Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Bachelor's Thesis Nr. 385b

Systems Group, Department of Computer Science, ETH Zurich

Declarative Power Sequencing using a CPLD

by

Manuel Hässig

Supervised by

Daniel Schwyn
Dr. Michael Giardino
Prof. Dr. Timothy Roscoe

September 2021 – February 2022

Abstract

Declarative power sequencing solves the difficult and tedious problem of bringing up and managing the power resources of a computing platform by describing the platform in a declarative model and applying constraint solving to obtain correct power sequences from the model. To date these generated power sequences are wholly executed on the board management controller (BMC). However, some platforms provide specialized hardware for the purpose of power sequencing which goes unused. The Enzian research computer for instance features two power sequencing Complex Programmable Logic Devices (CPLDs) which the BMC remote controls when executing power sequences. The goal of this thesis is to devise an algorithm to partition a generated power sequence such that it can leverage the capabilities of specialized hardware and to generate instructions for the power sequencing CPLDs on Enzian.

Acknowledgements

Before I start with my thesis I would like to take a few lines and express my gratitude. I would like to thank the Systems Group at ETH and especially my supervisors Daniel Schwyn, Dr. Michael Giardino, and Prof. Timothy Roscoe for giving me the opportunity to work on this thesis, for helping me through any and all problems, be they technical, lack of understanding or knowledge, or organizational, and for continually providing me with support during these six months.

Contents

1 Introduction

The process of gradually powering up different parts of the platform in order seems to be a straight forward task. However, with the ever-growing complexity of modern computing platforms which are home to a host of coprocessors the task of turning all devices on in the correct sequence becomes harder. Each device comes with its own voltage demands, sequencing constraints, and with this more complexity. Until recently a programmer would need to tame all this complexity by hand and configure, sequence, and monitor the platform without much help. But this task is tedious and error-prone and it is easy to make a mistake and end up with a cloud of the magic blue smoke that never comes back.

1.1 State of the Art

In order to address this complex task the idea of declarative power sequencing was born [1, 2]. This approach models a platform with a declarative model and specifies the constraints on the platform within this model. Then constraint solving is applied to obtain a correct-by-construction power sequence. This sequence is then executed by the board management controller (BMC) which controls the entire platform. This approach not only prevents errors of oversight, but it also enables easier maintenance of the power sequence as one does not need to fiddle with the intricate monolith of a handwritten sequence.

Further advances have enabled the optimization of certain aspects of the power sequence like the length of the sequence or the power consumption of the platform [3]. This work has also placed a bigger emphasis on computing and optimize power sequences using optimization modulo theories.

None of these works have covered low-level detail about the implementation of power sequence instructions on the executing devices until [4]. This work covered the generation of C-code to dispatch instructions for the Inter Integrated Circuit (I^2C) bus based a generated power sequence to control fans for the purpose of thermal control.

1.2 Contributions of this Thesis

Some platforms have special purpose hardware dedicated to power sequencing. For instance the Enzian research computer has two power sequencing Complex Programmable Logic Devices (CPLDs) included on the baseboard. However, because power sequences are to date only executed on the BMC we cannot take advantage of these CPLDs.

The key contribution of this thesis is the presentation of an algorithm that partitions power sequences among all controllers on a computing platform such that all of them are able to execute parts of the power sequence. In order to achieve this we present a platform model based on [3] where busses and bus actions are modelled explicitly and a model for sequences also based on [3] with a precisely defined semantics for which devices may execute a given instruction at what time.

2 Background

In this chapter we will cover some background which helps with the understanding of the rest of the thesis.

2.1 Enzian

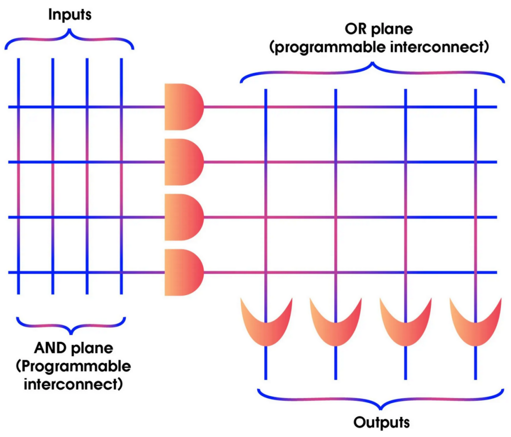
Enzian [5, 6] is a research computer purpose build by the Systems Group at ETH. It features a server grade ThunderX CPU on one node and a high-end FPGA on the other node. The two nodes are connected by a high-bandwidth cache-coherent interconnect. These features make highly-flexible access patterns possible that enable systems research on modern hardware that enables novel offloading techniques as they are applied in industry today but not possible to research on commercial off-the-shelf hardware.

The complex architecture of the Enzian also calls for advanced platform management techniques. The Enzian board features a baseboard management controller with an integrated FPGA as well as two specialized power sequencing CPLDs. Because everything on Enzian is built by the Systems Group, this enables access into parts of the firmware which are usually not accessible to research like the power sequencing of the platform. From this a line of research in to power sequencing [1–4] (see also section 3.1), which this thesis is part of, started.

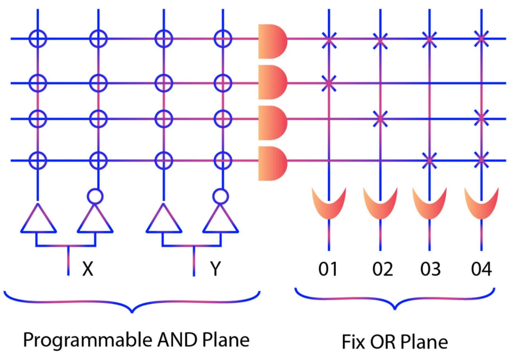
2.2 Baseboard Management Controller

The baseboard management controller, or short BMC, is a device that manages all the devices on the mainboard of a computing platform and monitors platform parameters like voltages, temperatures and much more [7]. It is the first thing that runs when turning a computer on because its job is to bring up the platform and start the boot process and it is the last thing that stops running.

Its job of managing the entire platform comes with wide-reaching privileges which makes it a prime target for attacks. Despite having the highest privileges it also used to have the lowest visibility and inspectability [7]. That is until the open-source BMC



(a) Programmalble logic array [12, figure 4]



(b) Programmalble array logic [12, figure 5]

Figure 2.1: Elementary building blocks of CPLDs.

implementations openBMC [8], u-bmc [9], and RunBMC [10] came along. All of this makes the BMC an interesting research subject.

2.3 Complex Programmable Logic Devices

The Enzian board includes two ISPPAC-POWR1220AT8 Complex Programmable Logic Devices (CPLDs) for power sequencing and monitoring. For convenience and legibility we will henceforth refer to them as ISPPACs. In this work we want to offload parts of power sequences to these CPLDs. For this reason we look at what potential benefits and drawbacks we might get from using CPLDs for this purpose.

2.3.1 What is a CPLD?

At their core CPLD consist of simple function blocks. These are based on programmable logic blocks and depending on the CPLD add some extra capabilities [11]. One possible base block is a programmable logic array (PLA) which consists of a programmable AND-interconnect and a programmable OR-interconnect (see figure 2.1a). For simpler applications designers also base their functional blocks on programmable array logic (PAL) which is similar to PLAs in that it has a programmable AND-interconnect but reduces complexity by having a fixed OR-interconnect (see figure 2.1b).

Such function blocks are then arranged in macrocells and connected by a programmable interconnect. This interconnect consists of a large switch matrix. It is able to connect all the inputs to the CPLD and outputs of all macrocells with the input of any macrocell

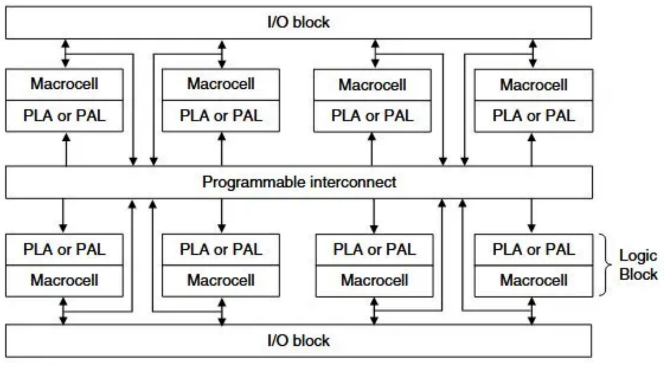


Figure 2.2: Architecture of a CPLD [12, figure 6]

[11] (see figure 2.2). Typically, the output of each macrocell has a dedicated output pin on the CPLD.

In order to store the program CPLDs make use of non-volatile memory technology like EEPROM or Flash memory. These program stores are then flashed using a JTAG serial interface. Further, some domain specific functions are added directly onto the die with the CPLD and connected to the interconnect so the CPLD can compute with provided values or provide input [11].

2.3.2 Advantages of CPLDs

CPLDs are best suited for simple applications due to them containing only a limited number of logic gates. More complicated applications are better accomplished using an FPGA. However, at these simple tasks CPLDs excel.

First and foremost, is the execution of a CPLD fully deterministic, because it is fully determined by logic formulas. Due to their programmable interconnect CPLDs have a deterministic signal delay which is also quite low compared with an FPGA. Because a CPLD operates at a reasonably low operating frequency and its operations are quite simple its power consumption is low enough so we do not need to care about thermal management. Because CPLDs have their programs stored in non-volatile memory, they do not need a special bootstrapping process or external memory storing the program as opposed to most FPGAs.

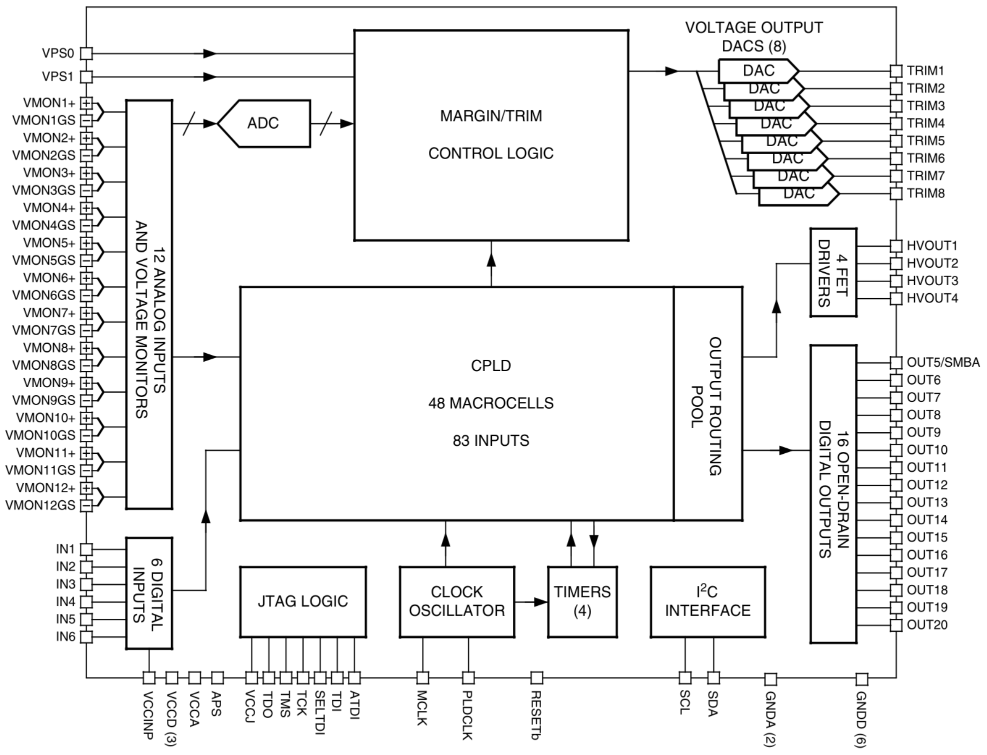

VOLTAGE OUTPUT DACS (8)
VPS0
VPS1
VMON1+ ... VMON12GS
12 ANALOG INPUTS AND VOLTAGE MONITORS
ADC
MARGIN/TRIM CONTROL LOGIC
DAC
TRIM1 ... TRIM8
4 FET DRIVERS
HVOUT1 HVOUT2 HVOUT3 HVOUT4
CPLD 48 MACROCELLS 83 INPUTS
OUTPUT ROUTING POOL
16 OPEN-DRAIN DIGITAL OUTPUTS
OUT5/SMBA OUT6 ... OUT20
IN1 ... IN6
6 DIGITAL INPUTS
JTAG LOGIC
CLOCK OSCILLATOR
TIMERS (4)
I²C INTERFACE
VCCINP VCCD (3) VCCA APS VCCJ TDO TMS TCK SELTDI TDI ATDI MCLK PLDCLK RESETb SCL SDA GNDA (2) GNDD (6)

Figure 2.3: Block diagram of the ISPPAC power sequencing CPLD [13, figure 1].

2.3.3 The ispPAC power sequencing CPLD

The Enzian baseboard features two ISPPAC power sequencing CPLDs which we are targeting to execuute power sequences on in this thesis. In this section we have a closer look at the device and how to program it.

Overview The ISPPAC features 12 analog voltage monitor inputs, six digital inputs, 16 digital outputs as well as four programmable timers an I^{2}C slave interface, and a JTAG interface for programming (see figure 2.3, there are more features we do not use on Enzian). It is powered by a 48 macrocell CPLD which can be programmed using the PAC-Designer tool.

