increased cooling factor.

In the final setup, the constant boundary condition is changed to a sine wave, simulating a variable cooling input. The sine wave ranges from -0.3 V to -0.17 V at a frequency of 1 kHz. The waveform is shown in Figure 14. The cooling input is constantly removing heat, which means that the system settles at negative saturation during the period between pulses. However, the amount of removed heat varies. This means that the pulse input adds less heat to the system when the cooling input is on its lowest level than when it is on its highest level. This creates the sine wave pattern through pulses that is visible in the waveform.

The different setups of the simulation have shown that the system can approximate the behavior of the heat equation under specific conditions and thus that it is possible to model PDEs on the AN231E04. However, there are some limitations to the performed simulations. When simulating PDEs, each grid point represents the value of the solution at a specific, discretized point in space. This means that in the ideal case, all points should be measured. However, this is difficult to achieve

Figure 12: Resulting waveform of periodic heating from 30 kHz pulses with constant -0.1 V cooling

Figure 13: Resulting waveform of periodic heating from 30 kHz pulses with constant -0.2 V cooling

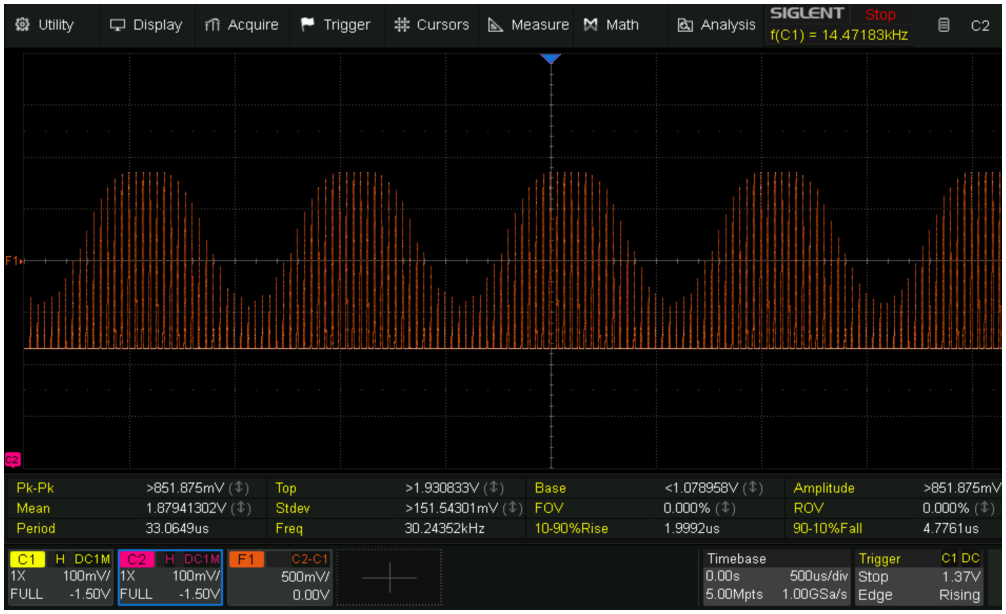

Figure 14: Resulting waveform of periodic heating from 30 kHz pulses with sine wave cooling

on real hardware. The setup of the experiment could be extended to gather more data using off-chip wiring, but this is not in scope for this thesis.

6 Related Work

Prior research has explored the simulation of PDEs on analog or analog-digital hybrid computers. However, most of this research targets purpose-built hardware with specialized components. This thesis expands on this research and introduces a new method for simulating PDEs on the general-purpose, commercially available Anadigm AN231E04 dpASP hybrid computer, thereby improving the accessibility of hybrid PDE simulation on affordable platforms.

In this section, several related studies on the analog simulation of PDEs are outlined. Papers on the implementation of FPAA chips using techniques relevant to the AN231E04 are also discussed. Additionally, a commercial product involving an alternative front-end to the AN231E04 is described.

