Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Bachelor's Thesis Nr. 488b

Systems Group, Department of Computer Science, ETH Zurich

Generating Platform Configuration from Netlists

by

Georg Wehrli

Supervised by

Daniel Schwyn
Prof. Dr. Timothy Roscoe

November 2023 - May 2024

D INFK

Abstract

Most system software needs information about the platform they are running on. System configuration can take different forms. Creating system configuration can be tedious and error-prone. Netlists are a way to represent the connectivity of circuits.

In this work we explore how we can utilize the information stored in netlists. We create an algorithm which traces wires through a netlist to determine how different parts are connected. With this connectivity information we can validate and generate pin assignments of designs for field programmable gate arrays. An algorithm for identifying devices on an I2C bus shows that also more complex structures can be extracted form a netlist. To implement these algorithms a rust program was created.

Acknowledgements

Throughout the writing of this thesis, I have received a lot of support and assistance. First, I want to express my gratitude towards my co-supervisor, Daniel Schwyn, who has invested countless hours helping me understand the different problems encountered during this work. His answers and feedback always helped me keep moving forward when I felt stuck. I would also like to thank my sister for the many learning sessions we spent together, motivating each other. Lastly, an enormous thank you goes to my girlfriend, who encouraged me along the way to keep working and for all the breaks we had that fully recharged me.

Contents

1 Introduction **1**

2 Background **2**
2.1 Enzian . . . 2
2.1.1 Baseboard Management Controller . . . 2
2.2 Netlist . . . 2
2.2.1 The Netlist of Enzian . . . 4
2.3 Inter-Integrated Circuit (I2C) . . . 5
2.4 Xilinx Design Constraints . . . 5
2.5 Electronic Components . . . 5
2.5.1 Capacitors . . . 6
2.5.2 Resistors . . . 6
2.5.3 Level Shifter . . . 6
2.6 Previous Work . . . 6

3 Implementation **7**
3.1 Problem . . . 7
3.2 Using Netlists . . . 7
3.3 Connection Between Components . . . 8
3.4 FPGA Configuration . . . 9
3.5 I2C Bus . . . 11
3.6 Connecting Netlists . . . 13
3.7 Augmenting the Netlist . . . 14
3.8 Structuring The Data . . . 16

4 Evaluation **19**
4.1 FPGA Configuration . . . 19
4.1.1 Validate Pin Assignments . . . 19
4.1.2 Generating PACKAGE_PIN Constraints . . . 24
4.2 I2C . . . 26
4.3 Characterizing . . . 26
4.3.1 Generating Netlists . . . 27
4.3.2 Converting Netlists . . . 28
4.3.3 Connection Search . . . 29

5 Conclusion **30**

Listings

3.1 Example specification file 10
3.2 Example of a file detailing the internal connections of components 14
3.3 Netlist struct 17
4.1 Constraints A 20
4.2 Constraints B 20
4.3 Extract pin pairs from netlist 20
4.4 Section of the specification for the DIMM connector 24
4.5 Generating a XDC file. 25
4.6 Differences between generated and and existing constraint files 25
4.7 Component specification detailing how signals going through the level shifter. 26
4.8 Extracting I2C bus devices. 26

Chapter 1

Introduction

Systems software needs information about the hardware it is running on. This information has different forms, for example the device tree in the linux kernel or Xilinx Design Constraint (XDC) files to create correct bitstreams for field programmable gate arrays (FPGAs).
Computer systems can be very complex. To facilitate the design process, computer-aided design tools (CAD tools) are used. A fundamental aspect of such a design is how the different electronic components are connected to each other. This crucial information, detailing the interconnections between components, is often stored in a netlist.
In this work, we will explore how we can extract information out of the netlist and how it can be used to aid in the process of creating platform configuration files.

Motivation

While it is possible to create system configuration files by hand, it is usually a tedious and error prone process, because the necessary information are often in different files and different formats. Since netlist contain the hardware topology, they can be used to propagate information used in system configuration files.

Chapter 2

Background

When working with netlists we need to understand what they are. To use the information from the netlist to aid in the creation of platform configuration it is crucial to know how they are structured and used.

In this chapter we will briefly shine a light on the Enzian platform. Then we will learn what netlists are and take a look at the netlist of the Enzian platform, followed by a small overview over the I2C bus and xilinx design constraints. And at the end the end there is a very broad introduction to a few electronic components and their function.

2.1 Enzian

The Enzian is a computer for hybrid systems research, developed by the Systems Group at ETH Zurich. It is a two-socket board and is equipped with a ThunderX-1 CN8890P-NP CPU and a Xilinx XCVU9P Ultrascale+ FPGA. The CPU and FPGA are connected via the CPU's native inter-socket cache coherence protocol. [1]

2.1.1 Baseboard Management Controller

A baseboard management controller (BMC) is a chip on a computer orchestrating the components on the board. It manages clocks and power distribution. On the Enzian a Enclustra Mercury ZX5 with a Xilinx Zynq 7000, running a OpenBCM, is used as the BMC. [1]

2.2 Netlist

Netlist describe how different components are connected to each other. This includes a variety of different components such as logic gates or parts on a circuit board. There is not a specification or a single format used to save netlist, but different formats for different purposes and CAD tools [13]. Whilst the exact format might differ, the content of a netlist usually a list of components, and a list of how those components are connected to each other (the nets).
In the specific case of the Enzian platform the netlist was exported from the Altium CAD tool and then parsed into a json file. This file has two lists: one for components and one for wires. A component is a json object with three fields:

- component_type: This field has information about what component type is used for this components.
- description: A field to add additional information about components.
- designator: This field contains an identification code for a component.

A wire is a json object with three fields:

- name: This field contains the name of the wire. In many cases it hints on the function of the wire, for example a name "I2C.SDA" is probably the serial data signal of an I2C bus.
- number: Unique number for each wire.
- pins: A list of pins which are connected to the wire.

A pin is a json object with five fields:

- part_value: This field is the component type of the component the pin is on.
- pin_name: This is the name of the pin. Often it is the same as the pin number, other times it hints at the function of the pin.
- pin_number: The pin number identifies the pin on the component. It can be a number or a mix of letters and numbers.
- pin_type: This field describes which type a pin has.
- reference: This field is the designator of the component on which the pin is on.

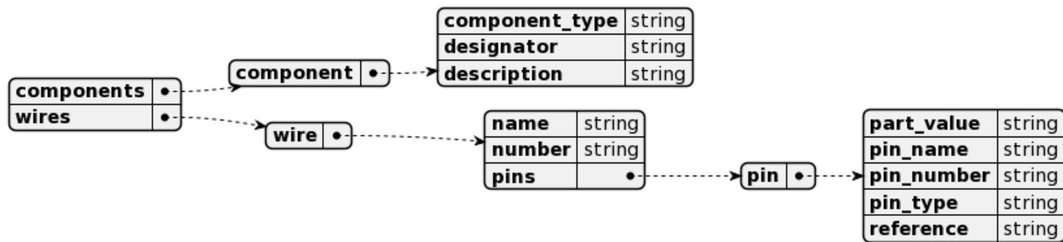


Figure 2.1: Illustration of the schema of the netlist from Enzian.

This structure fully captures how all the different components on the Enzian board are connected with each other. It creates a superset of the actual connections on the board, since there are some parts which are in the netlist but not used in the final design. In this thesis the term netlist will refer to a layout like this (if not stated otherwise).

In this thesis we assume that the component types, pin numbers and designators are correct. We do not use the pin names and wire names because even if they contain information about their function, there is no standardized way how these names come about.

2.2.1 The Netlist of Enzian

There are a few things to note about the netlist of Enzian.

The `component_type` field of component is in most components a specific identification number for an electronic component, but it also can be a more general description as for example "Capacitor", "Resistor" or "Test Point SMD". Having such general component types limits what can be directly done with the netlist, because for example if one wants to use it to estimate some electrical properties the component type does not reliably represent what exact component it is.

The `description` field can hint at a components function. At times it is empty; alternaternatively it contains some more descriptive text as for example "UBS Type A THT" or just the component type.