Voltage Monitors Each voltage monitor features two programmable voltage trip points, a lower trigger voltage and an upper trigger voltage, which return false if the input voltage is below the programmed voltage and true if it is above. The upper trigger is available to the CPLD on a pin VMONxA and the lower trigger on VMONxB. By programming these with over and under voltage thresholds we can determine if a wire has been powered correctly, which is the case if the lower trigger is true and the upper and the upper trigger is false. We can also program the monitor pin to operate in windowed mode, which is true in the condition described before. If the windowed mode is configured for a pin then the A pin shows the window value instead of the upper trigger [13]. The voltage triggers must be configured at compile time and they can be updated with a bus command. The output of all triggers can also be read over the bus.

Input Pins The six digital input pins provide digital input for the ISPPAC. The pin `IN1` can be configured to take its value from the pin itself or it can be controlled through the JTAG interface. The other input pins `IN[2..6]` can be configured to take their input from the pin or an I^{2}C-register. Both of these configurations cannot be changed at runtime.

Output Pins The 16 digital output pins are the primary outgoing interface of the ISPPAC to the outside world. These pins can either be controlled directly by the CPLD or from an I^{2}C-register. This configuration can also not be changed at runtime. Further, we can configure the reset value for the outputs pins after a reset to be high or low. The pin `OUT5` is special because it can be used as a normal output or like on Enzian as an SMBus alert pin [13].

Timers The ISPPAC features four programmable timers which can individually be programmed to times ranging from 32 μs up to 1.96 s. They can be used in the CPLD for timeouts or pausing the execution.

Programming The ISPPAC is programmed and configured in the PAC-Designer GUI-Program [14]. The program and configuration are then compiled to a JEDEC bitstream and can then be flashed with the Diamond programmer over the JTAG interface. Unfortunately, the program does not take any file input except its proprietary XML-based format to store projects.

PAC-Designer provides the LogiBuilder language as an abstraction for programming the ISPPAC. Unfortunately, the closest thing to a specification can be found in the PAC-Designer manual [14]. We quickly explain the instructions used in this work.

The `Begin Startup Sequence` and `Begin Shutdown Sequence` enable respectively disable exceptions. The outptut statement simply assigns a value to an output pin, e.g. `OUT8 = 1`. We can wait for a boolean expression to be true with `Wait For <boolean expr>` which pauses the execution until the expression is satisfied. Further, we can wait for a timer with `Wait For TIMER<number of timer>` which pauses execution for the time programmed into the timer. As one would expect, the `Halt` instruction ends the execution of that state machine.

The programming model of an ISPPAC is based on state machines. In fact, it can run multiple state machines at the same time. Inside a state machine we can program a sequence. Further, every state machine can have exceptions which are boolean expressions that interrupt the execution of the sequence, perform some action and jump to a specified step in the sequence when the expression is satisfied [14].

2.4 The I^2C Bus

The Inter Integrated Circuit bus, or more commonly I^2C for short, is a low-speed control bus for communicating among components on a circuit boards [15]. It follows a simple master/slave relationship on every device (the I^2C multi-master capabilities are out of scope in this work) and uses only two wires — one clock wire `SCL` and a data wire `SDA` — which is why it is popular as a simple and low-level means of communication for BMCs to control the devices on their board. We will not go into the low-level details which can be found in [4] and [16], which also provides an interesting comparison with other serial busses and other protocols with similar purposes.

On a high level, a bus master on an I^2C bus can send messages addressed to individual slave devices which are then, depending on the received data, to send some data in response. Slave devices are only able to write to the bus when they are called upon by the bus master.

Because of the popularity and simplicity there are bus protocols built on top of I^2C and which can coexist on the same bus. There is the System Management Bus (short SMBus) which markets itself as a protocol specifically for system and power management [17]. The SMBus specification lays out a set of transactions with possibly bidirectional transfers. It also features rudimentary remote procedure call capabilities on slave devices.

The Power Management Bus (PMBus) protocol is an extension the SMBus protrocol and defines a set of commands that abstract a power management device [17]. For example PMBus provides explicit commands for setting output voltages on voltage regulators, setting fault limits for measurements and turning outputs off and on.

All of these protocols are used for managing the devices on the Enzian baseboard.

3 Related Work

This chapter provides a survey on literature covering topics relevant for this thesis.

3.1 Declarative Power Sequencing

The research area of declarative power sequencing is — as far as we can tell — relatively young and was kicked off with Jasmin Schult's bachelor thesis [1] and the resulting paper [2] presented at EMSOFT 2021. In her thesis she laid the groundwork for declarative power sequencing by analyzing the power management of platforms and abstracting this into a declarative model which did not only describe the devices on the platform but also the power related constraints like voltage limits and sequencing constraints. By applying constraint solving to the model she was able to generate correct by construction power sequences which eliminate the tedious and error-prone process of implementing these by hand. She then went on to model the Enzian platform and applied her sequencing techniques to it and was able to successfully demonstrate that a complex server grade computing platform could be powered on and off using power sequences generated from a model of the platform.

Based on this work Morith Knüsel applied optimization modulo theories to the problem and was able to optimize aspects of power sequencing using a SAT solver [3]. With these optimizations he was able to produce much more compact sequences which are able to execute multiple independent instructions in parallel. Further, he was able to reduce the power consumption of a platform by optimizing the sequence such that lower than normal voltages were programmed in the regulators while still respecting the constraints of all devices.

In his Bachelor thesis, Linus Vogel tackled the generation of C code for the control of an as of yet unmodelled fan controller over I^2C [4]. His model explicitly contained busses and their commands. Special attention was given to the issue of non-compliant devices that violate the bus standard advertised in the datasheet.

While the topic of power management is mainly studied in the works outlined above, we were able to find some related work which tries to achieve similar goals in the realm of power management.

Especially interesting is the work surrounding the Unified Power Format (UPF) standard [18] which provides a method to specify power intent for an electronic design. It facilitates the consideration of power concerns in the design of low power hardware. With ever smaller semiconductor technologies and the "power wall" that has been hit. UPF shifts a former purely physical implementation concern early into the design process. It enables the hardware designer to specify power semantics inline in HDL designs. Around these specifications a small formal methods community has emerged with the goal of verifying the correctness of power techniques in modern chips [19–21]. An especially interesting paper addresses the challenge of hardware-software-interaction in power management [22] and provides an approach for using formal methods to verify this interface.

A different line of work around UPF is to enable higher level modelling of power management specifications, which could be viewed as a step in the direction of a declarative model of power management aspects. One such work revolves around extending SystemC [23]. Another models power consumption on a system level using timed transaction-level model of hardware [24].

3.2 Code Generation for CPLDs

The goal of this thesis is to generate code for CPLDs in order to execute power sequences generated from a delclarative model on them. However, as we found out there is next to no work published on high-level sythesis for CPLDs. While there is a host of literature on synthesizing low-level hardware descriptions and optimize logic formulas [25–27] we were mostly unsuccessful in finding works with targeted CPLDs to execute high level code.

A notable exception is the ConCISe [28] which uses a CPLD as a reconfigurable functional unit in a reduced instruction set processor. In the compiler required functional units were identified and clustered over the execution of the program and the CPLD would then be reconfigured to provide the desired functional unit at the appropriate time during the execution on the CPU. This is the only work we found, where high level code was compiled and synthesized to run on CPLDs.

We did however find a host of work describing how CPLDs are used in different applications. Examples include telemetry gathering in nuclear power plants [29], a teaching auxiliary board to enable student experiments [30], and a reconfigurable communication architecture [31].

While we were not able to find a lot of previous work on high-level synthesis for CPLDs there is a lot of work in the field of high level synthesis for FPGAs. Practical examples include using iterative synthesis based on UML models and profiles to generate HDL in a hard- and software co-design approach [32], a dynamically reconfigurable IoT research platform based on a Flash FPGA[33], and an FPGA-based surveillance platform with an image processing core built with a C-to-HDL compiler[34]. Further, a proposal for code-generation techniques for C-based high-level synthesis from a domain specific language embedded in C++ targeting FPGAs [35] and a framework for compiling C programs to VHDL consisting of data-flow analysis and loop transformations [36].

4 Observations & Goals

In this chapter we take a close look at the problem of using CPLDs in power sequencing and discuss possible ways to solve it at a high level. On the way we will identify challenges we will need to address in the design of our models for the platform and the sequence itself. We do this by trying to apply our goal of leveraging CPLDs for power sequencing to the Enzian platform.

4.1 Mapping Power Sequences to CPLDs

Our goal in this work is to extend the power sequencing techniques from previous works [1, 3, 4] to generate instructions on CPLDs. The first question we must ask ourselves is whether we can express power sequences on the CPLD.

We try to answer this by comparing the power sequences generated from the tools by Knüsel with the LogiBuilder instruction set as described in the software manual for PAC-Designer [14], which is used to program the ISPPAC power sequencing CPLD present on the Enzian baseboard. In this comparison we should see how good the former maps onto the latter.

The ISPPAC as it is connected on the Enzian can read if voltages are in a given range and read the value of a logical wire. Further, it can drive logical values. Below, there is a comparison between such instructions from a power sequence on the left and a corresponding LogiBuilder instruction on the right.

Reading, if a voltage is within a given interval:

```
(2.5
  monitor
  wire.12v-cpu1-psup
 ((pac-fpga VMON1 voltage (8500 13900))))
```

```
Step 3: WAIT FOR VMON1
```

Note that the voltage monitoring pins can be configured in different ways. In this example, `VMON1` is configured in window mode.

Reading, a logical value:

```
(2.5
  monitor                                          Step 3: WAIT FOR IN1
  wire.psup-pgood
  ((pac-cpu IN1 value (_ bv1 1))))
```

Writing, a logical value:

```
(6
  set-to
  (_ bv1 1)                                        Step 6: OUT9 = 1
  (pac-fpga OUT9)
  wire.en-vccint-fpga)
```

Further, the CPLD can handle optional and exceptional control flow as well as timeouts or waiting for a given amount of time[14].

The comparison above suggests that power sequences map directly onto the CPLD. Therefore, we should be able to simply take a power sequence and partition it among the available controllers. This approach leads to a separation of a front-end and a back-end in the generation of power sequences, where the front-end is the generation of the sequence and the back-end is the partition of that sequence.

4.2 Multiple controllers

In order to leverage the CPLDs for power sequencing we need to program them beforehand, as they cannot be reconfigured at runtime. Thus, we need to afford them some agency when executing the sequence. However, in previous works there was the assumption that there could only be one controller controlling the entire platform [1]. We do not necessarily need to break with this assumption, as we could simply model the subsystems controlled by the CPLDs as their own platform, where one CPLD is the sole controller. This workaround is predicated on the possibility that the sequence executed by the CPLDs can be executed independently of the rest of the platform.

While the power sequencing CPLDs on Enzian are able to execute power sequencing instructions that are within their capabilities, there is a host of operations only the BMC can perform. For one, the BMC is the only I^2C bus master and as such the only device able to configure devices or observe certain wires. Further, it is the only device that can sequence the clocks necessary for powering up the CPU and FPGA.

Due to this not even the manual power sequence for Enzian [37] contains independent subsequences for the two CPLDs even though the sequence would permit it if the voltage check of the DDR voltages were not only possible via the BMC. Therefore, we must drop the unique controller assumption.

4.2.1 Relaxing assumptions

The fact that controllers have different abilities suggests that different controllers might have different amount of information about the platform. This raises the question whether all controllers have sufficient information to be able to know when to execute the next step in the sequence.

But what information is needed for a controller to be able to execute an instruction? An instruction takes the platform from one known state to another known state. To execute the instruction the controller must realize that the platform has reached the initial state for the instruction and that it can now execute that instruction. If the controller is not able to monitor enough values, it must get this information from somewhere else. Since all controllers together must be able to observe the entire platform state, some other controller could communicate that information to our executing controller. However, not all controllers might be able to communicate with other controllers. For example, the CPLDs on the Enzian baseboard can only receive communication from the BMC, but they have no way of sending information to the BMC (which is not really necessary as the BMC can observe all pins of the CPLDs via the bus).

Therefore, our platform model must capture the communication abilities between controllers in order for us to determine which controllers are able to receive information from fellow controllers.

4.2.2 Where to execute an instruction

Without the unique controller assumption, we might have more than one controller able to execute some instructions. For example, any voltage monitored by the ISPPAC can also be monitored by the BMC using bus commands. Therefore, a decision is needed to determine on which controller an instruction is executed.

In order to make a decision, we need to be able to determine from our model which controller can control which parts of the platform. This is easy to do for pins connected with wires. Busses, however, are much more complex. Therefore, we need a clear interface with concise semantics for busses in our platform model.

These observations suggest, that we do not need to decide on where some instruction is executed at the time of generating the sequence and instead make this decision later. Therefore, we take the approach in this work to seperate the process of generating states and sequences from operating on the generated sequence and generating executable instructions for devices. We consider the first concern to be the “frontend” of declarative power sequencing and the latter as the “backend” (see figure 4.1 on the following page). The crucial step in the backend in order to leverage multiple controllers will be to partition the generated sequence into parts that are executed by multiple controllers.