6.1 PDE Simulation

An application note for the Analog Paradigm hybrid controller by Ulmann demonstrates simulation of the one-dimensional wave equation using a discretization method similar to the one used in this thesis [Ulm20]. In this work, the hybrid controller is only used to control inputs and read outputs of the analog computer, while the circuit itself is built manually. This contrasts with the approach in this thesis, which uses a custom compiler to automatically generate circuit designs for the AN231E04.

Analog reconfiguration is applied in a paper by Huang et al., which simulates the two-dimensional viscous Burgers equation using analog accelerators [HGS+17]. A C++ interface is used to configure the FPAA and to read its outputs. The resulting solution is used as an initial guess, which is then refined by a digital solver to improve the accuracy. The results show a substantial improvement in performance and energy efficiency. Without the use of the digital solver, the analog accelerators produce an approximation within 5% of the true solution and operate nearly 100× faster than a GPU solver. For this research, prototyped analog accelerators are used, each with four "tiles" closely resembling the CABs as described in this thesis. Unlike the AN231E04, the accelerators are not commercially available and each tile includes more components, such as four integrators and eight multiplier–gain blocks. This simplifies the configuration compared to the device used in this thesis.

6.2 Relevant FPAAs

A paper by Kutuk and Kang describes the implementation of an FPAA that uses switched capacitor techniques [KK98]. The FPAA also has other similarities with the AN231E04, including the use of four interconnected, fully configurable CABs and the application of programmable capacitor arrays, which are only used for unswitched capacitors in this design. A key difference between the chips is in the complexity of the configuration. The FPAA described in this paper uses 8-bit control words to program elements sequentially, with each word encoding both the target address and the desired value. This effectively changes the configuration to a serial process, as opposed to the configuration data of the AN231E04, which configures the complete system in parallel.

The implementation of the DPAD2 is discussed in a paper by Bratt and Macbeth [BM98]. This FPAA consists of twenty identical CABs, each containing a single op-amp. Notably, the DPAD2 architecture includes both local and global routing. Global routing enables connections between

arbitrary CABs and IO cells, while local routing supports efficient connections over short distances. This mirrors the implementation of IO cell routing on the AN231E04, where an IO cell can directly connect to nearby CABs or use global channels to connect to other CABs.

6.3 AN231E04 Front-end

The Zrna Research Group has developed a commercially available product based on the AN231E04 that primarily targets audio processing applications [Zrn25]. It is programmable via a Python API, similar to this thesis' C++ interface. However, one major difference is in the compilation and transmission of the configuration data. The Zrna API directly communicates with the digital microchip. When a gain parameter is changed through the Python API, a request is sent to the digital microchip, which handles the translation and compilation of configuration data. A separate command, triggered by a `run()` call, initiates the transmission of this configuration to the AN231E04. While the Python API is open source, the compilation firmware on the digital chip remains proprietary. This means that the Zrna API can only be used with its specific hardware platform. By contrast, the compiler developed in this thesis is fully open source and can be used with any digital system capable of communicating with the AN231E04.

7 Conclusions and Further Research

To simulate PDEs on hybrid computers, challenges need to be solved both on the theoretical side as well as on the hardware side. The theoretical aspect involves converting the system of PDEs to a system of ODEs, which is then split up into grid groups to allow constrained, physical hardware to simulate them. This conversion has been explored in previous work [VDE23] and this thesis has focused on the hardware aspect of the problem.

Simulating using the general purpose Anadigm AN231E04 dpASP hybrid computer is especially challenging considering its limited resources, unspecialized blocks and undocumented configuration. As a first step, a large part of the configuration data has been reverse engineered. This was done using a combination of the AnadigmDesigner2 software as well as the module circuits found in Anadigm's documentation. This has revealed many details about the configuration, mainly focusing on the required components used for PDE simulation. Based on these findings, capacitors, op-amps, and comparators in each of the four configurable analog blocks can be set up and interconnected via switches. The routing between blocks and to IO-cells has also been explored.

The findings have been used to build a compiler. The compiler uses a custom textual configuration file format to build configurations, which is then converted to the configuration data used to program the AN231E04. It models the analog chip and allows the user to easily add modules to the circuit and create connections between them. The compiler is also easily extensible, allowing users to extend the module hierarchy beyond the modules that were implemented for this thesis.