The designators of the components are unique except for the following: L100, L102, L104, L106, L108, L131, L133, L135, L138, L140, L142. For each of this designator there are two component types, namely "FBMH1608HM101-T", which is a ferrite bead, and "0603_TRUE_0R". Presumably this conflict of designator are options in the design, so either option could be used in the produced board.
When designators are not unique, the designator of a component type (the `reference` and `part_value` fields of pins) can be used to form a key for a specific designator. Some pins reference the component type with an OR in front of the actual component type, thus the pair can not be used as is. But when removing the preceding OR it works fine.

The designators start with one of the following prefixes "B", "C", "D", "DR", "ESD", "FD", "HS", "IC", "J", "L", "LM", "NT", "Q", "R", "S", "SN", "SW", "TP", "U" and are followed by a number. The prefixes group different component types (Table 2.1).

Table 2.1: Designator Prefixes

Prefix	Component Type
B	battery holder
C	capacitors
D	diodes
DR	drilled holes
ESD	ESD suppressor
F	fiducial
HS	heat sink
IC	integrated circuits (incl. CPU & FPGA)
J	connectors
L	inductors / ferrite beads /filters
LM	layer markers
NT	net tie
Q	transistors
R	resistors
S	tactile switches
TP	test points
U	integrated circuits

In the netlist of the Enzian the following pin types were observed: ***INPUT***, ***OUTPUT***, ***POWER***, ***PASSIVE***, ***OPEN COLLECT***, ***I/O***.

2.3 Inter-Integrated Circuit (I2C)

To understand how we can identify devices which are part of an I2C bus we first have to understand how the I2C bus is built.

The I2C-bus protocol is a method of communication for different integrated circuits in an circuit. The circuit connected on the bus are recognized by an unique address. There are controllers and targets, several controllers can be on the same bus due to an arbitration process. Controllers initiate data transfers on the bus and generate the clock signals for the transfer. The I2C bus consists of two wires; the serial data (SDA) and serial clock (SCL) wire. When no data is being transferred the wires are pulled up by a pull-up resistor. [4]

2.4 Xilinx Design Constraints

Understanding FPGA configurations is crucial to know what information we need from the netlist to validate and generate such configurations. FPGAs are highly configurable integrated circuits. The big FPGA on the Enzian and the FPGA of the BMC are both AMD FPGAs, hence this section is about creating designs in Vivado. Vivado is the design software for AMD adaptive SoCs and FPGAs.[12] Vivado can generate bitstreams which are uploaded to the FPGAs to program them. To create a correct bitstream Vivado needs some more information, so-called design constraints. The design constraint are commands that follow the Tcl semantic. Examples for such constraints are:

- Clock constraints for defining clocks, which are the time reference for reliably transferring data between registers.
- I/O delay constraints to model external the external timing context.
- Physical constraints to inform how the system should be layed out in the FPGA.

Physical constraints are commonly saved in a Xilinx Design Constraints (XDC) file. This is then read by Vivado when opening the design. Typically a physical constraint has the following layout: `set_property <property> <value> <object list>`.[11]

So an line in a XDC could look like this:

```
set_property PACKAGE_PIN AA19 [get_ports B_USERIO_1V8.LED1]
```

The property `PACKAGE_PIN` assigns a logical port, so to speak a signal of the FPGA code, to a specific pin of the FPGA. So in the example the pin `AA19` gets assigned to the port of the signal `B_USERIO_1V8.LED1`.

2.5 Electronic Components

In an electronic circuits there are often various different component types. In this section a select handful of components are briefly highlighted to get a basic understanding of their functionality.

2.5.1 Capacitors

A capacitor can store electrical energy. In a circuit it can have different functions. There are different functions a capacitor can fulfill in a circuit. Aside from storing charge they can be used for filtering, coupling and decoupling. [7]When used as a filter they can suppress unwanted frequencies. Coupling is when it is intended that a signal can be transferred. When decoupled you do not want a signal to be able to be transferred over a connection, for example in a constant power supply.

2.5.2 Resistors

Resistors have a wide variety of application. They can be used for limiting current, voltage division and more. A so-called pull-up resistor is a resistor used to ensure a well-defined logic level when a signal is not actively being driven by an input. For example the wires of an I2C bus can be driven by a pull-up resistor to ensure that they are HIGH when not pulled down by a device on the bus.

2.5.3 Level Shifter

A level shifter is used to interface between two different logic levels. For example on the Enzian platform there are I2C buses connected to the XCVU9P FPGA and DIMM, the side of the bus at the FPGA operates on 1.2 V while the DIMMs operate on 2.5; connected by a level shifter.[3]

2.6 Previous Work

Kruszewski et al. describe a way to generate design constraints for multi-board systems. They use pin mappings for the different board they connect to build a tree from which they can infer the constraints. It is not made clear how the mappings are made. [5]

Chapter 3

Implementation

3.1 Problem

Every computer program runs in an environment. This environment consists of everything the program needs to run. Operating systems create an abstraction of the hardware such that programs in user space can be agnostic about the hardware they run on. In other words, the operating system provides an environment in which the program is run. System level software does not have this abstraction layer and therefore needs to be aware of the hardware it is running on, the environment it is in. A device tree, describing the different components of a systems, or a Xilinx Design Constraint file, detailing how a bitstream for an FPGA should be created, are different examples for system configuration files. Often these system configuration files have to be created by hand. To tackle this problem the idea of using netlists to aid in the creation of configuration files came up.

3.2 Using Netlists

To make use of the netlist we create a program which can parse the information from a netlist and manipulate it to get useful information out of it. The programs requirements are determined by what we aim to extract from the netlist and how we want to do it. As a first step we will discuss what we could potentially use a netlist for.

At a very basic level we are interested in the connectivity of the different elements of the netlist. Knowing how different signals are wired between components allows us to direct information from one part of the netlist to an other part. If we have a specification on one component of the netlist, this then can be used to validate that this specification holds on a connected part on the netlist. A simple case that comes to mind is validating pin assignments or creating constraints for an FPGA. A more involved application is to not only propagate data from part a to part b of the netlist but also manipulate the data on the way.

As the netlist stores a whole circuit it also can be used to obtain parts of a circuit. For example if the question is which integrated circuits are connected to an I2C bus, the netlist holds this information. Extracting the power tree of a circuit which then can be used to automatically generate power sequencing [8] is a more comprehensive example.

In the following sections we will take a deeper look at different applications where we can use a netlist and how we can use it. In the last part the the structure used to implement the different functionality on is discussed.

3.3 Connection Between Components

As the netlist stores the connectivity of a circuit we can use it to propagate information along wires through the netlist. This aids us to figure out how different parts influence each other. Put differently, we want to extract how different components are connected. These connections might be one wire connecting two components directly or indirectly over multiple wires and components. More precisely we want to look at the pins of components, because often the function of a signal is defined on by a pin.

To simplify this problem we consider two components; A and B. The goal is now to extract from the netlist how the pins of A and B are connected. To achieve this we can take a pin on component A, take the wire this pin is connected to and then check if there are pins on the wire connected to component B. This gives us a list of pins on B which are connected to the pin on A. However this only reports direct connections which would limit the capabilities of the program immensely as there are many cases where there are components between two parts we are interested in. An example would be, when there is a capacitor, resistor or a level shifter connecting two wires. To propagate data over a capacitor or resistor it is relatively straight forward as they usually are connected to two wires. When coming from one end to such a component is clear where to continue. For a component with more than one two connections, as a level shifter, it is not clear from the netlist how a signal propagate through it.

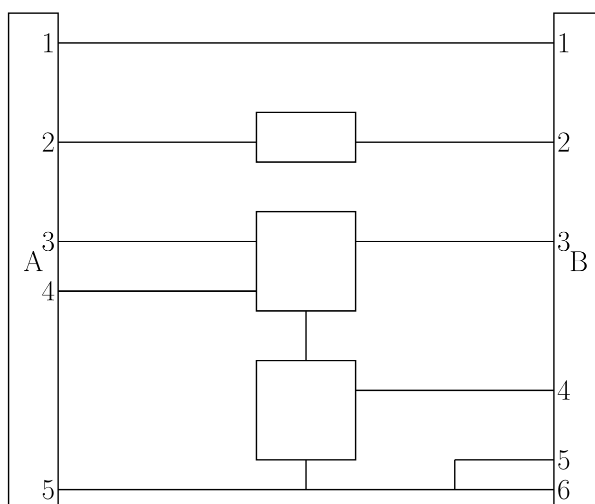


Figure 3.1: Diagram illustrating how different connections between parts can look. The boxes are arbitrary components and the lines represent wires connecting them. There is a direct connection, a connection with a simple component between the two parts and a more complex situation where it is not clear how signals go through the components.