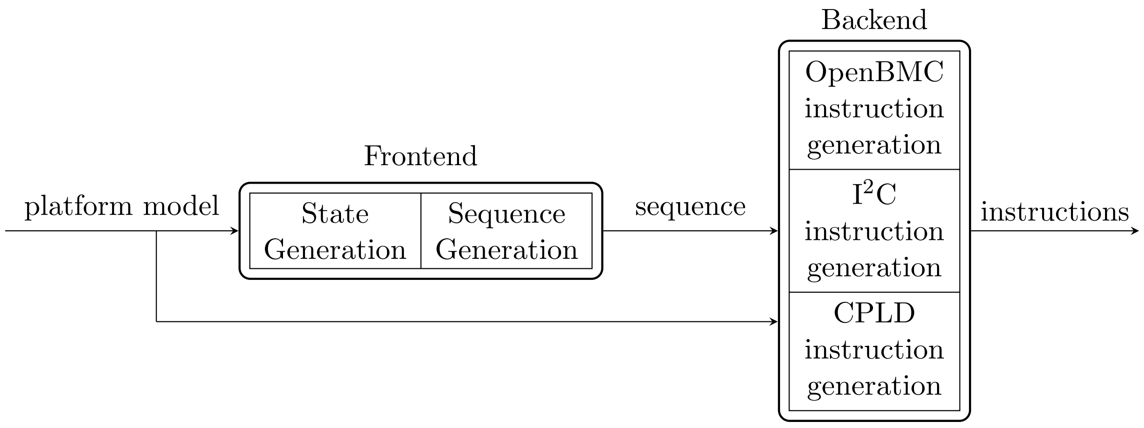


Figure 4.1: Overview over the declarative power sequencing pipeline.

4.3 Summary

In this chapter we looked at our problem of incorporating CPLDs in power sequencing at a high level. We realized that it is sufficient to merely partition a generated power sequence onto all available controllers instead of changing the sequence generation. Having multiple controllers brings its own challenges, however. Our platform description must be able to answer which controller can manipulate which parts of platform state and which controllers can communicate with each other. Further, we settled on an approach where we partition sequences generated by the frontend to determine which controller executes which instructions.

5 Platform Model

In this chapter we devise a model that describes a platform and its devices with the goal to abstract the complexities of hardware for the purpose of generating and manipulating power sequences. We design this model with in mind that it needs to answer the questions we devised in the previous chapter.

To start, we present a very basic platform model without any busses. Then we add in any bus related modifications step by step. Previous work [1, 3, 4] provides a basis for the presented model, and even more so for the basic model. There are however subtle differences, so we find it easier to understand and more convenient to present the complete model instead of merely pointing out differences. Since this work is only concerned with generating code for multiple controllers our model leaves out all things related to demands, constraints and state or sequence generation.

We denote our model in YAML and provide parts of the model with `<placeholders>` for parts of the model described in a different place. Whenever a list is described, we provide a description of an element as the first item in the list. The lists may contain more elements of the described format.

In this work we denote access to structures with a dot notation as it is used by some programming languages, i.e. the value of some field f of a structure s can be represented by $s.f$.

5.1 A basic Model

Here we build a basic platform model without any busses in order to have a baseline model. This section is mostly a summary of [3, section 2].

5.1.1 What is a platform?

Before we can model a platform we need to be sure what a platform really is. The scope of a platform varies depending on the purpose it is serving. For the purpose of power sequencing it is intuitively clear that the platform must contain all devices and wires contained in the power and clock tree.

For our purposes we use a slightly different definition than [1].

Definition 5.1 (Platform)**.** A platform is a collection of *device instances* connected by *wires*.

Since all instances, wires and devices must be described this gives rise to the general structure of our model as already presented in [3].

```
devices:
- <device definition>
platform:
  name: <platform name>
  instances:
  - <instance definition>
  wires:
  - <wire definition>
```

A platform is only useful if it carries some kind of state. For our goal of executing power sequences, we are primarily interested in the state of all wires, as they conduct the power we seek to provide [1]. However, we are also interested in certain internal state of devices and device instances, as they behave differently based on configuration. In fact, we need to sequence certain device configurations explicitly [3]. We use the definition of platform state given by [3, p. 8].

Definition 5.2 (Platform State)**.** A platform state is determined by the states of all devices and wires which comprise the platform.

5.1.2 Devices

Devices are the building blocks of platforms. They serve as an abstraction for the integrated circuits (ICs) on the PCB. However, there is not a one-to-one mapping of a device to an IC as a real IC will often need peripheral ICs or the internal sequencing of an IC is so complex that we model it as multiple devices [3]. Our abstraction should provide a clean interface for the purposes of sequencing.

Different devices serve different purposes. Therefore, we model three distinct device kinds [1, 3]: producers, consumers, and controllers. While these device kinds have differences in their behavior, they do share a common interface: input and output pins. Therefore, we are now able to describe the basic structure of device descriptions:

```
name: <device name>
kind: <device kind>
inputs:
```

```
- <input pin description>
outputs:
- <output pin description>
```

Device Kinds

Producers take usually take some platform resource as input and then provide a platform resource as output. A good example of a producer is a voltage regulator that takes some voltage as input and outputs a different voltage.

Consumers only consume platform resources. They do not have output pins for the purpose of power sequencing. Examples of consumers are CPUs or FPGAs.

Controllers are the producers which coordinate the sequencing of the platform. On Enzian the controllers are the BMC and the two CPLDs. We are able to fully determine their behavior with respect to managing other devices on the platform. In case of the BMC we do this by generating code that is then executed by it. But we might also generate Verilog code and then build a controller fully in hardware. This controller might not be reconfigurable, but we are still able to fully determine its behavior of coordinating the platform. In fact, the only requirements we have on controllers is that we are able to fully control them by generating code and that they implement the interface described in the basic model of this section.

Pins

Pins are the interfaces of devices to the rest of the platform. They are distinguished by names, which are unique on a per-device basis. Further, Pins are typed in the sense that they can only understand and operate one kind of value. There are three signal types [3]:

- voltages of direct current (`kind: dc`),
- frequencies of alternate current (`kind: freq`), and
- logical values of width one or more (`kind: logical`).

With this we can give the structure of pin descriptions.

```
pin: <pin name>
kind: <signal type>
```

For logical pins we also denote the number of bits (width) the logical signal contains.

```
pin: <pin name>
kind: logical
width: <signal width>
```

5.1.3 Device Instances

A platform may contain the same device multiple times. In order to distinguish these, we introduce device instances, or simply instances. Instances have a name unique among all instances and specify of which device they are an instance.

```
instance: <instance name>
device: <device name>
```

5.1.4 Wires

Wires connect multiple instances on a platform. They take the value of its source pin, which is an output pin of some instance, and drive a set of sink pins, which are input pins of some instance, to that value. In that sense wires are typed, as all sink pins must be the same kind of pin as the source pin. Wires have a name unique among wires.

```
name: <wire name>
from:
  instance: <instance name of source pin>
  pin: <name of the source pin>
to:
- instance: <instance name of some sink pin>
  pin: <name of some sink pin>
```

We can consider wires to be directed edges in a graph of pins, where for each wire there is an edge from the source pin to each sink pin. This graph describes the flow of values between pins.

5.1.5 Modelling Example

After presenting this simple platform model, we model a real device, that only needs these basic building blocks. We model the ISL6334 voltage regulator [38] that is used on the Enzian baseboard as a voltage regulator for the DDR power supplies [6].

The ISL6334 has a DC input pin `VCC_DC` supplying it with power and a DC output pin `VOUT` that outputs the programmed voltage. Further, it features two logical input pins for enabling the device `EN_PWR` connected to a controller and `EN_VTT` connected to the power supply for the CPU [38].

A specialty of this device is that the voltage is programmed via eight `VID` pins. Naively, one would model them as a logical pin with `width: 8`. However, this would mean that the model and the sequence have to know the proper conversion between bits and voltages in order to generate a correct sequence. Therefore, we abstract these pins to a DC input pin `VID` which will have the value of the voltage the device should output. With this the details about the conversion from voltages to bits is pushed to the level of device drivers.

These considerations yield the following device model.

```
name: ISL6334
kind: producer
inputs:
- pin: VCC_DC
  kind: dc
- pin: VID
  kind: dc
- pin: EN_PWR
  kind: logical
  width: 1
- pin: EN_VTT
  kind: dc
outputs:
- pin: VOUT
  kind: dc
```

5.1.6 Determining controllability

As we alluded in the previous chapter, we need to be able to determine which controllers are able to make which changes to the platform state. Here we provide a short explanation why this is the case for the presented model.

When controlling the platform we execute instructions that specify which values a controller must change in order to reach a desired change [3]. Given a wire we need to change the value of, we can determine which controller controls the source pin and given a pin we need to change we can follow the wire graph in the opposite direction in order to find out which controller controls the source pin of the connected wire.

Determining whether a controller can read some value works analogously. Given a wire, we look for the controllers that are connected to the wire via a sink pin and given a pin, we determine the wire and do the same thing.

With this we can determine everything we need regarding controllability and observability in this reduced setting.

5.2 A more complex Model

The previous section outlined a model using only "conventional" electric signals. Since most power sequencing nowadays provide some sort of serial interface like I^2C, we need to determine how to integrate the semantics of busses and preserve our ability do fully determine controllability and observability unambiguously.

We will first decide on how to model busses explicitly and then add different notions enabled by busses one at a time.

5.2.1 Busses

Serial busses allow for two-way communication between two devices connected to the bus. In this work we focus on the I^2C bus protocol and its derivatives. We know from (TODO: Reference to background about busses) that I^2C busses are simply two wires connected to multiple devices. Naively we can model them as a wire with its source pin at the bus-master and it is connected to sink pins at any slave device. Further, we assume that a bus-master is always a controller. With this we will always be able to control what changes to the platform state are incurred via bus commands.

Remark. Even though I^2C is capable of multi-master operations [16], we assume that busses in our platform model only ever have one bus-master. If we consider our application domain where we have controllers that manage producers, a multi-master setting does not really make sense.

While it is useful to model the bus connection as a wire, the semantics of busses are vastly different from the wires we described above. If the source pin of a wire is driven to a certain value, then all pins connected to the wire will also take that value and the value will have some kind of effect on the device. Conversely, if a bus-master sends a message over the bus is (usually) is addressed to a specific device. So while all devices connected to the bus can read the message on the bus, the message will only affect the device which has been addressed in the message. Further, while we consider the state of a wire to be part of the platform state, we do not consider the state of the bus wire to be platform state, as the messages change the platform state by controlling a specific device. Additionally, busses break the notion that a wire is a unidirectional flow of information for source to sinks as slave devices reply. This makes any pin connected to a bus both an input and an output pin.

To model busses on the device level, we introduce a new list for bus pins. A bus pin is described by a pin name unique to the device and a list of supported protocols. This list of supported protocol is used for "typechecking" the model. Our model supports multiple bus pins which is required for modelling the BMC on Enzian, which is bus-master for three distinct busses [6].

```
devices:
- name: <device name>
  inputs: <list of input pins>
  outputs: <list of output pins>
busses:
- pin: <bus pin name>
  protocols: <list of supported protocols>
```

Each device instance that has a bus pin will have a different bus address (at least if the instances are on the same bus). Therefore, the instance model needs to be supplemented with a section on busses, that assigns a bus address in hex to every bus pin of the device that is connected to a bus.

```
instances:
- name: <instance name>
  device: <device name>
  busses:
  - pin: <name of bus pin>
    address: <bus address in hex>
```

In order to model the connectivity of the bus and which instance is the bus-master, we abstract the bus as a single wire. The bus-master is the source of the wire and all slave devices connected to the bus are sinks on the wire. While busses are not compatible with the semantics of wires, the information introduced on the device and instance levels is enough such that we only need the connectivity information in order to be able to reason about bus actions.

5.2.2 Monitors

Some devices provide functionality to read a value of some pin or some sensor over the bus. For instance the MAX15301 supports reading the voltage of the input power, voltage and current of the output power it provides as well as the device temperature by using the appropriate PMBus commands [39].

But how do we know if a wire can be monitored over the bus? In order to determine this, we describe in the description of a pin what kind of values can be observed via bus for the given pin. Note that this is only possible if the device is actually connected to a bus.

```
pin: <pin name>
kind: <signal type>
monitor: <list of value types which can be observed>
```

Depending on the signal type different value types can be observed. For DC pins `voltage` and `current` are possible, for frequency pins only `frequency` values are possible and for logical pins we can monitor `logical` values with matching width (raw bits). Of course the possible value types also depend on the capabilities of the monitoring device. However, we do not permit a `binary` monitor type like [3] in order to model power-good pins with a `binary` monitor directly on the output pin. While this might correspond to some PMBus command, we think that the meaning of a `binary` monitor is too ambiguous. Therefore, we either explicitly model a power-good pin (most devices have such a pin anyway) or achieve the functionality through other types of monitors.

Here an example description of the MAX15301 output power pin.

```
name: MAX15301
kind: producer
outputs:
- pin: V_OUT
  kind: dc
  monitor: [voltage, current]
```

This notion allows us to determine what state some bus-master is able to observe. When determining which controllers can observe some wire we can now also determine which controllers can do that via bus. Given some wire we want to observe, we need to find all pins connected to it that have a monitor for the value type logical (raw bits). For each monitor we then determine the bus-master for the bus the given instance is connected to and find another possible observing controller.

5.2.3 Remote Control of Pins

Similarly to monitoring the values of pins via bus, it is also possible to control the output of pins via bus. In the case of the MAX15301 the output power can be enabled and disabled via the `OPERATION` PMBus-command [39]. Again we must settle on how to incorporate this notion into our model in order to be able to determine which pins can be remote controlled by which controller.

To this end we supplement the pin description of remote controlled pins by a field `remote: true`. With this field we know that this pin may be controlled by any of the bus-masters of the busses connected to the device.