To test the feasibility of PDE simulation on the AN231E04, several experiments were performed. The one-dimensional heat equation was chosen for these experiments. It was discretized to a system of ODEs of four grid points, of which two were defined by ODEs and the two boundaries were defined by inputs to the analog system. By modeling the equations using analog modules and simulating the resulting circuits, the results were obtained. The waveforms successfully approximate the heat equation, which shows that PDEs can be simulated on the device.

7.1 Further Research

This thesis can be used as a starting point for research in various directions. In this section, general research will be discussed, while Section 7.2 focuses on improvements to the compiler specifically.

In the general toolchain of the analog PDE simulation compiler, various enhancements can be researched. One enhancement combining the theoretical with the practical side of the project is the addition of a type system. This would allow the compiler to automatically deduce the domain of variables and could extend the compilation of boundary values beyond constants. The intermediate compiler can also be improved to handle the grouping of variables and compile time evaluation of expressions involving static constants, among other optimizations.

As only a part of the configuration data of the AN231E04 has been reverse engineered for this thesis, exploring this further could be an area for further research. This could be focused on analyzing the routing to the other IO-cells which was ignored for this thesis, as well as how the device configures components such as the look-up table and successive approximation registers. The depth of reverse engineering could also be extended, developing new insights into the usage of bitmaps, multiplexer

selectors or other techniques used to configure the device.

Additionally, as the toolchain consists of modular compiler tools, alternative backends can be developed targeting other analog computers or digital simulators. Research could focus on comparing results between devices on measures such as accuracy or simulation time, utilizing the strengths of different options which may include specialized analog blocks for operations such as multiplication.

7.2 Improvements to ODE Backend Compiler

The compiler that was developed in this thesis allows for further improvements. The module system of the compiler is easily extensible, which accommodates future enhancements. The compiler could also be extended with an option to read from input files using a parser, which would make the tool more accessible. Furthermore, other challenges relating directly to ODE simulation can be addressed, which includes the automatic configuration of the reset value of the integrator or the set up of reading and writing from the digital chip, allowing for the storage of intermediate variables.

Finally, the compiler could be extended to accept a higher-level input format which does not directly specify the chip layout and instead gives a more general overview of the desired functionality. This would allow the compiler to apply more advanced optimizations, which does not only relate to optimizing the amount of modules that can fit in a block, but also involves the physical layout of analog blocks in relation to the IO-cells. This impacts the complexity of the configuration and may improve the efficiency and accuracy of the system on real hardware.

References

[Ana03] Anadigm, Inc. *Field Programmable Analog Arrays - User Manual - AN121E04 / AN221E04*, 2003. Available at https://archive.org/details/manualzilla-id-6881502.

[Ana25] Anadigm, Inc. *AnadigmDesigner2 Software Documentation*. Okika Devices Corporation, 2025. Available at https://okikadevices.com/collections/buy-products/products/anadigm-designer-2-graphical-analog-design-software-for-flexanalogtm-products.

[BD01] William E Boyce and Richard C DiPrima. *1.3 Classification of Differential Equations*, page 17–23. John Wiley Sons, Inc, 7 edition, 2001.

[BM98] Adrian Bratt and Ian Macbeth. Dpad2—a field programmable analog array. *Field-Programmable Analog Arrays*, page 67–89, 1998.

[Cra16] Linda Crane. Back to analog computing: Columbia researchers merge analog and digital computing on a single chip, Nov 2016.

[Dui24] Walt Duivenvoorde. Simulating differential equations using hybrid digital-analog computers. Bachelor's thesis, Leiden University, 2024.

[GHM+16] Ning Guo, Yipeng Huang, Tao Mai, Sharvil Patil, Chi Cao, Mingoo Seok, Simha Sethumadhavan, and Yannis Tsividis. Energy-efficient hybrid analog/digital approximate computation in continuous time. *IEEE Journal of Solid-State Circuits*, 51(7):1514–1524, 2016.

[Has20] Jennifer Hasler. Large-scale field-programmable analog arrays. *Proceedings of the IEEE*, 108(8):1283–1302, Aug 2020.