Tracing a connection through components is not straight forward. Components can have complex functionality, which is not reflected in the netlist. Additionally the way we trace connections through components might depend on what we want to achieve. The only data in the netlist that could hint to how pins of a component relate to each other are the pin names. For example the component with the designator ESD1 (an electrostatic discharge suppressor) has two pins with the name IO1 and two pins with the name IO2 which are connected internally respective, the component with the designator U7 (a bus switch) has 10 pin pairs with the names A1-10 and B1-10 or the component U5 (a multiplexer) which has 4 pin triplets with the names 1-4A, 1-4B1 and 1-4B2. These

might look like a good way to find out how signals are wired through components, but as mentioned in section 2.2 the names are not a reliable source since they do not follow a standard.

Considering this we get the following algorithm (Algorithm 1) to find the connections from a pin of component A to pins of component B. The algorithm has a wire queue. For each wire in the queue it checks the all the components connected to the wire if they are component B. If the component is component B the pin on which the wire is connected to component B is added to the result list, else all connected wires of the component are added to the wire queue.

Algorithm 1 Search Connections (1)

```
1:  Input: pin_a, component_B
2:  wire_queue ← pin_a.wire
3:  for wire in wire_queue
4:      for pin in wire.pins
5:          if pin.component == component_B then
6:              result ← pin
7:          else
8:              wire_queue ← pin.component.wires
9:          endif
10:     endfor
11: endfor
12: return result
```

The algorithm captures all connections, but in many cases also connections we do not want. To understand better when it works fine we can remove the two components A and B from the netlist. This splits the graph into several connected components (Here we do not mean components of the netlist but parts of a graph). These connected components are the connections we find in the netlist. If the components A and B would be removed from Figure 3.1 there would be three connected components. One connecting pin 1 and 1, one connecting pin 2 and 2 and on connecting pins 3,4,5 and 3,4,5,6 of component A and B respectively. In other words we are only able to extract a one-to-one pin pair for connectet Nevertheless if we use the algorithm on every pin of component a we can create pairs of lists of component which we then can use in other applications as checking and creating FPGA configurations.

3.4 FPGA Configuration

The highly customizable nature of FPGAs leads to a lot of configurations which have to be created. Some of the configurations directly depend on what function is implemented and how the FPGA is connected to other peripherals on the circuit board. Hence it would be very helpful to verify and generate different configurations.

Xilinx design constraint (xdc) files have a wide spectrum of properties they can specify, some of which can be inferred with the help of netlists. An interesting property for us is the `PACKAGE_PIN` property. This property informs Vivado (the tool used to program Xilinx FPGAs) which logical signal should be on which pin of the FPGA. To which pin a signal connects is mostly determined by its function. The logic for an I2C module in

the FPGA should be connected to the appropriate pins which are connected to a I2C bus on the board. The function of the wire on a board is determined by the component it is connected to or respectively what it represents, for example a PCIe bus or a wire powering an LED. Not all pins of the FPGA can be assigned by the user. There are some pins with a predefined function and power pins. For the large XCVU9P FPGA the pin AE12 is an example of a pin with a predefined function as it is used for the PROGRAM_B_0 signal. In the packaging and pinouts product specification for the FPGA [10] there is an overview over every pin of the FPGA.

To validate if Vivado has generated the desired pin assignments a few things are required. Firstly the assignments from Vivado are needed, fortunately they can be exported easily as a csv file after the synthesis of a project. Secondly something to compare the exported pin against is needed. This can be any specification, for example the pinout specification of a PCIe connector. Then the key here is that the specification is on a different part of the circuit and is transferred over the netlist to the FPGA. This gives a mapping from the FPGA pins to the expected signals. They then can be compared to the actual signals the pins were assigned after the synthesis and errors can be detected. A small side note is that the logical signals in the FPGA can have arbitrary names. In practice they are usually named after the function they have. So the name "i2c_0_scl" will be a part of an I2C bus. So if we want to compare the logical signal names to a specification it is crucial that the two are somehow comparable. In other words if we want to use a specification from a component (e.g. a pinout sheet of a connector) we have to modify it such that it is clear how it relates to the logical signal.

Optimally the names in the specification are the signal names used in the FPGA. If they do not match we need a function to convert the signal names. As this function also has be provided it can be considered as a part of the specification. If we have several connections of a certain type we might want to use such a function to add a prefix to the signal names such that we do not have duplicate names.

To generate the PACKAGE_PIN property for a XDC file we can use a similar approach, where a specification the required constraints on a component of the netlist and then gets propagated through the netlist to the FPGA pins.

The format used for the specification is as follows. It is a JSON file with a field to specify what part the specification is for and a list of pairs detailing which pin has which signal.

Listing 3.1: Example specification file

```
{
"part": "part",
"pin_signal_pairs": [
    {
        "pin_nr": "1",
        "signal": "a"
    },
    {
        "pin_nr": "2",
        "signal": "b"
    }
]
}
```

Given such a file and the designator of corresponding part and the FPGA we can generate a constraint file by first using the search connections algorithm (algorithm 1) to find the pin connections of every pin. Then for the pins where there is only one corresponding pin found on the FPGA we can save `set_property PACKAGE_PIN <FPGA pin> [get_ports {<signal from specification>}]` to an output file. The pins for which it was not clear how they are connected are reported to the user such that they can be manually added.

3.5 I2C Bus

A somewhat more intricate goal is to extract I2C bus topologies. For this process we can use more than just how one signal is connected to different components but how two signals are connected.

In some cases extracting it might be as simple as gathering all components on a SCL or SDA wire to extract the bus topology, but this is not necessarily the case. When there is more then one controller on the bus there needs to be a pull-up resistor. Since the I2C signals need to be high unless pulled down by a component, this voltage has to be supplied. If there is one controller on the bus the controller can pull the lines to high. Sometimes even if there is only one controller and certainly when there is multiple controllers the I2C wires are pulled to HIGH by a pull-up resistor. [4] So if we would take all components connected to the SDA or SCL wire we would also have the pull-up resistor in that result. Fortunately design of the I2C bus gives us some constraints on which components of the netlist we can view as part of the I2C bus and which we can discard. Mainly we know that a component on the I2C bus has to connect to both the SDA and SCL wire. A simple algorithm is the following, where we have as an input two wires making up the I2C bus. The algorithm makes two sets, a set for the components connected to the SDA line and a set for components connected to the SCL line. Then it returns the components which are in both sets.

Algorithm 2 Extract I2C topology (1)

1: **Input:** SCL_wire, SDA_wire
2: $SCL_components \leftarrow SCL_wire.components$
3: $SDA_components \leftarrow SDA_wire.components$
4: $I2C_components \leftarrow SCL_components \cap SDA_components$
5: **return** $I2C_components$

While with this approach we capture all the components if they are directly attached, cases exist where this might not be the case. An example is a situation where the components connected to the bus work on different voltages and thus require a level shifter between the I2C bus and the component. To discover the component connected to the I2C bus we would need to "step" over the level shifter. This could possibly happen in two different ways: Either there is a level shifter for both lanes or both lanes connect to the same level shifter. With the approach so far we would miss the part of the I2C bus with a different voltage level in both cases. In the first case we would assume that the level shifters are not an important part of the bus at all and in the latter case we would mistake the level shifter as part of the I2C bus.

With only the information contained the netlist it is not clear what components do.

As a result we cannot know if the component we assumed to be on the I2C bus really is a part of it or if it is a component the bus passes through like a level shifter.