When modelling the MAX15301 there are actually two possibilities to model the remote control of the output power. The former makes the output pin a remote.

```
name: MAX15301
kind: producer
inputs:
- pin: PWR
```

```
  kind: dc
  monitor: [voltage]
- pin: EN
  kind: logical
  width: 1
outputs:
- pin: V_OUT
  kind: dc
  monitor: [voltage, current]
  remote: true
- pin: PGOOD
  kind: logical
  width: 1
busses:
- pin: SDA
  protocols: [PMBus, SMBus, I2C]
```

The latter makes the pin `EN` a remote.

```
name: MAX15301
kind: producer
inputs:
- pin: PWR
  kind: dc
  monitor: [voltage]
- pin: EN
  kind: logical
  width: 1
  remote: true
outputs:
- pin: V_OUT
  kind: dc
  monitor: [voltage, current]
- pin: PGOOD
  kind: logical
  width: 1
busses:
- pin: SDA
  protocols: [PMBus, SMBus, I2C]
```

While these two possibilities have the same outcome, the second actually breaks with our notion of how we control a pin. If we were to model the enable pin as a remote, the bus-master might set it to some value, but the value of the enable line will not have changed. Therefore, such an approach would lead to inconsistencies with the driving pin and possible observers of the enable line. For that reason we only ever model a pin as remote if the effect of the remote control is a change in the value of that pin which is always an output pin.

5.2.4 Device Configuration

Devices like the MAX15301 allow even more configuration over the bus. The list ranges from switching frequency over temperature limits to timing configuration [39]. Of course such configurations can only occur once a device is powered. Therefore, the sequence generation explicitly places such configurations in the sequence (TODO: reference to configuration instructino specification) in order to ensure these sequencing requirements.

Since such configurations occur via bus, the only possible controllers to perform these actions are bus-masters of connected busses.

5.2.5 Internal State

Some parts of device configuration are even more relevant. An example of this is the configuration of the output voltage as it is possible via bus on the MAX15301. The value we configure is essential for the correctness of the sequence which is why it is incorporated into the model of a device. The value of the output voltage in the MAX15301 is either read from its internal EEPROM or it is configured via bus [39]. This value is part of the internal state of the device. So important in fact that we will model it explicitly in our device model [3].

An internal variable of a device has a name unique among variables of a device. The value it stores is typed with one type out of the signal-types (`dc`, `frequency`, `logical` with width). Internal variables also take a default value, which is either a value of its type or whatever the device happens to have as default value, which we denote with `default`. Using `default` as default value is useful either when we do not care what the initial value is because we will overwrite it anyway.

```
internals:
- name: <variable name>
  kind: <variable type (one of [dc, freq, logical])>
  default: <default value>
```

Just like ordinary device configuration, writing to internal variables can only be performed by bus-masters of busses connected to the device.

5.2.6 Alerts

Another piece of device configuration especially interesting for our purpose is the configuration of fault levels. Most bus capable devices like the MAX15301 have the capability to send out an alert if the values for certain values, like input or output voltage for instance, are out of bounds. These values are again determined by the sequence generation and the corresponding configuration instructions are to be found in the sequence.

While fault handling itself is outside the scope of this work, we do need to model how the warning levels for alerts are configured as the CPLDs on Enzian need these in order to configure over- and undervoltage thresholds on the monitor pins.

There are three possibilities for programming warning levels: they can be configured by bus commands, in case of the CPLDs they can also be complied into the initial pin configuration, and they might already be preprogrammed, as would be possible with the MAX15301s EEPROM memory.

We model the alert capability by supplementing the pin description with an `alert` field. On the right side of the field we specify how the warning levels are programmed using one of `bus-programmed`, `compiled`, or `pre-programmed`.

```
- pin: <pin name>
  kind: <signal type>
  monitor: <possible value types to monitor>
  remote: <true|false>
  alert: <bus-programmed|compiled|pre-programmed>
```

In case of a bus-programmed alert, just like with all bus actions, only bus-masters of busses connected to the instance may execute such a configuration.

5.2.7 Controller Communication

Since we will have to answer questions regarding which controllers are able to communicate, we need to model this explicitly as the current model is not able to provide this information.

The notion of two controllers communicating is necessary when there is need to pass information between them in order to enable the cooperative execution of a power sequence. Our model does not define a specific format for these communications. The format needs to be determined for every controller separately in the code that is generated. The model only cares that these communications are actually possible. In case of the CPLDs we will see that one bit actually suffices to provide this information.

We model communication among controllers as a directed graph where the nodes are controller instances and an edge from controller c to another controller c' denotes that c can communicate information to c'. Note that we do not allow loops in the communication graph as communications of a controller with itself would not require any messages, even though such a communication is obviously possible and thus require $c \neq c'$.

We describe this graph in the platform description by listing the edges in the graph.

```
name: <platform name>
instances: <list of instance definitions>
wires: <list of wire definitions>
controller-communication:
- from: <controller instance name>
  to: <controller instance name>
```

We define the following predicate for convenience:

Definition 5.3 (Controller Communication)**.** Let c and c' be two distinct controllers on a platform p. Let $G = (V, E)$ be the controller communication graph for p. We define the predicate comm(c, c') to be true if and only if the edge (c, c') is an edge in G, i.e.

$$\mathrm{comm}(c, c') :\Longleftrightarrow (c, c') \in E.$$

One possible issue when dealing with controller communication is if the communication is not bidirectional, i.e. for two distinct controllers c, c' not both comm(c, c') and comm(c', c) are true. In that case the controller that is able to send must be able to observe at least as much of the platform as the controller that is only able to receive. This is the case for the BMC which is able to send information to the CPLDs but the CPLDs are only able to receive communications. But the BMC is able to observe everything the CPLDs are able to observe, because it can monitor all the input pins of the CPLDs over the bus. For the purpose of this work we assume that this property holds for all platforms.

5.3 An Example Platform

In figure 5.1 on the next page we have the drawing of a small Enzian-like platform we are going to use as an example in this work. In order to stay somewhat coherent with Enzian naming we will call it "Gamserrugg". It is designed to show problems that appear on real world systems like Enzian in a smaller setting.

The yellow boxes are consumers, the green boxes producers, and the red boxes controllers. Each signal-type of wire has its own color as well. Orange wires have type `dc`, green wires type `freq`, and blue wires type `logical` with `width: 1`. The red wire is a bus with the `bmc` as bus master.

The full model for Gamserrugg can be found in Appendix A on page 62.

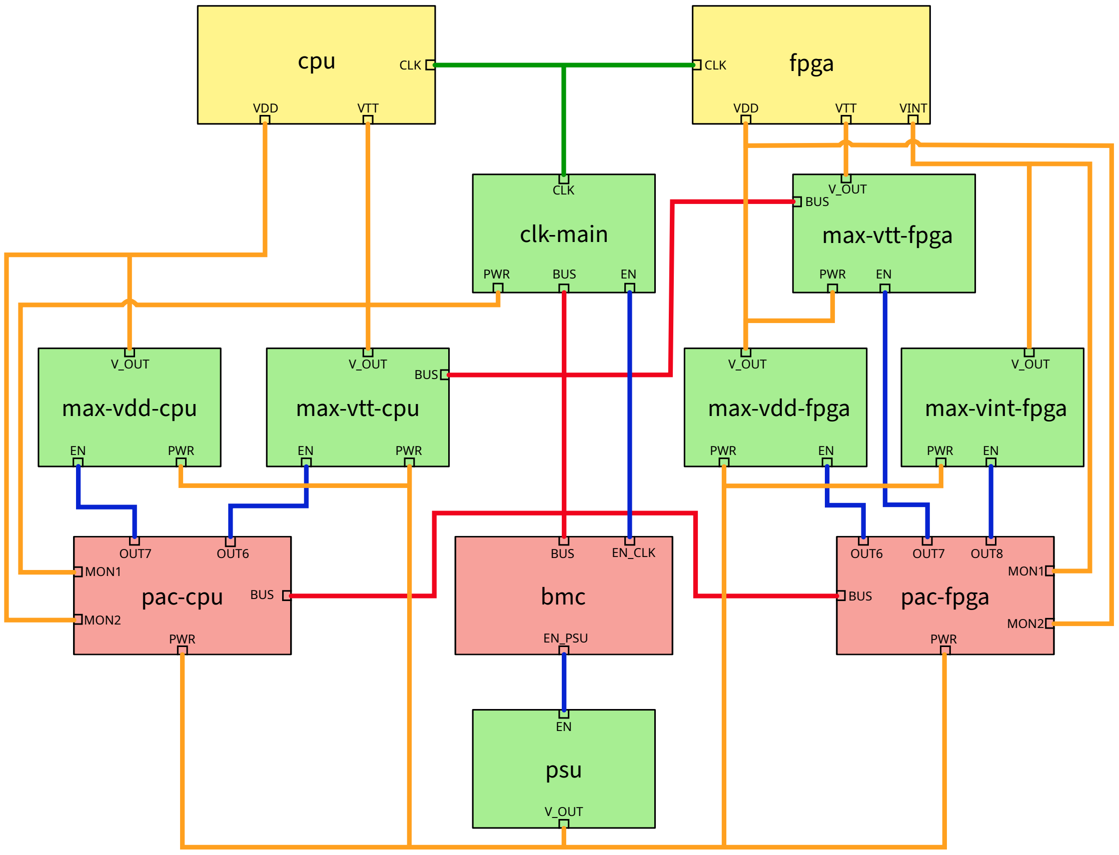


Figure 5.1: Schematic of the Gamserrugg platform.

6 Power Sequences

In this chapter we present the structure and semantics of power sequences based on the previously established platform model. Using the semantics for power sequences, we then explore how we can partition the execution of sequences among multiple controllers.

6.1 Sequence Model

In this section we present a model for the power sequence we receive from the sequence generation in the frontend. Most notable for the goal of this work is that this sequence is agnostic about which controller might execute some instruction in the sequence. The main information the sequence provides is the relation between instructions established by their execution times.

A power sequence as described informally in [3] takes a platform from a starting state to a target state by executing a number of instructions in a given order. This leads to the conclusion that instructions must operate on platform states. Further we observe that instructions in such a sequence either modify or observe some part of the platform state.

Definition 6.1 (Instruction)**.** An *instruction* describes an action to modify or observe platform state that may be executed by some controller on the platform.

With this definition we can go on to define power sequences.

Definition 6.2 (Power Sequence)**.** A *power sequence* is a set of instructions which have an order that determines when they may be executed. The execution of a sequence in the defined order takes a platform from an initial platform state to a target state.

Note, that this definition does not specify the order of instructions, only that there needs to be an order.

From definition 5.2 we know that the platform state is composed of the states of platform components. The set of valid platform states and state transitions is limited by constraints defined in the model for the purpose of state and sequence generation [3]. For a sufficiently large platform (e.g. Enzian) it is likely to happen that the effects of two state transitions in different parts of the platform do not interact with each other (e.g. enabling some power line for the CPU and the FPGA on Enzian respectively). The corresponding instructions may thus be executed at the same time if the sequence generation deems this safe [3]. To take this into account, we organize instructions into steps.

Definition 6.3 (Step)**.** A step is a set of instructions operating on independent parts of the platform such that they may be executed at the same time. It is referred to by some $n \in \mathbb{N}_1$.
A step contains two substeps: a set of write instructions writes(n) that modify platform state and a set of read instructions reads(n) that observe platform state.

Remark. We denote the set of natural numbers excluding 0 as $\mathbb{N}_1 := \mathbb{N} \setminus \{0\}$.

We can think of the read substep and read instructions in general as barriers that keep controllers on lockstep and from running a step ahead.

Further, we deliberately model all instructions as either a read or a write instruction because at their core reading and writing data is what they actually do. Even waiting for a given time can be viewed as reading a time from a timer.

6.1.1 Instruction Structure

The definition of instructions (definition 6.1) thus far leaves out details on what information the instructions actually carry. From definition 6.3 we know that all instructions are organized in steps and that there are two types of instructions, namely reads and writes. Instructions always concern a specific component of the platform like a wire or a device instance. We call this the endpoint of an instruction. Obviously, instructions must also carry information what the expected value of a read is or what value is written. This yields the following structure of an instruction.

```
instruction: <read|write>
step: <step number>
endpoint:
  <endpoint description>
# for reads
read: <read value>
# for writes
write: <write value>
```

Endpoints Instructions read or write information from or to a specified component of the platform. We call this the endpoint of an instruction. Below, we describe all possible endpoints in a instruction.

The `wire` endpoint is one of the most commonly used and refers to a single wire by its name. The usage of a wire endpoint is only permitted if the wire exists.

```
endpoint:
  wire: <wire name>
```

The `pin` endpoint refers to a pin on a given instance. Using this endpoint in an instruction is only permitted if the pin on the named instance exists. It is either connected to a wire or the operation can be done with a bus command. In case of a write instruction this would be a remote and in case of a read instruction a monitor on the specified pin

```
endpoint:
  instance: <instance name>
  pin: <pin name>
```

The `var` endpoint enables the modification of internal state on instances. The usage of this endpoint is only legal if the variable to be modified exists on the specified instance and the instance is connected to a bus.

```
endpoint:
  instance: <instance name>
  var: <variable name>
```

The `device-instance` endpoint allows for modification of the internal state of the specified instance. The usage is only permitted if the specified instance exists and is connected to a bus.

```
endpoint:
  instance: <instance name>
```

The `timer` endpoint is used to model the wait instruction as a read. It refers to the ability of a controller to pause its execution for a set amount of time. This is usually achieved by a wait instruction of the controller. We assume that every controller has this capability as we could even implement this in hardware with an oscillator and a counter.

```
endpoint:
  timer: # deliberately empty
```

Read Instructions The different types of possible read values corresponds with the possible monitor types. In fact, the instruction is only valid if the read value matches its monitor type. All read instructions have in common that they prevent the sequence from advancing as long as they were not able to successfully read their specified read value.