[HGS+17] Yipeng Huang, Ning Guo, Mingoo Seok, Yannis Tsividis, Kyle Mandli, and Simha Sethumadhavan. Hybrid analog-digital solution of nonlinear partial differential equations. *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, page 665–678, Oct 2017.

[KK98] Haydar Kutuk and Sung-Mo Steve Kang. A switched capacitor approach to field-programmable analog array (fpaa) design. *Field-Programmable Analog Arrays*, page 51–65, 1998.

[LH98] Edward K. Lee and Wai L. Hui. A novel switched-capacitor based field-programmable analog array architecture. *Field-Programmable Analog Arrays*, page 35–50, 1998.

[Mac07] Bruce J MacLennan. A review of analog computing. *Department of Electrical Engineering & Computer Science, University of Tennessee, Technical Report UT-CS-07-601 (September)*, pages 19798–19807, 2007.

[Oki25a] Okika Devices Corporation. *AN231E04 Datasheet Rev 2.1*, 2025. Available at https://okikadevices.com/cdn/shop/files/AN231E04DSv2.1.pdf.

[Oki25b] Okika Devices Corporation. *FlexAnalogTM FPAA User Manual - AN231E04*, 2025. Available at https://okikadevices.com/cdn/shop/files/FlexAnalog-AN231E04UM_User_Manual.pdf.

[Sun05] Nancy Y Sun. *A DC stabilized fully differential amplifier*. PhD thesis, Massachusetts Institute of Technology, 2005.

[Ulm20] Bernd Ulmann. Anabrid application note #23: Solving pdes on an analog computer, Jan 2020.

[VDE23] Dirck Van Den Ende. Towards a compiler for partial differential equations for analog computers. Bachelor's thesis, Leiden University, 2023.

[Zil18] Dennis G Zill. *1.3 Differential Equations as Mathematical Models*, page 22–34. Cengage Learning, 2018.

[Zrn25] Zrna Research, 2025. Available at https://zrna.org.

A Appendix: ACF File for the Heat Equation

Here, an ACF input file for the custom compiler is shown which configures the one-dimensional heat equation with two ODEs and two boundary conditions. The process of discretization is described in Section 2.2, and this system is simulated in the experiments in Section 5.

The system is configured over multiple CABs. The inverting sum stage in CAB 1 is used for the first ODE and the inverting sum stage in CAB 2 is used for the second ODE. These sum stages are connected to the integrators 1 and 2 and to the inverting gain stages 1 and 2, respectively.

```
# Main clock at 4,000 kHz
let ClkA = 1;

# alpha = alpha_1 * alpha_2
let alpha_1 = 0.04;
let alpha_2 = 4;

# controls direct negative feedback from grid cell
let beta = 2;

chip {
    io: [
        input,
        input,
        output
    ],
    cabs: [
        cab 1 with clocks ClkA, - {
            cams: [
                SumInv as sum1 {
                    inputs: 3,
                    gain1: alpha,
                    gain2: alpha * feedback,
                    gain3: alpha,
                }
            ]
        },
        cab 2 with clocks ClkA, - {
            cams: [
                SumInv as sum2 {
                    inputs: 3,
                    gain1: alpha,
                    gain2: alpha * feedback,
                    gain3: alpha,
                }
            ]
```

```
        },
        cab 3 with clocks ClkA, - {
            cams: [
                GainInv as gain1 {},
                GainInv as gain2 {}
            ]
        },
        cab 4 with clocks ClkA, - {
            cams: [
                Integrator as integ1 {
                    integ_const: beta,
                },
                Integrator as integ2 {
                    integ_const: beta,
                }
            ]
        }
    ],
    routing: [
        io1 -> sum1:1,
        integ1 -> sum1:2,
        gain2 -> sum1:3,
        sum1 -> integ1,
        integ1 -> gain1,

        gain1 -> sum2:1,
        integ2 -> sum2:2,
        io2 -> sum2:3,
        sum2 -> integ2,
        integ2 -> gain2,

        integ2 -> io3
    ]
}
```