If the two bus lines are connected over two separate components, it is possible to infer which wires are the bus lanes. The following is a proposal how to achieve this: We start the same way as before, check for all the components on each of the lines if they are connected to both lines. The wires of components which are not connected to both bus lines are collected in two different sets. One for the wires indirectly connected to the SDA lane and on for the wires indirectly connected to the SCL lane. For example if we have a resistor which is connected to the SCL line and a voltage supply wire we add the voltage supply wire to the set of wires for the SCL wires. Then we remove the wires which are present in both sets from said sets. This is done on two assumptions. The first assumption is that two parts of an I2C bus lane only are connected at one singular point. In other words if we have to two SDA wires which are connected to each other to form one SDA wire of an I2C bus, they are connected via a single connecting component. This is a reasonable assumption because having two connections would not only increase complexity but most likely also have a negative impact on signal integrity. The second assumption is that only I2C bus lanes are not connected to both wires indirectly. For example the two pull-up resistor connected to the two bus lines are connected to a the same voltage supply wire. If that was not the case we would assume the wires connected to those resistors are part of the I2C bus. Wires indirectly connected to an I2C bus are most likely to power the circuit where again it makes sense that the same power supplies are used for signal integrity. This potentially leaves us with two non-empty sets of wires. If the sets contain one wire each it is simple and the two wires are used as the new input for the algorithm. It is more challenging when the sets contain more than one wire because then we have to match them such that we get a new pair of wires to continue. To achieve this we have to use some heuristics. A simple approach is to just compare how many components are on the wires but clearly this only works if two possible buses do not have the same amount of components. An other possibility is to figure out which wire were connected with the same component type. This assumes that different bus parts are connected with component types. Maybe also the wire names could be compared to figure out which pair up or the user has to decide how to match up the wires. Then with the new pair of wires the algorithm do the procedure again of checking which components of the new wires are part of the bus and which wires are indirectly connected, leaving out wires already visited. This is repeated until there are no more matching wires that could be part of the I2C bus, then the found components on the bus are returned. The resulting algorithm looks like this:

Algorithm 3 Extract I2C topology (2)

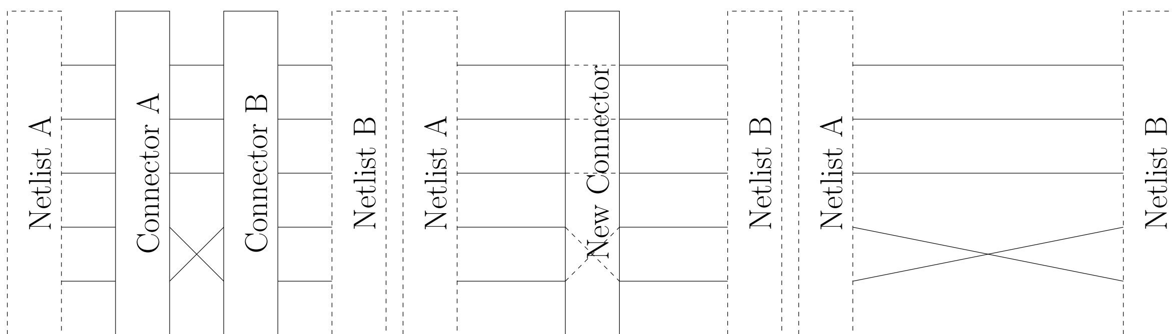
1: **Input:** SCL_wire, SDA_wire
2: $(SCL_queue, SDA_queue) \leftarrow (SCL_wire, SDA_wire)$
3: **while** $(SDA_wire, SCL_wire) \leftarrow$ **match_wires**(SCL_queue, SDA_queue) **do**
4: $\quad SCL_components \leftarrow SCL_wire.components$
5: $\quad SDA_components \leftarrow SDA_wire.components$
6: $\quad I2C_components \leftarrow SCL_components \cap SDA_components$
7: $\quad SCL_queue \leftarrow (SCL_components \setminus I2C_components).wires$
8: $\quad SDA_queue \leftarrow (SDA_components \setminus I2C_components).wires$
9: $\quad$ **remove_intersection**(SCL_queue, SDA_queue)
10: **endwhile**
11: **return** $I2C_components$

While this algorithm can cover a larger set of possible I2C bus topology layouts its output quality might suffer because of the different assumptions it takes. A case not covered by this algorithm is when a single level shifter is used to shift the voltage levels of the wires. In section 3.7

3.6 Connecting Netlists

When working with netlists we might not only use a single netlist. The Enzian platform uses a daughter board for its network connectors, so if we want to propagate some information from a network connector it would be useful to be able to connect different netlists. A connection between two netlists can be established through two connectors. There are several different way two netlists could be connected.

Figure 3.2: Different ways to connect two netlists



(a) New wires on the existing connectors connecting two netlists.

(b) A new component replacing the connectors connecting two netlist.

(c) The connectors removed and wires merged together to connect two netlists.

Changing the two components that should be connected is an option (a). Each pin on the components are doubled and new wires are created to connect the new pins. A slightly different approach is to merge the two connectors together (b). Both options are limited by the fact that it is not possible to infer how wires connect over components with the netlist alone. When doubling pins or merging two components without changing the pin numbers we end up with components which have pairs of the same pin number.

This opens up the option to step over the connecting components by matching up the pin numbers, but this fundamentally goes against the fact that pin numbers are unique. And even if that was no issue it only solves the problem for the components which are used to connect the netlists. For all other components it still is unclear how to pass data through them.

To circumvent the problem of not knowing how signals go through a connector both connectors can be omitted and the wires can be merged (c). In a way this is a very natural way of connecting two netlists, as in the real world if you connect to wires they will act as if it was one. A downside of this approach is that a lot of wires have to be removed and changed, but with some carefully written code this is no problem. As mentioned not have a problem not now to pass information over the formed connection but again it only solves the problem for the components which were connected, but it solves it in such a way that we need no further information than what is in the netlist.

While in most cases it might be enough to have a one-to-one matching of the pins, so pin 1 of connector one connects with pin 2 of connector two, sometimes there are connections where this does not hold. In this case how the pins should connect has to be provided by the user. All three different options can deal with this.

In the end we implemented the first option as this variant only modifies the two connectors and adds wires but does not remove components.

A whole other issue which arises when connecting two netlists: the identifiers might not be unique anymore. To address this we can change the identifiers to include a short tag for the respective netlist at the beginning. So if we connected the Enzian netlist with the BMC the designator `IC1` on the Enzian could be changed to `E.IC1` and the designator `IC1` on the BMC could be changed to `B.IC1`. This way it is possible to unambiguously address components on connected netlists.

3.7 Augmenting the Netlist

Both applications, I2C and xdc verification, are limited by the fact that the netlist only contains information about the connectivity between the components on the board. While as we have seen we can follow signals through the netlist and at least partially extract the I2C bus topology without the exact knowledge about how components behave, we could achieve more precise results if we could add additional information to the netlist. information it immensely limits the capabilities. This raises the question how we can augment the information in the netlist.

As the information one might want to add to the netlist is very versatile, and the actual data might even depend on the use case. For this reason it does not seem practical to save this data within the netlist, separate files are more reasonable.

Both while verifying constraint files and extracting I2C bus topologies we ran into the problem that the netlist is missing information about how signals are connected through components. Here a file with this information would make the results more reliable no assumptions about the connectivity have to be made. Such a file could have the following form: It is a list of components which themselves have lists of connected pins.

Listing 3.2: Example of a file detailing the internal connections of components

```
[
    {
        "component_type": "example_component_a",
```

```
            "connected_pins": [
                ["1","3"],
                ["2","4","5"]
            ]
        },
        {
            "component_type": "example_component_b",
            "connected_pins": [
                ["1","2"],
                ["3","4"]
            ]
        }
    ]
```

This is in essence a netlist again, but to save it in the same format as the netlist of the board would be overkill. If we only want to store the information how a component connects pins internally we do not need the notion of different components, pins and wires. Something to note here is that most components do not propagate logical signals through them but some components as for example level shifters or multiplexers certainly do. This is a very simple format and only hold information about how signals pass through components, which is enough to know how different pins are connected. If we for example wanted to model more complex behavior it might be possible to create files which store the necessary information to achieve this goal.

To create such a file holding this information about how the components propagate signals, the schematics or data sheets of the components have to be inspected.

With this information we can greatly simplify the tracing of the wires used for the verification of constraint files and the extraction of the I2C bus.