All read values for some electric characteristic (voltage, current, frequency) follow the same scheme. Since devices have certain tolerances, we want to ensure that endpoints of these signal-types are within a minimum and a maximum value. If that is the case the read is successful.

```
read:
  min: <minimum value> <unit>
  max: <maximum value> <unit>
```

Note that the read value for these types is only differentiated by the units. Any prefix of the proper units is accepted, e.g. `4.2 V` as well as `0.0042 kV` is acceptable for voltages.

To read values with logical signal type we ensure that the value we monitor is equal to the value specified in the instruction. The instruction is only valid if the width of the endpoint matches the specified width. The read is successful if the monitored value is equal to the specified value.

```
read:
  value: <hex value>
  width: <number of bits in value>
```

A special read value is the duration. With this we model a wait instruction as a read of a clock. An instruction with a duration read value may only read a `timer` endpoint. In fact, all instructions for a `timer` endpoint without a duration read value are invalid. The read is successful if the specified amount of time has passed since invoking the instruction.

```
read:
  duration: <duration> <time unit>
```

The endpoints `var` and `device-instance` cannot be read from. Technically, this is possible with some PMBus commands, but this capability is currently not used by declarative power sequencing.

Write Instructions Write instructions modify the platform state. Unlike read instructions, the end of a write instructions is not known unless its effect is observed, which is what read instructions are for.

We set values of type DC, frequency or logical with the corresponding write instructions described below. These write instructions are valid with endpoints of type `wire`, `pin`, and `var`. The logical write value may not contain more bits than the width of the endpoint.

```
- write:
    dc: <voltage> <voltage unit>
- write:
    freq: <frequency> <frequency unit>
- write:
    logical: <hex value>
```

In order to set canonical device configurations which are also relevant for the sequence generation we need a special instruction. It is only valid with `device-instance` endpoint.

```
write: configuration
```

The warning-level write value sets alert levels on pins. Currently, it is only valid with endpoints of type `pin` where the pin has type DC. After this value has been written, the device will produce an alert if the value of the pin wanders outside of the interval [`min`, `max`]. We have voltage warning-levels because clock regulators usually use a phase lock indicator (frequency analogue to a power-good pin) instead of warning-levels. Other power related warning-levels like current are not used because currently voltages are the only quantity the model imposes constraints on [3].

```
write:
  warning-level:
    min: <voltage> <voltage unit>
    max: <voltage> <voltage unit>
```

Which instructions can be executed on which endpoints? In order to summarize the text above, table 6.1 shows which instructions are allowed to be paired with which endpoints in an instruction.

instruction \ endpoint	`wire`	`pin`	`var`	`device-instance`	`timer`
write `dc`	t	t	tb	x	x
write `freq`	t	t	tb	x	x
write `logical`	t	t	tb	x	x
write configuration	x	x	x	b	x
write warning level	x	tb	x	x	x
read `dc`	t	t	nb	x	x
read `dc`	t	t	nb	x	x
read `dc`	t	t	nb	x	x
read `duration`	x	x	x	x	✓

Table 6.1: Under which conditions may an instruction be executed on an endpoint.
x: not allowed, ✓: always allowed, t: allowed if signal types match, b: allowed with bus connection, n: possible but not used

6.1.2 Execution

Now that we know what instructions look like we concern ourselves with the question of when a given instruction may be executed by some controller. Note a power sequence as defined above is agnostic about which controllers may execute an instruction.

The definition of power sequences (definition 6.2) already alludes to the fact that instructions need to have an order. In the following definition we use the facts that some step n is executed after another step m if $m < n$ and that writes are executed after reads as defined in definition 6.3.

Definition 6.4 (Instruction Order)**.** For two distinct instructions i_1, i_2 we define that i_1 is executed before i_2, i.e.

$$\begin{aligned}&i_1 \prec i_2 :\Longleftrightarrow \\ &\qquad i_1.\,\mathrm{step} < i_2.\,\mathrm{step} \lor i_1.\,\mathrm{step} = i_2.\,\mathrm{step} \land i_1 \in \mathrm{writes}(i_1.\,\mathrm{step}) \land i_2 \in \mathrm{reads}(i_2.\,\mathrm{step}).\end{aligned}$$

Note that instructions in the same substep are not comparable in this relation. Instructions which are not comparable may be executed at the same time.

This order is a proxy for sequencing constraints on the platform, as the order is defined by the sequence generation which itself depends on the sequencing constraints to produce a valid sequence. Therefore, we do not need any specific information on sequencing constraints in order to operate on a generated power sequencing – all the required information is already encoded.

The question we need to ask now is how can we ensure that we obey this ordering on instructions when executing a sequence. Intuitively, we need to ensure not to start executing some instruction before all preceding instructions are done executing. So the question becomes: How can we tell that some instruction is done executing?

The easiest way to determine that an instruction is done with its execution, is to observe its effect. For read instructions this is trivial, as their effect is observing platform state. Therefore, a controller has observed the effect of the read instruction when it has received the information obtained from the read instruction. For write instructions this is more involved. The effect of some write instruction w is that the endpoint w.endpoint takes on the value of w.write. Actually, for some endpoints like wires many more endpoints on the platform take on this value. The effect of a write instruction is either not observed (e.g. configuration) or it is observed by a read instruction. But since read instructions are blocking until they have successfully read the expected value, we may start the execution of reads(n) immediately the execution of all instructions in writes(n) in some step n has started. If a step does not contain any reads, that is, all writes were just configuration instructions, we just continue with executing the next step after the execution of all write commands has started. While this might seem reckless, remember that the sequence generation did not impose any further constraints on this ordering of the sequence.

It follows that some step may only start execution once it has observed the effects of all reads in the previous step (common case) or if all writes in the previous step have started their execution (rare case).

Instructions need to be executed by some controller on the platform. But not all controllers are able to execute all instructions. In our description of the platform model we paid special attention that we are able to deduce which controller is able to control or read the value of some endpoint on the platform.

Definition 6.5 (Possible Executors)**.** For some instruction i we define the set

$$\text{possible}(i)$$

to contain all controllers which are able to control or observe the platform in order to achieve the effect of i.

With that we know when and where a given instruction may be executed.

6.2 Partitioning Power Sequences

The goal of this work is to execute a power sequence while leveraging multiple controllers at the same time. In order to achieve this we partition the sequence among the available controllers. That is we assign instructions to controllers for execution.

As per usual when executing things in parallel, problems are bound to pop up. Before we come up with an algorithm to partition power sequences, we first must understand how the execution of a power sequence is affected by multiple executors. We then take these findings into account when devising the sequence partitioning algorithm.

When looking at the Enzian platform, we realize that the power tree for the CPU and FPGA sides are almost but not quite independent. When looking at sequences generated from [3] we see the CPLDs rarely have long streaks of instructions independent of the rest of the platform. Our conclusion from this observation is that an execution model where the CPLDs control independent "subplatforms" does not suffice. Therefore, we use a cooperative execution model where all controllers work in lock step on the same sequence with instructions executed by different controllers. For this cooperative execution model we enforce the following rules:

- Every controller is executing instructions from the same substep in a sequence.
- Every controller that executes a write instruction in a step must observe all effects from the previous step.
- If a controller cannot observe all effects from the previous step another controller that can may communicate to it that all effects of the previous step have been observed.
- Reads may be executed as soon as the preceding step on the controller has been completed.

Note that the last rule states that reads multiple steps ahead may already be executed when the current step has completed. This is possible because reads are blocking execution on the controller until they are able to observe the expected value.

Our rules rely heavily on the observation of effects from the previous step. The following definition specifies which controllers are able to observe reads in the previous step.

Definition 6.6 (Complete Observers)**.** We call some controller c a *complete observer* (also: complete controller) in step $n \in \mathbb{N}_1$ iff c is able to observe all reads in step n, i.e.

$$c \in \text{complete}(n) :\Longleftrightarrow \forall i \in \text{reads}(n).\ c \in \text{possible}(i).$$

Observing writes in the previous step is only possible for the executing controller. Therefore, if there are no reads in a step, all controllers executing writes must communicate to all possible controllers that they are done executing the writes assigned to them.

Based on the rules above our controller choice for write instructions is rather unconstrained as any possible controller on a well-designed platform is able to observe the effects of the previous step or if it is not able to observe the previous step it is able to receive a communication from another controller. This gives us tremendous freedom when choosing executing controllers for write instructions and allows us to apply strategies to make this choice.

Such strategies are merely preferences for controllers with certain characteristics. Applying a strategy must not make the partitioning of a sequence impossible. In this work we will implement three different strategies. The *centralize* strategy prefers the central controller (in case of Enzian the BMC) whenever it is available. The *distribute* strategy prefers non-central controllers (in case of Enzian the CPLDs) whenever possible. The *distribute without communication* strategy prefers non-central controllers that do not need additional communication. This means that for writes we prefer complete controllers. All strategies prefer controllers that do not need additional communication as a secondary preference.

This freedom in assigning write instructions to controllers then becomes a constraint when thinking about assigning read instructions. So much so that our partitioning algorithm will assign reads in a separate pass. In this pass we assign all read instructions of a step at once, since our rules require all controllers executing reads to be complete observers. When assigning the reads, we make sure to only assign reads to controllers where the information produced by reads is actually used, i.e. the controller is a writer in the next step or communicated the information to another controller.

As mentioned above, if a controller that is not a complete observer in the previous step is a writer in the current step then this “incomplete” controller needs to receive a communication from a complete controller when it has observed all effects of the previous step.

To summarize we outline our partitioning algorithm here:

1. Assign each write instruction in the sequence to one controller according to the chosen strategy.
2. In a second pass, assign all read instructions of a step to all controllers where the information produced is needed.
3. Insert communications in steps where they are needed based on the choices in the previous two passes.

The following subsections describe each step in this algorithm in detail.

For convenience, we denote the set of controllers executing write instructions in step n with writers(n) and denote the set of controllers executing read instruction the same step with readers(n) respectively.

6.2.1 Write Assignment

When assigning write instructions to a controller, we iterate over all write instructions. Consider some write instruction i in step n.

Any controller that may execute i must be able to execute i and it must be able to either observe all reads in the preceding step or it must be able to communicate with a complete controller of the previous step. Mathematically, a controller c is a candidate for executing i if

$$c \in \text{possible}(i) \wedge (c \in \text{complete}(n-1) \vee \exists c' \in \text{complete}(n-1).\ \text{comm}(c', c)).$$

From this set of candidate controllers we can then choose one controller to execute i according to the chosen strategy.

6.2.2 Read Assignment

When assigning reads we iterate over all steps in a sequence. Consider some step n. We assign all instructions in reads(n) to as many controllers as are needed to supply the controllers in the next step to continue the execution of the sequence.

First, we find a set of candidate controllers for the read instructions. A candidate must be a complete observer in step n and in the next step it must either use the information obtained by the reads itself (i.e. it is a writer in the next step) or it passes on the information to some other controller that needs the information in the next step. Mathematically, a controller c is a candidate for executing all writes in step n if

$$c \in \text{complete}(n) \land (c \in \text{writers}(n+1) \lor \exists c' \in \text{writers}(n+1).\ \text{comm}(c, c')).$$

In order to find all necessary controllers we instantiate the set information-needed with all writers of the next step, i.e. writers$(n+1)$. Then we iteratively perform the following procedure until either information-needed or the set of candidates is empty.

In the iteration we choose one controller c according to the chosen strategy form the set

$$\{c \mid c \in \text{candidates} \land (c \in \text{information-needed} \lor \exists c' \in \text{information-needed}\,.\ \text{comm}(c, c'))\}.$$

Then we remove the chosen controller c from the set of candidates and information-needed. Further, we remove all controllers c' from information-needed where comm(c, c') holds.

After the iteration the set information-needed must be empty. Otherwise, it might not be possible to partition the given sequence. Then we assign all instructions in reads(n) to all the chosen controllers.

6.2.3 Communication Insertion

The insertion of communication can be performed in the same pass as read assignment and right after the assignment for a given step is done.

Intuitively, we need to insert a communication for every time we have removed a controller from information-needed in the iteration for determining read controllers based on the fact that the chosen controller can communicate with it. However, we also need to insert communications to all writers in the next step if the current step did not contain any read.

Concretely, we need to insert a communication from a controller c to another controller c' in step n if they are able to communicate, c' is a writer in the next step and either there were no read instructions in this step and c was a writer, or otherwise c was a reader in this step and c' was not. Mathematically, the communication between c and c' in step n is inserted if

$$\begin{aligned}\text{comm}(c, c') \land c' \in \text{writers}(n+1) \land \big(&\\ &\text{reads}(n) = \emptyset \land c \in \text{writers}(n)\\ &\quad \lor \text{reads}(n) \neq \emptyset \land c \in \text{readers}(n) \land c' \notin \text{readers}(n)\big).\end{aligned}$$

While this step is called “communication insertion” we do not actually insert instructions for communication between controllers into the sequence. The actual implementation of the communication depends heavily on the different devices of the controllers and is delegated to the device driver level. We only note for each step which controllers send a communication to which other controllers.

While the low-level implementation of communications is delegated to a lower level the high-level implementation is always the same. If controller c sends a communication to c' in step n then c will perform a write in step $n + 1$ to send the information and c' will perform a read in step n to receive the information. With this mechanism we ensure that receiving controllers do not continue to the next step before they have received the communication.

7 Implementation

In this chapter we describe how we implemented the ideas of this work into a working program which partitions a sequence and outputs code for the BMC and the CPLD.

Our program is a CLI written in the Rust programming language. In a first step we modelled the device and sequence model as Rust data-structures and parsed them from YAML files. With these models as a basis we implemented the partitioning algorithm. The implementation of which neatly derives from the formulas presented in the previous chapter by transforming these to set operations. After this followed the implementation of the code generation for specific controllers, which is much more interesting.

The first step in code generation for any controller instance is to transform the frontend instructions assigned to the particular controller and transform it into a sequence of concrete instructions with information on how and where the instructions are executed (see listing 7.1). For certain instructions there is more than one possibility for how that instruction might be executed by the chosen controller. Our implementation generally made the choice resulting in less bus operations. Further, the communications must be converted to concrete instructions.

7.1 Communication between BMC and CPLD

In order to facilitate communication between two controllers, the sending party needs to be able to send information to the receiving party. The CPLDs on Enzian have three unconnected logical input pins `IN[4..6]`, which can be configured to receive information from the I^2C register (see figure 7.1 on the following page). With this we have three bits of information the BMC can use to pass information to the CPLDs. Unfortunately, the CPLDs have no possibility of communication to the BMC as they do not have any spare output facilities. Technically, we could use the voltage or trim outputs, but we decided against that because the communication from the CPLDs to the BMC is not strictly necessary as the BMC can observe all inputs and outputs on the CPLDs via bus and therefore our assumption on one-sided communication holds.

With these three bits at our disposal we devise a small communication protocol. We use the pin `IN4` as a kind of clock where a clock edge signifies a communication. When starting the sequence the BMC will set the pin to 1. Whenever the CPLD expects to

```rust
enum ConcreteInstruction<'a> {
    ReadPin {
        step: usize,
        pin: PinName<'a>,
        value: ReadValue,
    },
    ReadBus {
        step: usize,
        bus: PinName<'a>,
        value: ReadValue,
        from_instance: InstanceName<'a>,
    },
    WritePin {
        step: usize,
        pin: PinName<'a>,
        value: WriteValue,
    },
    WriteBus {
        step: usize,
        bus: PinName<'a>,
        value: WriteValue,
        to_instance: InstanceName<'a>,
    },
    Wait {
        step: usize,
        duration: Time,
    },
}
```

Listing 7.1: Rust representation of a concrete instruction.

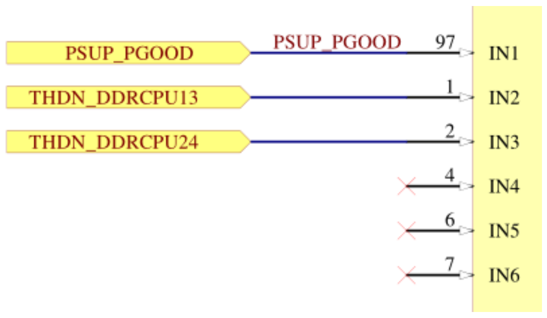


Figure 7.1: Unconnected input pins `IN[4..6]` of the CPLDs on Enzian [6] used for receiving communication from the BMC.

receive a communication that the effects of some step have been observed, it waits for the inverse of the value previously set by the BMC. So in the initialization the CPLD waits for `IN4`, which is referred to by its alias `STEP_CLK` in the generated code, to have the value 1. In the first communication it receives it then waits for `NOT STEP_CLK` (i.e. 0, the inverse of the previously set value 1). In the second expected communication it waits for `STEP_CLK` and so forth. Therefore, a communication in some step n becomes a write instruction in step $n+1$ of the correct "clock" value for the BMC and a read instruction for the same value on pin `STEP_CLK` in step n for the CPLD.

We use the remaining pair of bits `IN[5..6]` to encode four different pieces of information. Recall that the ISPPAC can have multiple state machines (2.3.3) and that a platform can have multiple sequences like a powerup and a powerdown sequence. Thus, we use the bitstring `0b01` signify the powerup sequence and `0b10` the powerdown sequence. Each sequence would have its own state machine and we would only execute the state machine that received its corresponding code during initialization. The remaining two bitstrings could be used for potential error conditions or other sequences.

7.2 Code Generation for CPLDs

The first problem when generating code for the ISPPAC CPLDs is that the design software PAC-Designer does not take any file input except its proprietary XML-based `.PAC` format. Since this work is supposed to be about computer systems and not the automation of GUIs, we opted to simply mimics the format of an export which includes almost all information needed to fully program the CPLD and supplement information that is not in the original export format with the goal that the output can be taken as instructions to program the device in PAC-Designer.

When programming the CPLD we not only need to describe the program we also need to fully configure all pins of the device. The voltage monitoring pins have to be configured with the appropriate warning levels if these are declared to be compiled in the platform model. We need to take special care of the pin `VMON1` which has a voltage divider in order for it to be able to monitor voltages higher than the maximum voltage rating of the pin. Therefore, we need to scale down the warning-levels for this pin according to the voltage divider in the schematics [6].

The logical input pins can either take their input from the connected wire or from an I^{2}C-register. We configure `IN[1..3]` to take their input from the respective wire and pins `IN[4-6]` to take their input from the I^{2}C-register. The logical output pins can either be controlled by the CPLD or an I^{2}C-register. By default, we configure the pins to be controlled by an I^{2}C-register. For all pins that are used in the concrete sequence executed by the CPLD we change this configuration such that they are controlled from the CPLD.

Since the ISPPAC only has four timers and thus four possible times for wait instructions, we need to make sure that wait instructions executed on one ISPPAC do not collectively need more than four distinct times. If there are more than four distinct times needed we look for the two closest times and remove the smaller of the two. Here we use an assumption that wait instructions in power sequences seek to achieve that the execution is paused for *at least* a set amount of time (e.g. wait until the output voltage of a regulator has stabilized) and that we therefore can increase the wait time and still have a correct power sequence. We repeat this "contraction" of wait times until we are left with four distinct times.

Generating LogiBuilder code for the CPLDs using concrete instructions is fairly straight forward. First, we use a different state machine for each sequence we want to execute. We initialize each execution in a state machine by waiting for the pins `AGOOD` and `IN4` to be high and the pins `IN[5..6]` to be equal to the bitstring of the sequence we want to execute. Each read instruction is emitted as a `Wait For <boolean expression>`, write instructions as an assignment instruction `<pin> = <value>`, and wait instruction produces a wait-for-timer instruction `Wait for <duration> using Timer<[1-4]>` to pause the execution for a set amount of time. The duration in this instruction is predetermined by the duration that is programmed into the timer in the pin configuration.

With all these transformations, we are able to generate programs for the ISPPAC CPLD. An example sequence generated by our implementation can be found in listing 7.2.

```
============================================
CPLD logiBuilder
--------------------------------------------
State Machine 0
PscProgram

  ProgramSize = 20

   Step 0   Begin Startup Sequence
   Step 1   Wait for AGOOD AND (STEP_CLK AND NOT IN5 AND IN6)
   Step 2   Wait for NOT STEP_CLK
   Step 3   EN_UTIL_3V3 = 1
   Step 4   Wait for STEP_CLK
   Step 5   EN_VCCINT_FPGA = 1
   Step 6   Wait for NOT STEP_CLK
   Step 7   EN_VCCINTIO_BRAM_FPGA = 1
   Step 8   Wait for STEP_CLK
   Step 9   EN_VCC1V8_FPGA = 1
   Step 10   Wait for 3 ms using TIMER1
   Step 11   Wait for VCC1V8_OK
   Step 12   EN_SYS_2V5_24 = 1, EN_SYS_2V5_13 = 1, EN_SYS_1V8 = 1, EN_VADJ_1V8_FPGA
   ↪  = 1, EN_VDD_DDRFPGA24 = 1, EN_VCCINTIO_BRAM_FPGA = 1
   Step 13   Wait for NOT STEP_CLK
   Step 14   EN_MGTAVCC_FPGA = 1
   Step 15   Wait for MGTAVCC_FPGA_OK
   Step 16   EN_MGTAVTT_FPGA = 1
   Step 17   Wait for STEP_CLK
   Step 18   EN_MGTVCCAUX_L = 1, EN_MGTVCCAUX_R = 1
   Step 19   Halt
--end-of-PscProgram
```

Listing 7.2: Generated program for the pac-fpga CPLD on Enzian. Partitioning was performed using the strategy *distribute*. The highlighted steps contain the program generated with the partitioning strategy *distribute without communication*.

8 Evaluation

In this chapter we evaluate the implementation by demonstrating its abilities to partition power sequences for Enzian by examining the partitions generated by the different strategies.

8.1 Setup

For this section we use the sequence generated by the code for [3][1]. This sequence and the corresponding sequence model is then translated into the YAML-format we described in this work. Some adjustments in the modelling had to be made in the modelling of the platform.

- Busses are now explicitly modeled as wires connecting to bus pins.
- All bus related pin functions like monitors, alerts and remotes were adjusted remodeled to conform to the new model.
- The monitoring device INA226 is now modelled as a consumer instead of a controller.
- The PSU was remodeled to reflect the fact that it is one device with multiple output cables.
- The clock regulator SI5395 was modeled closer to the datasheet, especially for the monitoring of the phase lock.
- The voltage regulator ISL6334 was remodeled as explained in section 5.1.5.

The translated model and sequence were then both fed into our sequence partitioning implementation described in the previous chapter. We partition the sequence with all three implemented strategies centralize, distribute, and distribute without communication (explained in section 6.2).

[1]Command: `$ ./find-seq.scm -d enzian-desc -p enzian power-on-both`

8.2 Partitioned Sequences

In this section we analyze the partitioned sequences generated for each strategy. The table 8.1 shows the number of instructions each controller executes when the sequence is partitioned using a given strategy. Note that for 12 instructions on the two CPLDs it is debatable whether they constitute "real" instructions. These are actually the instructions that configure the warning levels for the 12 voltage monitoring pins on the ISPPAC. These are assigned to the CPLDs in every strategy because we modelled all voltage monitoring pins with `alert: compiled`. In order to be thorough and transparent we indicate the number of instructions that entail an actual execution in parentheses.

Strategy:	Centralize	Distribute	Distribute w/o communication
`bmc`	88	66	79
`pac-cpu`	12 (0)	21 (9)	15 (3)
`pac-fpga`	12 (0)	29 (17)	22 (10)
Communications	0	8	0

Table 8.1: Number of instructions executed on each controller for a sequence partitioned by a given strategy.

For the partition generated using the centralize strategy we find that all instructions are assigned to the central controller, which in case of Enzian is the BMC. This corresponds directly with the current implementation where the BMC remote controls all pins of the CPLDs via bus commands in order to be able to execute all instructions.

The partition which was computed using the strategy distribute turned out as advertised. As table 8.1 shows a significant amount of instructions that were able to be offloaded to the CPLDs. The generated code for the controller `pac-fpga` can be found in listing 7.2. The partition generated using the strategy distribute without communication was still able to offload some instructions, but much less. Actually, the distributed partitions without communication are always contained in the distributed partitions with communication. In listing 7.2 the highlighted steps 10, 11, 12, 15, and 16 contain the 10 instructions which are assigned to `pac-fpga` according to table 8.1.

Notice, that the instructions on the CPLDs when partitioned with the strategy distribute without communication are always independent subsequences, which can be observed and executed entirely be the CPLDs. Thus, these independent subsequences can be found in the partitions generated with the distribute strategy because they always start with the first read instruction of one of the voltage monitoring pins after a communication and end with the last write instruction before the following communication. Ideally, these independent subsequences would be as long as possible to utilize the CPLDs as much as possible independently of the BMC.

The difference between the number of instructions the partitioning algorithm is able to offload with the distributed and the distributed without communication strategy is a measure for how tightly coupled a platform is, i.e. how separate are the CPU and FPGA branches in the Enzian power tree, and how compact the sequence generation generated the power sequence. More tightly coupled platforms result in shorter distributable independent subsequences as these subsequences will be more likely to be broken up by constraints imposed by the coupling. Compact sequences reduce the number of distributable independent subsequences because they “parallelize” the available work and thus insert instructions which cannot be observed by some controller in a given step and communications become necessary. This is exactly what happens in our case with the sequence generated from the code of [3] as minimization of sequence length was one of the goals of that work (see section 3.1).

9 Conclusion

In this chapter we discuss our work and possible future work and summarize what we have done.

9.1 Discussion

Over the course of this work we encountered some caveats and other bits of interest that we will discuss in this section.

9.1.1 Is waiting really a Read?

In this work we modelled wait instructions as read instructions with the reasoning that waiting is just monitoring a timer until the specified time was reached which fits nicely into our model of read instructions. However, when generating instructions for controllers this abstraction quickly fell apart. Our implementation of concrete instructions (see listing 7.1) we added a wait instruction again. This has multiple reasons. While we can model wait instructions as reads they are different from reads in their nature and usually compile to a special wait function on a controller. Further, waits could occur at any point in a step. In fact, we had to assume that waits only should occur as a delay between the write substep and the read substep.

This assumption was made explicit in [4] where wait instructions could be specified as a delay in the read instruction which is quite elegant as it does not suffer from the same ambiguities our approach suffers from. Another more flexible possibility would be to drop the two-instruction-abstraction altogether and introduce wait instructions that specify at what time they are executed.

9.1.2 Bidirectional Communication Problems

Our method for generating instructions from communications has a critical flaw if two controllers are able to communicate in both directions, which is not the case on Enzian. To illustrate consider two controllers c_1 and c_2 with comm(c_1, c_2) and comm(c_2, c_1) and suppose that in step n both controllers need to send a communication to the other. As described in section 6.2.3 this would result in both controllers waiting in step n to receive a message from the other which it would send as a write in step $n + 1$. We have a deadlock!