Connection Between Components

With this additional information about how signals are carried through a component it is much easier to find out how pins are connected. Although it might seem that we would need no information if we should pass over the component, but that is not the case. If we have a signal of an I2C bus there needs to be a pull-up resistor pulling the wire to high, but we do not want to follow the signal over this resistor as it connected to the power supply in which we are not interested if we want to follow the I2C bus. With this we can create algorithm 4.

Here the connected_pin functions returns the pins of a component which are internally connected to a given pin of a component, if there is no connection nothing is returned. The information of how the signals are connected within components has to be created manually. This makes it possible to make errors and is time consuming.

I2C Bus

To improve the algorithm for finding we can use a file specifying the internal connections of a components. This file can tell us if a component is not a device of the I2C bus. When a signal of the I2C bus is transferred through a component we know that it is not a device on the I2C bus. With this we can create the following algorithm.

Algorithm 4 Search Connections (2)

```
1: Input: pin_a, component_B, component_information
2: wire_queue ← pin_a.wire
3: for wire in wire_queue
4:    for pin in wire.pins
5:      if pin.component == component_B then
6:        result ← pin
7:      else
8:        wire_queue ← component_information.connected_pins(pin).wires
9:      endif
10:   endfor
11: endfor
12: return result
```

Algorithm 5 Extract I2C topology (3)

```
1: Input: SCL_wire, SDA_wire, component_information
2: (SCL_queue, SDA_queue) ← (SCL_wire, SDA_wire)
3: while (SDA_wire, SCL_wire) ← match_wires(SCL_queue, SDA_queue) do
4:    (SCL_components, SCL_queue) ←
          component_information.get_connected(SCL_wire)
5:    (SDA_components, SDA_queue) ←
          component_information.get_connected(SDA_wire)
6:    I2C_components ← SCL_components ∩ SDA_components
7: endwhile
8: return I2C_components
```

The get_connected method of the component information returns the components and wires connected to a given wire. The components it returns are components where the signal ends, if a wire is connected to pin 1 and this pin is not connected internally the component the pin is on is returned. The wires the method returns are wires which are connected to the starting wire, i.e. if the given wire connects to pin 1 and the pin is connected internally to an other pin of the component the wire to the other pin is returned.

If the list of components which transfer the signal internally is correct this algorithm allows us to get at least all devices on I2C bus. This is because with this algorithm we essentially know the whole extend of the SDA and SCL wires and we do not have to guess how the wires are connected.

3.8 Structuring The Data

The structure of the netlist (as described in 2.2) is tedious to work with. In this section we will show how we restructured the netlist to use in the rust tool we created. While with the existing layout of the netlist it is easy to know how wires connect to pins and pins connect to components, it is hard to know the pins on a specific component and given a pin which wire it connects to. The relation between pins and components is given by the `component_type` and `reference` field of the pins, which create a unique key for a

component in the component list. To know which are on a component we have to go over every pin of every wire an check if the pin is on the desired component. Additionally the pins are stored within the data of the wire. This makes clear how the wire connects to pins but given a pin we once again have to go over all the wires to find the wire on which the pin is located. To make the connections in all directions clearer we add references from components to pins and pins to wires. Additionally we change the list of pins in the wires to a list of references to pins to make the whole structure more consistent.
This results in a structure similar to an adjacency list with three different node types.

- **Components** have a designator as an identifier, a component type which gives some information about what kind of component it is and a list of references to pins on that components.
- **Wires** have a unique wire name and a list of references to pins connected to that wire.
- **Pins** Pins have a number, a name, a pin type and a reference to the component and wire it connects.

To keep track of all the parts there is a list for every type. This also makes it simple to search for a wire by its name or a component by its designator. For simplicity the references are the index into the respective list. For example the component reference of a pin is the index of the component in the component list. To capture this data layout the netlist struct was created.

Listing 3.3: Netlist struct

```
struct Netlist {
    wires: Vec<Wire>,
    pins: Vec<Pin>,
    components: Vec<Component>,
}

struct Wire {
    name: String,
    pins: Vec<usize>,
}

struct Pin {
    name: String,
    number: String,
    pin_type: String,
    component: usize,
    wire: usize,
}

struct Component {
    component_type: String,
    description: String,
    designator: String,
    pins: Vec<usize>,
```

```
}
```

This layout creates the foundation of our tool. It captures the connectivity of all the parts.

A downside of this structure is that the different elements (the wires, pins and components) of the netlist have no direct reference to other elements of the netlist. As a result all accesses to different elements have to have to be accessed via the netlist struct.

Chapter 4

Evaluation

In this chapter we will show that the algorithms described in the previous chapter. To do so we will conduct qualitative experiments. To characterize the program we set up benchmarks to measure the performance of the program.

4.1 FPGA Configuration

4.1.1 Validate Pin Assignments

An interesting use case of the program is to verify the pin assignment for the FPGA on Enzian. This will help to catch errors before the time consuming process of building the bitstream and uploading it to the machine to see that it does not work.

To qualitatively show that this is possible to verify pin assignments we will look at an example of a real issue that was found working with the Enzian. The goal was to use a module which creates an interface from a four lane PCIe connection coming from the board of Enzian to a internal AXI-stream on the FPGA for an NVMe over PCIe application. This module is a so called "hard IP",so for this module to work properly the ports of the module have to be assigned to the respective FPGA pin receiving the signal from the board. The pin assignment can be changed for the FPGA using a constraint file. On the board we can track the signals from the connector, where the signal "enter" the board, to the FPGA, this is what we consider the truth and what we compare the pin assignments against. The PCIe connection has four lanes each of which has a transmit an receive differential pair, resulting in 16 wires. The module for the FPGA names the signals:

- pcie_7x_mgt_rtl_0_txp[<*0-3*>] are the four positive transmit signals.
- pcie_7x_mgt_rtl_0_txn[<*0-3*>] are the four negative transmit signals.
- pcie_7x_mgt_rtl_0_rxp[<*0-3*>] are the four positive receive signals.
- pcie_7x_mgt_rtl_0_rxn[<*0-3*>] are the four negative receive signals.

These signals are assigned by Vivado to pins of the XCVU9P FPGA. We will create three different setups for the synthesis of the design. One with the default constraint of the IP for the PCIe signals and two with constraints for the pin assignments.

Listing 4.1: Constraints A

```
set_property PACKAGE_PIN Y2 [get_ports {pcie_7x_mgt_rtl_0_rxp[1]}]
set_property PACKAGE_PIN W4 [get_ports {pcie_7x_mgt_rtl_0_rxp[2]}]
```

Listing 4.2: Constraints B

```
set_property PACKAGE_PIN {} [get_ports {pcie_7x_mgt_rtl_0_rxp[1]}]
set_property PACKAGE_PIN {} [get_ports {pcie_7x_mgt_rtl_0_rxp[2]}]

set_property PACKAGE_PIN Y2 [get_ports {pcie_7x_mgt_rtl_0_rxp[1]}]
set_property PACKAGE_PIN W4 [get_ports {pcie_7x_mgt_rtl_0_rxp[2]}]
```

For each of the synthesized designs we can export the pin assignment by first opening the synthesized design and then export the I/O ports under the menu `File > Export > Export I/O Ports...` and then export a CSV file. This file contains many different data points including the pin number and signal name. From the netlist we can extract how the PCIe connector connects to the FPGA. The PCIe connector we want has the designator `J27_4` and the FPGA has the designator `IC3`. With this knowledge we can load the netlist and use the program to extract the pin pairs:

Listing 4.3: Extract pin pairs from netlist

```
let netlist = Netlist::from_platform_netlist("netlist_enzian.json");

let connector = netlist.get_component_by_designator("J27_4");
let fpga = netlist.get_component_by_designator("IC3");

let connected_pins =
    netlist.get_connected_pins_of_components(connector, fpga);
...
```

On the connector the pin assignment is as follows, as can be read from the schematics. The pin names are used to represent the function of the pins but also the connected wires could be used as they convey the same information: Note: The table only shows the pins of the connector forwarding the receive and transmit wires of the PCIe lanes, the others (for example pins connected ground or a clock) are left out for simplicity.