In order to resolve this problem we need to make sure c_1 and c_2 will not wait on their respective communications at the same time. We could achieve this by requiring that for every pair of controllers on the platform which are able to communicate bidirectionally their communications are executed sequentially. Applying this to our example we could define that the controller with the lower index may send its communication first. With this we insert a virtual step before the original step $n + 1$ into our sequence which only consists of c_1 sending the communication and c_1 reading the communications from c_2, which it sends in step $n + 1$. We call this a virtual step because we do not require other controllers to observe its events as it is perfectly fine for them to continue with execution of step $n + 1$.

The illustrated problem could come up in case a step does not contain any read instructions and the writers of this step all broadcast communications when they are done executing. If two of these controllers have bidirectional communication then we have a deadlock. Our sketched solution might be able to solve the problem, but it might also help if we start verifying changes to device configuration using read instructions. Which this all steps without any reads would be eliminated in the sequences generated for Enzian. Note, that there are other situations where the described deadlock could arise.

9.1.3 Partitioning multiple Sequences

In order for platform controllers to be useful for the purpose of power sequencing, they must be able to execute multiple sequences at different times. For instance, controllers must not only be able to power up the platform, but they also must be able to shut it down gracefully again. The sequence generation already supports specifying different sequences that may be generated. Concretely, we are able to generate a power up and a power down sequence from [3].

However, having multiple sequences at the same time, brings us into trouble when partitioning power sequences for the ispPAC. Not only do we need to know all the sequences at compile time, but we also need to make the same decisions when partitioning all the sequences.

The output pins of the ispPAC can be configured at compile time to either be controlled by the CPLD or by an I^{2}C register, such that the BMC can control it remotely. This configuration cannot be changed at runtime. Thus, we can run into the problem that we partition a power up sequence and make certain decisions on which pins are controlled by the CPLD and that we partition a power down sequence afterwards and settle on a completely different set of pins that should be controlled by the CPLD. Therefore, we would need to ensure that the partitioning algorithm makes the same decision on which controllers control which parts of the platform for every sequence that needs to be programmed for that platform.

However, this constraint only applies to devices where we need to configure from where pins are controlled beforehand. We would have to take care that we do not introduce unnecessary constraints in our partitioning procedure.

9.1.4 Does it pay off to use CPLDs in Power Sequencing?

After doing all this work to enable CPLDs to play an active and autonomous part in controlling the platform while executing power sequences we have to ask whether incorporating CPLDs in an active role actually makes sense. In order to make a judgement to that end we weigh potential benefits with potential drawbacks.

A major advantage of CPLDs is their simplicity. At their core, they are a bunch of logic gates arranged in a useful and reconfigurable way. The fact that CPLDs are programmed by implementing logical formulas in disjunctive normal form leads to hard real-time guarantees on the execution time as the delay logic delay as well as the delay from all the connective wiring is entirely determined by the programming. Thus, even though some timings are not known a priori we are able to estimate a hard upper bound for the runtime.

Further, this simplicity also makes the implementation of a power sequence much easier to understand as it translates directly to logic formulas instead of being build on top of an entire complex software stack like openBMC. The programs are also easy to understand as they either wait for some logical condition to be satisfied and then execute some assignments once they are.

A big downside to CPLDs (at least the ispPAC) is the hassle it takes to reconfigure. In order to flash a new program onto the device we need to access the mainboard, connect the JTAG connector and make sure we followed the manual for the device to a tee. To add insult to injury the process of programming the CPLD is itself really painful as the PAC-Designer program does not take any file or other textual input. So we are left with clicking our generated programs into the PAC-Designer GUI.

But once the CPLD is reconfigured, the parts of the sequence stored and executed on the CPLD cannot be meddled with until we connect a JTAG connector again. This makes the CPLD a secure “enclave” where parts of power sequences are safe from vulnerabilities of the BMC.

In order to take as much advantage as possible from this “enclave” we would want to execute sequences independent of the BMC which are as long as possible. But as we saw in the previous chapter, highly coupled platforms and compact sequence generation will diminish returns on this front.

A major consideration for computer systems generally is their complexity. By including CPLDs in the execution of power sequences we undoubtably increase the complexity of the platform and the declarative power sequencing pipeline. This work is proof of this fact as it would not have been necessary, if CPLDs were not to add any complexity.

In summary, we think that the CPLDs are not particularly useful when developing power sequences where we would potentially need to reconfigure them repeatedly and may benefits can not really be seen. However, in a long-running production system where almost no reconfiguration are required, we think that CPLDs make the platform more solid and more understandable.

9.1.5 Do Declarative Power Sequencing Techniques generalize to Hardware?

In the introduction we stated that we would want to challenge the established abstractions and techniques in declarative power sequencing to new applications in hardware and thus check how robust these abstractions are. Also, we want to answer this question because it was a claim in [2].

We do think that these techniques are not only limited to the realm of software and instead generalize for hardware. First of all, consider the interface a controller needs to have by necessity. That is some digital input and digital output (we do not need any voltage sensing if we use regulators with power-good pins). This interface is common to all logical ICs. Second of all, remember that the CPLD boils down to a reconfigurable grid of logic gates. So instead of programming the CPLD with the partitioned sequence we might as well output some Verilog code that achieves the same purpose and etch a PCB based on that HDL-description.

9.2 Future Work

Based on our gained understanding of the problem at hand we have some suggestions for some future work.

Enable partitioning multiple sequences at the same time. In order to make partitioned sequences useful to real platforms we need to solve the problem of making sure that the partitioning of multiple sequences makes the choices which devices and pins control which part of the platform consistently for all the partitioned sequences.

Model and implement fault handling. The current platform model does not fully model how faults can occur and how they can be handled. Merely the alert levels are modelled at this time and in this work they are mostly used to configure the voltage triggers for the monitoring pins on the CPLD. Ideally, we would be able to generate code that is able to recover the platform gracefully from faults that have occurred. Since the ISPPAC has the capabilities for exceptions it would be extra nice, if we were able to use them. Although, we would have to solve the same problem as above.

Allow non-complete readers. In this work we required that all controllers executing read instructions are complete observers in that step. While this requirement simplified this work, it is not really necessary. Consider two read instructions in a step that are executed by two different controllers. While no controller is complete, if both sent the other respective other controller a communication that it has observed its read instruction then both would have observed all reads on the platform. To achieve this we would need much more sophisticated ways of computing which communications are necessary.

Push partitioning frontend. Many problems we need to solve when partitioning the sequence could also be tackled with constraint solving by pushing more work into the frontend. This way we would also be able to optimize the generated sequence to be as "distributable" as possible. Also, some of the more complicated proposed extensions to the partitioning algorithm might be more easily implemented by doing constraint solving.

Automate LogiBuilder. As a perk to reduce the turnaround time when changing the power sequence, it would be useful to automate the PAC-Designer GUI, so we do not have to click our program into there. An alternative approach to the automation of the GUI would be to reverse engineer the proprietary XML-based file format.

9.3 Summary

We have seen how we can partition a power sequence to execute on multiple controllers cooperatively. To achieve this we first showed what assumptions need to be weakened and which problems we might encounter as a result of that. In order to mitigate these problems we foresaw we presented our adjusted platform model which modelled all bus related components and actions explicitly as a key difference to prior work. Based on that model we presented a sequence model and devised an algorithm to partition such sequences between multiple controllers. We implemented these models and algorithms and demonstrated that we are able to partition sequences using three different strategies.

Bibliography

[1] J. Schult, “A model-based approach to platform-level power and clock management,” en, B.S. Thesis, ETH Zürich, Zürich, 2020. DOI: 10.3929/ethz-b-000490632.

[2] J. Schult, D. Schwyn, M. Giardino, D. Cock, R. Achermann, and T. Roscoe, “Declarative power sequencing,” *ACM Trans. Embed. Comput. Syst.*, vol. 20, no. 5s, Sep. 2021, ISSN: 1539-9087. DOI: 10.1145/3477039.

[3] M. Knüsel, “Optimizing declarative power sequencing,” en, M.S. Thesis, ETH Zürich, Zürich, Sep. 2021. DOI: 10.3929/ethz-b-000533011.

[4] L. U. Vogel, “Generating power management code from declarative descriptions,” en, B.S. Thesis, ETH Zürich, Zürich, Oct. 2021.

[5] “Enzian is a research computer built by the systems group at eth zürich.” (2022), [Online]. Available: https://enzian.systems (visited on 02/23/2022).

[6] D. Cock, *The enzian research computer. design and production data*, en, Zürich, 2021. DOI: 20.500.11850/517619.

[7] J. Frazelle, “Opening up the baseboard management controller: If the cpu is the brain of the board, the bmc is the brain stem.,” *Queue*, vol. 17, no. 5, pp. 5–12, Oct. 2019, ISSN: 1542-7730. DOI: 10.1145/3371595.3378404.

[8] “Openbmc.” version 2.11.0. (2015 – 2022), [Online]. Available: https://github.com/openbmc/openbmc (visited on 02/23/2022).

[9] “U-bmc.” (2018 – 2022), [Online]. Available: https://github.com/u-root/u-bmc (visited on 02/23/2022).

[10] E. Shobe and J. Mednick, “Runbmc: Ocp hardware spec solves data center bmc pain points,” Aug. 2019. [Online]. Available: https://dropbox.tech/infrastructure/runbmc-ocp-hardware-spec-solves-data-center-bmc-pain-points (visited on 02/23/2022).

[11] B. Zeidman, *Designing with FPGAs and CPLDs*, 1st ed. CRC Press, 2002. DOI: 10.1201/9780080494456.

[12] E. Robledo. “What is a cpld?” (Aug. 2021), [Online]. Available: https://www.autodesk.com/products/fusion-360/blog/cpld-overview/ (visited on 02/16/2022).

[13] *Isppac-powr1220at8 datasheet*, en, datasheet, version 2.0, Lattice Semiconductor Corp., Apr. 2019. [Online]. Available: https://www.latticesemi.com/view_document?document_id=17601 (visited on 12/08/2021).

[14] *Pac-designer software user manual*, en, version 6.32, Lattice Semiconductor Corp., Jun. 2014. [Online]. Available: https://www.latticesemi.com/view_document?document_id=51568 (visited on 12/08/2021).

[15] *I2c bus-specification and user manual*, NXP Semiconductors, Apr. 2014.

[16] *I2c manual*, en, Philips Semicondictors, Mar. 2003. [Online]. Available: https://www.nxp.com/docs/en/application-note/AN10216.pdf (visited on 02/15/2022).

[17] *System management bus (smbus) protocol specification*, version 3.1, System Management Interface Forum, Mar. 2019.

[18] "Ieee standard for design and verification of low-power, energy-aware electronic systems," Mar. 2019. DOI: 10.1109/IEEESTD.2019.8686430.

[19] A. Lungu, P. Bose, D. J. Sorin, S. German, and G. Janssen, "Multicore power management: Ensuring robustness via early-stage formal verification," in *Proceedings of the 7th IEEE/ACM International Conference on Formal Methods and Models for Codesign*, ser. MEMOCODE'09, Cambridge, Massachusetts: IEEE Press, 2009, pp. 78–87, ISBN: 9781424448067.

[20] "Beyond upf & cpf: Low-power design and verification," in *Design, Automation & Test in Europe Conference & Exhibition*, Los Alamitos, CA, USA: IEEE Computer Society, Mar. 2011, p. 1. DOI: 10.1109/DATE.2011.5763051.

[21] R. Sharafinejad, B. Alizadeh, and M. Fujita, "Upf-based formal verification of low power techniques in modern processors," in *2015 IEEE 33rd VLSI Test Symposium (VTS)*, Los Alamitos, CA, USA: IEEE Computer Society, Apr. 2015, pp. 1–6. DOI: 10.1109/VTS.2015.7116288.

[22] R. Mukherjee, P. Dasgupta, A. Pal, and S. Mukherjee, "Formal verification of hardware / software power management strategies," in *VLSI Design, International Conference on*, Los Alamitos, CA, USA: IEEE Computer Society, Jan. 2013, pp. 326–331. DOI: 10.1109/VLSID.2013.209. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/VLSID.2013.209.

[23] D. Macko, K. Jelemenská, and P. Cicák, "Power-management specification in systemc," in *Proceedings of the 2015 IEEE 18th International Symposium on Design and Diagnostics of Electronic Circuits & Systems*, ser. DDECS '15, USA: IEEE Computer Society, 2015, pp. 259–262, ISBN: 9781479967803. DOI: 10.1109/DDECS.2015.16.

[24] T. Bouhadiba, M. Moy, and F. Maraninchi, "System-level modeling of energy in tlm for early validation of power and thermal management," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '13, Grenoble, France: EDA Consortium, 2013, pp. 1609–1614, ISBN: 9781450321532.

[25] D. Kania, "Logic decomposition for cpld synthesis," *IFAC Proceedings Volumes*, vol. 33, no. 1, pp. 49–52, 2000, IFAC Workshop on Programmable Devices and Systems (PDS 2000), Ostrava, Czech Republic, 8-9 February 2000, ISSN: 1474-6670. DOI: https://doi.org/10.1016/S1474-6670(17)35585-4. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1474667017355854.

[26] D. Kania, "A new approach to logic synthesis of multi-output boolean functions on pal-based cplds," in *Proceedings of the 17th ACM Great Lakes Symposium on VLSI*, ser. GLSVLSI '07, Stresa-Lago Maggiore, Italy: Association for Computing Machinery, 2007, pp. 152–155, ISBN: 9781595936059. DOI: 10.1145/1228784.1228824.

[27] R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. L. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis* (The Kluwer International Series in Engineering and Computer Science), 1st ed. Springer, 1984, vol. 2, ISBN: 9780898381641. DOI: 10.1007/978-1-4613-2821-6.

[28] B. Kastrup, A. Bink, and J. Hoogerbrugge, "Concise: A compiler-driven cpld-based instruction set accelerator," in *Proceedings of the Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, ser. FCCM '99, USA: IEEE Computer Society, 1999, p. 92, ISBN: 0769503756. DOI: 10.1109/FPGA.1999.803671.

[29] S. Raveendran, P. Talwai, T. Khan, R. Balasubramanian, and K. Agilandaeswari, "Design of ioim for vme bus based cpu using cpld for nuclear power plants," in *Proceedings of the International Conference and Workshop on Emerging Trends in Technology*, ser. ICWET '10, Mumbai, Maharashtra, India: Association for Computing Machinery, 2010, pp. 991–993, ISBN: 9781605588124. DOI: 10.1145/1741906.1742137.

[30] D. C. Dyer and Y. L. Aung, "A multi-paradigm approach to teaching students embedded systems design using fpgas and cplds," in *Proceedings of the FPGA World Conference 2014*, ser. FPGAWorld '14, Stockholm and Copenhagen, Sweden: Association for Computing Machinery, 2014, ISBN: 9781450331302. DOI: 10.1145/2674095.2674099.

[31] D. Lymberopoulos, N. B. Priyantha, and F. Zhao, "Mplatform: A reconfigurable architecture and efficient data sharing mechanism for modular sensor nodes," in *Proceedings of the 6th International Conference on Information Processing in Sensor Networks*, ser. IPSN '07, Cambridge, Massachusetts, USA: Association for Computing Machinery, 2007, pp. 128–137, ISBN: 9781595936387. DOI: 10.1145/1236360.1236378.

[32] J. Vidal, F. de Lamotte, G. Gogniat, P. Soulard, and J.-P. Diguet, "A co-design approach for embedded system modeling and code generation with uml and marte," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '09, Nice, France: European Design and Automation Association, 2009, pp. 226–231, ISBN: 9783981080155.

[33] E. Aras, S. Delbruel, F. Yang, W. Joosen, and D. Hughes, "Chimera: A low-power reconfigurable platform for internet of things," *ACM Trans. Internet Things*, vol. 2, no. 2, Mar. 2021, ISSN: 2691-1914. DOI: 10.1145/3440995.

[34] R. Brzoza-Woch, A. Ruta, and K. ZielińSki, "Remotely reconfigurable hardware-software platform with web service interface for automated video surveillance," *J. Syst. Archit.*, vol. 59, no. 7, pp. 376–388, Aug. 2013, ISSN: 1383-7621. DOI: 10.1016/j.sysarc.2013.05.007.

[35] O. Reiche, M. Schmid, F. Hannig, R. Membarth, and J. Teich, "Code generation from a domain-specific language for c-based hls of hardware accelerators," in *Proceedings of the 2014 International Conference on Hardware/Software Codesign and System Synthesis*, ser. CODES '14, New Delhi, India: Association for Computing Machinery, 2014, ISBN: 9781450330510. DOI: 10.1145/2656075.2656081.

[36] Z. Guo, W. Najjar, and B. Buyukkurt, "Efficient hardware code generation for fpgas," *ACM Trans. Archit. Code Optim.*, vol. 5, no. 1, May 2008, ISSN: 1544-3566. DOI: 10.1145/1369396.1369402.

[37] "Enzian bmc power management tools." version commit: 62a16b6e. (2020 – 2022), [Online]. Available: https://gitlab.inf.ethz.ch/PROJECT-Enzian/enzian-bmc-powermgmt (visited on 01/28/2022).

[38] *Isl6334, isl6334a datasheet*, en, datasheet, version 3.00, Renesas Electronic Corp., May 2016. [Online]. Available: https://www.renesas.com/us/en/document/dst/isl6334-isl6334a-datasheet?r=497346 (visited on 02/13/2022).

[39] *Max15301: Intune automatically compensated digital pol controller with driver and pmbus telemetry*, en, datasheet, version 3, Maxim Integrated Products, Inc., Nov. 2013. [Online]. Available: https://datasheets.maximintegrated.com/en/ds/MAX15301.pdf (visited on 02/15/2022).

A Gamserrugg Platform Model

Here we have the platform model for our example platform “Gamserrugg”. A schematic representation can be found in figure 5.1 on page 32.

```yaml
# TODO: Update with correct model from code
devices:
- name: MAX15301
  kind: producer
  inputs:
  - pin: PWR
    kind: dc
    monitor: [voltage]
  - pin: EN
    kind: logical
    width: 1
  outputs:
  - pin: V_OUT
    kind: dc
    monitor: [voltage]
  - pin: PGOOD
    kind: logical
    width: 1
  busses:
  - pin: SDA
    protocols: [PMBus, SMBus, I2C]
  internals:
  - name: VID
    kind: dc
    default: default

- name: MAX8869
  kind: producer
  inputs:
  - pin: V_IN
    kind: dc
  - pin: SHDN
    kind: logical
    width: 1
  outputs:
  - pin: V_OUT
    kind: dc
```

```yaml

- name: MAX15053
  kind: producer
  inputs:
  - pin: V_IN
    kind: dc
  - pin: V_EN
    kind: logical
    width: 1
  outputs:
  - pin: V_OUT
    kind: dc

- name: SI5395
  kind: producer
  inputs:
  - pin: VDD
    kind: dc
  outputs:
  - pin: CLK
    kind: freq
    monitor: [frequency]
  busses:
  - pin: SDA
    protocols: [I2C]

- name: PSU
  kind: producer
  inputs:
  - pin: EN
    kind: logical
    width: 1
  outputs:
  - pin: V_OUT
    kind: dc

- name: CPU
  kind: consumer
  inputs:
  - pin: VDD
    kind: dc
  - pin: VTT
    kind: dc
  - pin: CLK
    kind: freq

- name: FPGA
  kind: consumer
  inputs:
```

```yaml
  - pin: VDD
    kind: dc
  - pin: VTT
    kind: dc
  - pin: VINT
    kind: dc
  - pin: CLK
    kind: freq

- name: ISPPAC
  kind: controller
  inputs:
  - pin: VCC
    kind: dc
  - pin: IN1
    kind: logical
    width: 1
    monitor: [logical]
    remote: true
  - pin: IN2
    kind: logical
    width: 1
    monitor: [logical]
    remote: true
  - pin: IN3
    kind: logical
    width: 1
    monitor: [logical]
    remote: true
  - pin: IN4
    kind: logical
    width: 1
    monitor: [logical]
    remote: true
  - pin: IN5
    kind: logical
    width: 1
    monitor: [logical]
    remote: true
  - pin: VMON1
    kind: dc
    monitor: [voltage]
    alert: compiled
  - pin: VMON2
    kind: dc
    monitor: [voltage]
    alert: compiled
  - pin: VMON3
    kind: dc
    monitor: [voltage]
    alert: compiled
  - pin: VMON4
    kind: dc
    monitor: [voltage]
    alert: compiled
  - pin: VMON5
    kind: dc
    monitor: [voltage]
    alert: compiled
  - pin: VMON6
    kind: dc
    monitor: [voltage]
    alert: compiled
  - pin: VMON7
    kind: dc
    monitor: [voltage]
    alert: compiled
  - pin: VMON8
    kind: dc
    monitor: [voltage]
    alert: compiled
  - pin: VMON9
    kind: dc
    monitor: [voltage]
    alert: compiled
  - pin: VMON10
    kind: dc
    monitor: [voltage]
    alert: compiled
  - pin: VMON11
    kind: dc
    monitor: [voltage]
    alert: compiled
  - pin: VMON12
    kind: dc
    monitor: [voltage]
    alert: compiled
  outputs:
  - pin: OUT6
    kind: logical
    width: 1
    monitor: [logical]
    remote: true
  - pin: OUT7
    kind: logical
    width: 1
    monitor: [logical]
    remote: true
  - pin: OUT8
    kind: logical
    width: 1
    monitor: [logical]
    remote: true
  - pin: OUT9
    kind: logical
    width: 1
    monitor: [logical]
    remote: true
  - pin: OUT10
    kind: logical
    width: 1
    monitor: [logical]
    remote: true
  - pin: OUT11
    kind: logical
    width: 1
    monitor: [logical]
    remote: true
  - pin: OUT12
    kind: logical
    width: 1
    monitor: [logical]
    remote: true
  - pin: OUT13
    kind: logical
    width: 1
    monitor: [logical]
    remote: true
  - pin: OUT14
    kind: logical
    width: 1
    monitor: [logical]
    remote: true
  - pin: OUT15
    kind: logical
    width: 1
    monitor: [logical]
    remote: true
  - pin: OUT16
    kind: logical
    width: 1
    monitor: [logical]
    remote: true
  - pin: OUT17
    kind: logical
    width: 1
    monitor: [logical]
    remote: true
  - pin: OUT18
    kind: logical
    width: 1
    monitor: [logical]
    remote: true
  - pin: OUT19
    kind: logical
    width: 1
    monitor: [logical]
    remote: true
  - pin: OUT20
    kind: logical
    width: 1
    monitor: [logical]
    remote: true
  busses:
  - pin: SDA
    protocols: [SMBus, I2C]

- name: BMC
  kind: controller
  outputs:
  - pin: EN_PSUP
    kind: logical
    width: 1
  busses:
   - pin: BUS
     protocols: [PMBus, SMBus, I2C]


platform:
  name: example
  instances:
  - name: psu
    device: PSU
  - name: bmc
    device: BMC
    busses:
    - pin: BUS
      address: 0x11
  - name: pac-cpu
    device: ISPPAC
    busses:
    - pin: SDA
      address: 0x60
  - name: pac-fpga
    device: ISPPAC
    busses:
```

```yaml
    - pin: SDA
      address: 0x61
  - name: cpu
    device: CPU
  - name: fpga
    device: FPGA
  - name: clk-main
    device: SI5395
    busses:
    - pin: SDA
      address: 0x6A
  - name: max-vdd-cpu
    device: MAX15053
  - name: max-vdd-fpga
    device: MAX15053
  - name: max-vint-fpga
    device: MAX15301
  - name: max-vtt-cpu
    device: MAX15301
  - name: max-vtt-fpga
    device: MAX8869

  wires:
  - name: en-psu
    from:
      instance: bmc
      pin: EN_PSU
    to:
    - instance: psu
      pin: EN
  - name: psu-pwr
    from:
      instance: psu
      pin: V_OUT
    to:
    - instance: pac-cpu
      pin: PWR
    - instance: pac-fpga
      pin: PWR
    - instance: max-vdd-cpu
      pin: PWR
    - instance: max-vtt-cpu
      pin: PWR
    - instance: max-vdd-fpga
      pin: PWR
    - instance: max-vint-fpga
      pin: PWR

  - name: en-vdd-cpu
```

```yaml
    from:
      instance: pac-cpu
      pin: OUT7
    to:
    - instance: max-vdd-cpu
      pin: EN
  - name: vdd-cpu
    from:
      instance: max-vdd-cpu
      pin: V_OUT
    to:
    - instance: cpu
      pin: VDD
    - instance: pac-cpu
      pin: MON2

  - name: en-vtt-cpu
    from:
      instance: pac-cpu
      pin: OUT6
    to:
    - instance: max-vtt-cpu
      pin: EN
  - name: vtt-cpu
    from:
      instance: max-vtt-cpu
      pin: V_OUT
    to:
    - instance: cpu
      pin: VTT
    - instance: clk-main
      pin: PWR
    - instance: pac-cpu
      pin: MON1

  - name: en-vdd-fpga
    from:
      instance: pac-fpga
      pin: OUT6
    to:
    - instance: max-vdd-fpga
      pin: EN
  - name: vdd-fpga
    from:
      instance: max-vdd-fpga
      pin: V_OUT
    to:
    - instance: fpga
      pin: VDD
    - instance: max-vtt-fpga
      pin: PWR
    - instance: pac-fpga
      pin: MON2

  - name: en-vtt-fpga
    from:
      instance: pac-fpga
      pin: OUT7
    to:
    - instance: max-vtt-fpga
      pin: EN
  - name: vtt-fpga
    from:
      instance: max-vtt-fpga
      pin: V_OUT
    to:
    - instance: fpga
      pin: VTT

  - name: en-vint-fpga
    from:
      instance: pac-fpga
      pin: OUT8
    to:
    - instance: max-vint-fpga
      pin: EN
  - name: vint-fpga
    from:
      instance: max-vint-fpga
      pin: V_OUT
    to:
    - instance: fpga
      pin: VINT
    - instance: pac-fpga
      pin: MON1

  - name: en-clk
    from:
      instance: bmc
      pin: EN_CLK
    to:
    - instance: clk-main
      pin: EN
  - name: clk
    from:
      instance: clk-main
      pin: CLK
    to:
```

```yaml
    - instance: cpu
      pin: CLK
    - instance: fpga
      pin: CLK

  - name: bus
    from:
      instance: bmc
      pin: BUS
    to:
    - instance: pac-cpu
      pin: BUS
    - instance: pac-fpga
      pin: BUS
    - instance: clk-main
      pin: BUS
    - instance: max-vtt-cpu
      pin: BUS
    - instance: max-vtt-fpga
      pin: BUS
```

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Declarative Power Sequencing using a CPLD

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Hässig	Manuel

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zürich, 23.02.2022	[illegible]

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.