Results

Running the synthesis with the three different constraints results in the following pin assignments. For increased readability the signal names shortened, but their meaning preserved. For example `pcie_7x_mgt_rtl_0_txp[3]` was changed into TX3+. Using the tool to extract how the connector pins are connected to the FPGA we get the following pairs: Table 4.4 is a result of combining 4.1 and 4.2 using table 4.3. The external signal is what we expect the connections to be on the connector `J27_4`. In the middle it is shown how the pins of the connector and FPGA are connected. No and A are the results of the synthesis with no constraint and constraints A which do not differ and B is the result of the synthesis with with constraints B. The colored entries are where the pin assignments were different. Red indicates that the signals do not match and green is to indicate that they do.

Table 4.1: Description of Connections

Pin Number	**Pin Name**
A4	RX1+
A5	RX1-
A7	RX3+
A8	RX3-
B4	RX0+
B5	RX0-
B7	RX2+
B8	RX2-
C4	TX1+
C5	TX1-
C7	TX3+
C8	TX3-
D4	TX0+
D5	TX0-
D7	TX2+
D8	TX2-

Table 4.2: Pin Assignment Results

Pin Nr.	**No Constraints**	**Constraints A**	**Constraints B**
AA9	TX3+	TX3+	TX3+
Y7	TX2+	TX2+	TX1+
W9	TX1+	TX1+	TX2+
V7	TX0+	TX0+	TX0+
AA8	TX3-	TX3-	TX3-
Y6	TX2-	TX2-	TX1-
W8	TX1-	TX1-	TX2-
V6	TX0-	TX0-	TX0-
AA3	RX3-	RX3-	RX3-
Y1	RX2-	RX2-	RX1-
W3	RX1-	RX1-	RX2-
V1	RX0-	RX0-	RX0-
AA4	RX3+	RX3+	RX3+
Y2	RX2+	RX2+	RX1+
W4	RX1+	RX1+	RX2+
V2	RX0+	RX0+	RX0+

Table 4.3: Pin Pairs

J27 Pin Nr	FPGA Pin Nr
A4	Y2
A5	Y1
A7	AA4
A8	AA3
B4	V2
B5	V1
B7	W4
B8	W3
C4	Y7
C5	Y6
C7	AA9
C8	AA8
D4	V7
D5	V6
D7	W9
D8	W8

Table 4.4: Combined Results

External Signal	J27 Pins	FPGA Pin	No and A	B
RX1+	A4	Y2	RX2+	RX1+
RX1-	A5	Y1	RX2-	RX1-
RX3+	A7	AA4	RX3+	RX3+
RX3-	A8	AA3	RX3-	RX3-
RX0+	B4	V2	RX0+	RX0+
RX0-	B5	V1	RX0-	RX0-
RX2+	B7	W4	RX1+	RX2+
RX2-	B8	W3	RX1-	RX2-
TX1+	C4	Y7	TX2+	TX1+
TX1-	C5	Y6	TX2-	TX1-
TX3+	C7	AA9	TX3+	TX3+
TX3-	C8	AA8	TX3-	TX3-
TX0+	D4	V7	TX0+	TX0+
TX0-	D5	V6	TX0-	TX0-
TX2+	D7	W9	TX1+	TX2+
TX2-	D8	W8	TX1-	TX2-

Interpretation

This clearly shows that it is possible to get useful information out of the netlist. In the netlist we can check how different components are connected and then use this information to check if properties at different parts of the board match up. In this case we check if the signals of a PCIe connection match up. Since the netlist and the information it contains does not change, it suffices to extract how the pins are connected once. In this experiment setup the converting of the signal names and the alignment of the result in table 4.2 was done manually. For use in practice it definitely makes sense to create a small script to automate this work, which could be done in future works investigating this.

4.1.2 Generating PACKAGE_PIN Constraints

In this subsection we will qualitatively show that we can generate PACKAGE_PIN properties. To be precise we will create these properties for one of the four memory connections.

To generate the PACKAGE_PIN property constraints for the signals of one of the four DDR4 DIMM 288 pin interfaces of the board of Enzian and compare the generated constraints with existing constraints for the FPGA [6]. The DIMM 288 connector we use in this example has the designator J17_5. To generate the constraint file we use the specification for the DDR4 DIMM modules. Pins related to power are left out as they do not need a pin constraint on the FPGA.

Listing 4.4: Section of the specification for the DIMM connector

```
{
    "part": "J17_5",
    "pin_signal_pairs": [
        {"pin_nr": "3","signal": "DQ4"},
        {"pin_nr": "5","signal": "DQ0"},
        {"pin_nr": "7","signal": "DQS9_t"},
        {"pin_nr": "8","signal": "DQS9_c"},
        {"pin_nr": "10","signal": "DQ6"},
        {"pin_nr": "12","signal": "DQ2"},
        ...
        {"pin_nr": "47","signal": "CB4"},
        {"pin_nr": "49","signal": "CB0"},
        ...
        {"pin_nr": "58","signal": "RESET_n"},
        {"pin_nr": "60","signal": "CKE0"},
        {"pin_nr": "62","signal": "ACT_n"},
        ...
        {"pin_nr": "68","signal": "A8"},
        {"pin_nr": "69","signal": "A6"},
        {"pin_nr": "71","signal": "A3"},
        {"pin_nr": "72","signal": "A1"},
        ...
        {"pin_nr": "87","signal": "ODT0"},
        {"pin_nr": "89","signal": "CS1_n"},
        {"pin_nr": "91","signal": "ODT1"},
        ...
        {"pin_nr": "285","signal": "SDA"}
    ]
}
```

To convert the signal names of the specification to the names used in the FPGA a function which does this conversion. The conversion is mostly turing the signal names to lower case and rearranging the exact layout and it adds the prefix ddr4_0_ to every signal. A few examples:

- DQ4 → fpga_0_dq[4]
- DQS9_t → fpga_0_dqs_t[9]
- CB0 → fpga_0_dq[0 + 64]

- RESET_n → fgpa_0_reset_n
- A1 → fpga_0_adr[0]

With the specification and the conversion function we can simply run the following code to generate the constraint file:

Listing 4.5: Generating a XDC file.

```
let netlist = Netlist::from_platform_netlist("netlist_enzian.json");
netlist::xdc::create_xdc_package_pins(
    &netlist,
    "J17_5",
    "IC3",
    "memory_spec.json",            // the specification for the pins
    "generated_constraints.xdc", // path of the output file
    conversion_function,          // function which converts the
                                  // specification signals to fpga
                                  // signals
);
```

After generating the constraint file it is compared to to the existing constraint file to see if it created different constraints.

Results

When generating the constraint file the following signals of the specification could not be assigned: EVENT_n, C2, SCL, SDA, SA0, SA1, SA2. These pins could not be assigned because there was not a unique pin found for them on the FPGA. To complete the generated constraint file the constraints for the remaining signals have to be created manually for example by inspecting the schematics.

Comparing the two constraint files yields the following differences:

Listing 4.6: Differences between generated and and existing constraint files

```
Generated:
    set_property PACKAGE_PIN AN29 [get_ports {ddr4_0_reset_n}]
    set_property PACKAGE_PIN AP30 [get_ports {ddr4_0_act_n}]
    set_property PACKAGE_PIN BC33 [get_ports {ddr4_0_par}]
    set_property PACKAGE_PIN BF35 [get_ports {ddr4_0_alert_n}]

Existing:
    set_property PACKAGE_PIN AN29 [get_ports ddr4_0_reset_n]
    set_property PACKAGE_PIN AP30 [get_ports ddr4_0_act_n]
    set_property PACKAGE_PIN BC33 [get_ports ddr4_0_par]
    set_property PACKAGE_PIN BF35 [get_ports ddr4_0_alert_n]
```

Interpretation

These results show that we can generate the PACKAGE_PIN property. The difference observed in the generated and existing is that in the existing file not every signal is in curly braces as in the generated (by design) every signal is. The curly braces are a way

to group information into words. [9] In this case they are used to make sure that the signal names are passed to the get_ports function as is. Without them the numbers in the square brackets of signal names would be treated as commands.

In this example the conversion function is very crucial bun in principle the specification could have been directly written converted.

4.2 I2C

To qualitatively show that we are able to extract the devices of an I2C bus we will extract the devices of an I2C bus of Enzian.

The I2C bus we will inspect is called F_I2C0This bus operates on two dwo different voltages which are connected through a level shifter.

Listing 4.7: Component specification detailing how signals going through the level shifter.

```
[
    {
        "component_type": "LSF0102DCUR",
        "connected_pins": [
            ["3","6"],
            ["4","5"]
        ]
    }
]
```

The wires used as an input were the wires with the name F_I2C0_1V2.SDA and F_I2C0_1V2.SCL.

Listing 4.8: Extracting I2C bus devices.

```
let netlist = Netlist::from_platform_netlist("netlist_enzian.json");

let sda = netlist.get_wire_by_name("F_I2C0_1V2.SDA");
let scl = netlist.get_wire_by_name("F_I2C0_1V2.SCL");

let i2c_components = get_i2c(
    &netlist,
    scl,
    sda,
    "component_spec.json"
);
```

The result of running this experiment is shown in table 4.5. The algorithm returned the FPGA and the four DDR4-DIMM connectors.

A close inspection of the schematics [3] shows that the result is indeed correct. This indicates that we are able to extract the I2C topologies.

4.3 Characterizing

In this section we will characterize the program we built in the thesis. To characterize the program we measured the the time it takes to open netlist and how long it takes to

Table 4.5: Extracting I2C bus devices result.

Designator	Component Type
IC3 XCVU9PFLGB2104	
J17_5	DDR4-DIMM SMD 288 pos
J17_8	DDR4-DIMM SMD 288 pos
J17_6	DDR4-DIMM SMD 288 pos
J17_7	DDR4-DIMM SMD 288 pos

explore an entire netlist using the implementation of the Search Connections algorithm (Algorithm 1).

Benchmark Environment

All measurements were made on a HP ProBook x360 435 G7. The laptop has a AMD Ryzen 5 4500u CUP and 16 gigabytes of memory. The tests were run under the Ubuntu 22.04.4 LTS and compiled with the rust compiler version 1.76.0. To run the benchmarks the Criterion rust crate (version 0.3.0) was used.[2]. The laptop was plugged in and in normal power mode.

4.3.1 Generating Netlists

To measure the performance of the program we need a way to generate netlist of different sizes. For simplicity we create a netlist with the following layout: The parameter n

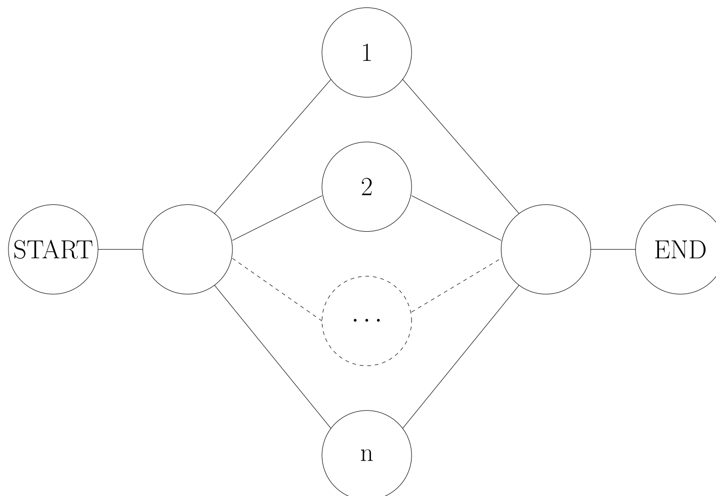


Figure 4.1: Illustration of the netlist generated for the characterization.

changing the size of the netlist defines how many components there are in the center of the netlist. The number of different elements of the netlist depends on n as shown in table. Observe that all the elements grow linearly as n grows, preserving the ration between them. 4.6 The Even if this structure is not very similar to an actual circuit it provides a scalable way to generate netlists to use in our experiments.

Table 4.6: Relation between n and the number of elements.

Pins	$4n + 4$
Components	$n + 4$
Wires	$2n + 2$
Total	$7n + 10$

4.3.2 Converting Netlists

In this subsection we will measure how long it takes to convert netlists from the form described in section 2.2 to the representation used internally by the program. This is a central part of the program because to use any netlist we have change it into the representation used by the program. To quantify this we will measure how long it takes to convert netlists of different sizes.

To measure the relation between the netlist size and the time it takes to convert it from the representation described in section 2.2 to the representation used by the program detailed in section 3.8. For the measurements netlists of the form described in the previous section were created with the values 500, 1000, 1500, 2000, 3000, 4000, 5000, 7500, 10000 and 15000 for n. The netlists were loaded into memory as we are not interested in how fast we can load the file from disk but the time it takes to convert the netlist. For each n 10 seconds of warmup executions were run and then 100 executions were timed.

The results of the experiment are shown in figure 4.2. The y-axis is the average execution time in milliseconds and the x-axis shows the size of the netlist.

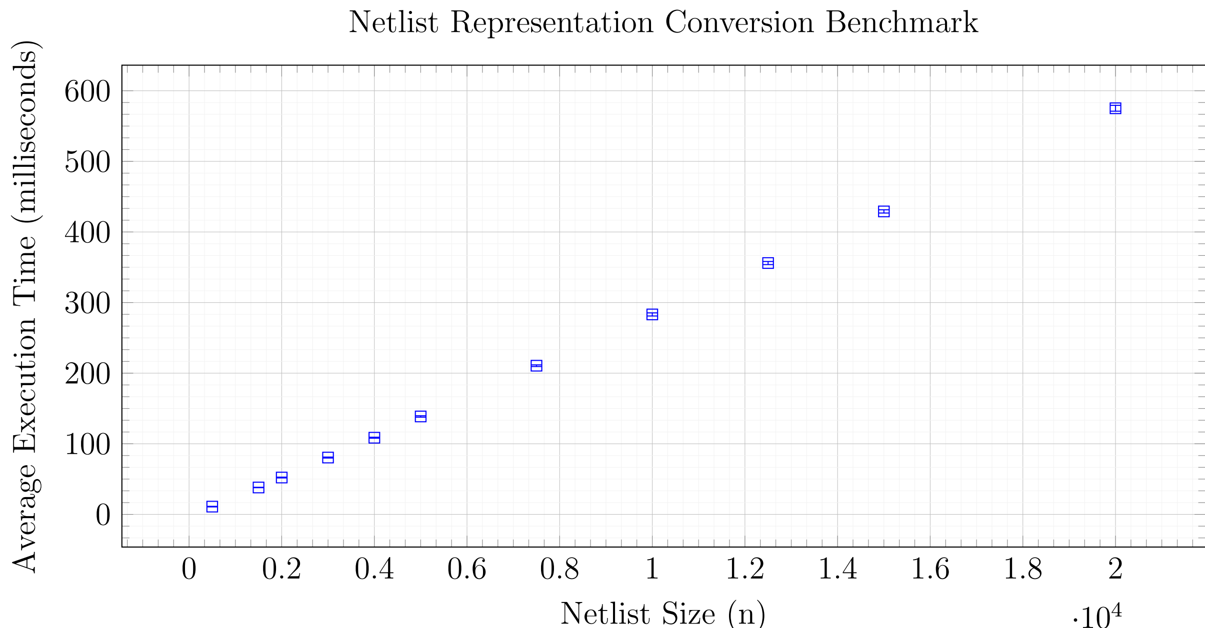


Figure 4.2: Benchmark results showing the relationship between average execution time to convert the netlist and netlist size.

The result show a clear linear relationship between the netlist size and the time it takes to convert it. For a netlist roughly of the size of the netlist of Enzian it takes about a tenth of a second to convert.

4.3.3 Connection Search

To understand how the program performs searching for connected pins in the netlist we measure the execution time for the implementation of 1 as the size of netlist increases.

To measure the relation between execution time and netlist size, different values to generate netlists as described above were used. The values of n used were 500, 1000, 1500, 2000, 3000, 4000, 5000, 7500, 10000 and 15000. The measurements were made in two batches. The fist batch was for the measurements up to and including n = 4000. For this batch 100 measurements were taken. The second batch measured from n = 5000 up to n = 15000. As the execution time increased fo the second batch only 25 measurements were taken. For each n first a netlist was generated and it was warmed up for 10 seconds. As the source pin the pin of the START component was used and as a target the END component. This way the whole netlist was visited by the algorithm.

Running the benchmarks we get the get the results shown in figure 4.3. The y-axis is the average execution time for the different runs in seconds. The x-axis shows how many center nodes were used in the netlist.

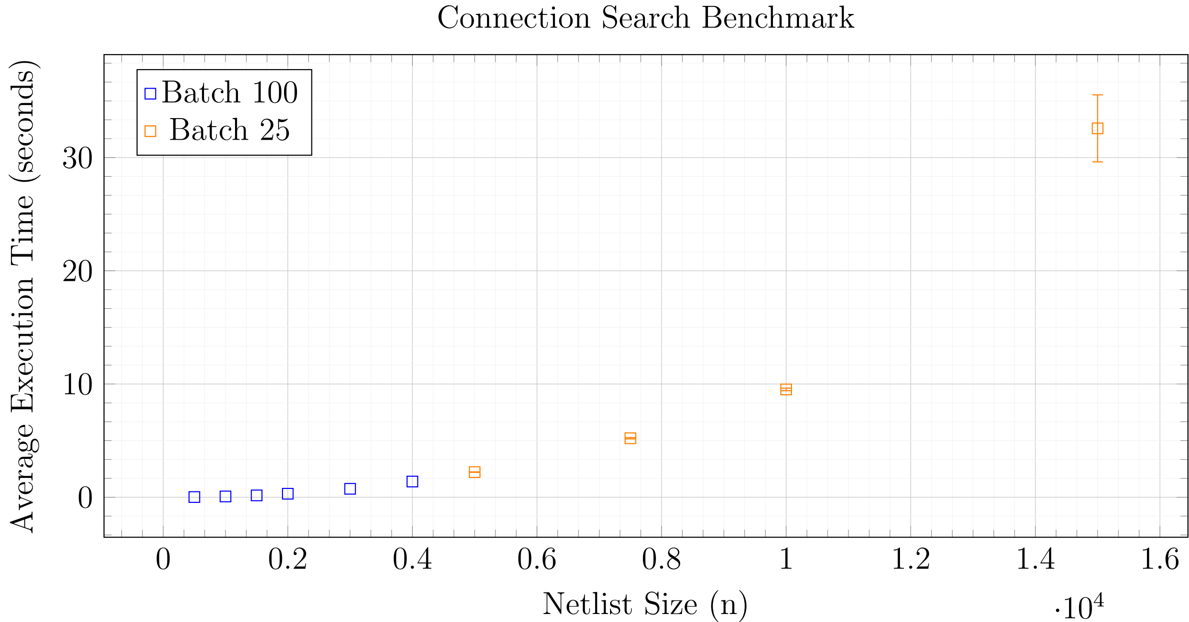


Figure 4.3: Benchmark results showing the relationship between average execution time and netlist size. Orange showing the data points with 25 iterations and and blue showing the data points with 100 iterations.

This represents a worst case for the algorithm as the whole netlist will be visited and hence gives an estimation for what sizes of netlists the program can deal with in reasonable time. To set this into perspective we can consider the netlist of Enzian which has 27090 elements in total, so a generated netlist with $n \approx 3850$ would be of a similar size. For this size it takes about a second for the function to finish. The measurements clearly show that the execution time grows faster than linear so for very large netlists will take a long time in the worst case.

Chapter 5

Conclusion

Platform configurations are crucial for systems software to run. These configurations can have different forms and can be tedious to create. Netlist can capture the topology of a system.

In this thesis we explored if we can use netlists to generate such system files. We were able to make an algorithm which can find connections within the netlist. This connectivity information has been successfully used to validate the pin assignments of a synthesized designs. We also were able to generate the PACKAGE_PIN property for Xilinx Design Constraint files with the help of specifications and a function transforming the signal names of the specification to names used in the code for the FPGA.

Additionally we have shown that it is possible to extract the topologies of I2C buses. To achieve this we take advantage of the I2C bus design, namely the fact that the bus has two wires.

To make it easier to work with the netlist and implement the algorithm discussed we created a different data layout. This was used to create a rust program which provides an interface for manipulating netlists.

Future Work

This thesis shows that netlists can be used to generate and validate simple system configurations. Subsequent investigations could expand this work by exploring if more properties of XDC files can be generated.

Further it would be interesting to explore if it is possible to extract more complex topologies from the netlist as for example power and clock distribution topology.

For both topics it could be interesting to investigate how the information in the netlist could be even further augmented.

Bibliography

[1] David Cock et al. “Enzian: An Open, General, CPU/FPGA Platform for Systems Software Research”. In: *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. Lausanne Switzerland: ACM, Feb. 2022, pp. 434–451. ISBN: 978-1-4503-9205-1. DOI: 10.1145/3503222.3507742. (Visited on 05/09/2024).

[2] *Criterion in Criterion - Rust*. https://docs.rs/criterion/latest/criterion/struct.Criterion.html#. (Visited on 05/26/2024).

[3] *Enzian Mainboard Schematics*. https://enzian.systems/documentation/. (Visited on 04/01/2024).

[4] *I2C-bus Specification and User Manual*. https://www.nxp.com/docs/en/user-guide/UM10204.pdf 2021. (Visited on 04/13/2024).

[5] Michał Kruszewski and Wojciech Marek Zabołotny. “Safe and Reusable Approach for Pin-to-Port Assignment in Multiboard FPGA Data Acquisition and Control Designs”. In: *IEEE Transactions on Nuclear Science* 68.6 (June 2021), pp. 1186–1193. ISSN: 1558-1578. DOI: 10.1109/TNS.2021.3074530. (Visited on 12/10/2023).

[6] *PROJECT-Enzian / Enzian FPGA · GitLab · Enzian_v3_fpga_constraints_ddr_included.Xdc*. https://gitlab.inf.ethz.ch/PROJECT-Enzian/enzian. July 2020. (Visited on 05/25/2024).

[7] Stephen Sangwine. *Electronic Components and Technology*. Tutorial Guides in Electronic Engineering. ISBN: 978-0-8493-7497-5.

[8] Jasmin Schult et al. “Declarative Power Sequencing”. In: *ACM Transactions on Embedded Computing Systems* 20.5s (Oct. 2021), pp. 1–21. ISSN: 1539-9087, 1558-3465. DOI: 10.1145/3477039. (Visited on 05/18/2024).

[9] *TCL Language*. https://www.tcl.tk/about/language.html. (Visited on 05/25/2024).

[10] *UltraScale and UltraScale+ FPGAs Packaging and Pinouts Product Specification*. https://docs.amd.com/viewer/book-attachment/pUJmWcGC˜qni8WKt2P3cEQ/8lPue6lfpx7Qp 2023. (Visited on 05/27/2024).

[11] *Vivado Design Suite User Guide Using Constraints*. https://www.xilinx.com/support/documents/ vivado-using-constraints.pdf. 2022. (Visited on 12/27/2023).

[12] *Vivado Overview*. https://www.xilinx.com/products/design-tools/vivado.html. (Visited on 05/09/2024).

[13] *What Are Netlists in PCB Design Projects?* https://resources.altium.com/p/what-are-netlists-pcb-design-projects. Jan. 2023. (Visited on 12/16/2023).

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every written paper or thesis authored during the course of studies. In consultation with the supervisor, one of the following three options must be selected:

(●) I confirm that I authored the work in question independently and in my own words, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used no generative artificial intelligence technologies[1].

( ) I confirm that I authored the work in question independently and in my own words, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used and cited generative artificial intelligence technologies[2].

( ) I confirm that I authored the work in question independently and in my own words, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used generative artificial intelligence technologies[3]. In consultation with the supervisor, I did not cite them.

Title of paper or thesis:

Generating Platform Configuration from Netlists

Authored by:
If the work was compiled in a group, the names of all authors are required.

Last name(s):	First name(s):
Wehrli	Georg

With my signature I confirm the following:
- I have adhered to the rules set out in the Citation Guide.
- I have documented all methods, data and processes truthfully and fully.
- I have mentioned all persons who were significant facilitators of the work.

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

Place, date	Signature(s)
Winterthur, 27.05.2024	G Wehrli

If the work was compiled in a group, the names of all authors are required. Through their signatures they vouch jointly for the entire content of the written work.

[1] E.g. ChatGPT, DALL E 2, Google Bard
[2] E.g. ChatGPT, DALL E 2, Google Bard
[3] E.g. ChatGPT, DALL E 2, Google Bard