Masteroppgave

NTNU
Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for elektroniske systemer

Patric André Berthelsen

Optimizing Routing Architectures for Small-Scale Heterogeneous Systems

Masteroppgave i Electronic Systems Design (MSELSYS)
Veileder: Snorre Aunet
Medveileder: Marchuk, Vitalii; Vestli, Snorre; Waagen, Johannes
Februar 2024

NTNU
Kunnskap for en bedre verden

Patric André Berthelsen

Optimizing Routing Architectures for Small-Scale Heterogeneous Systems

Masteroppgave i Electronic Systems Design (MSELSYS)
Veileder: Snorre Aunet
Medveileder: Marchuk, Vitalii; Vestli, Snorre; Waagen, Johannes
Februar 2024

Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for elektroniske systemer

NTNU

Kunnskap for en bedre verden

Preface

This thesis concludes my M.Sc degree in the Electronic Systems Design (MSELSYS) study programme, with a specialization in Embedded Systems at NTNU. The work was started in September 2023, and concluded in February 2024.

Many of the tools used in this work, such as the *generate_testarch* script, are in part borrowed from the *python-fpga-interchange* repository from CHIPS Alliance [1], which are released under an ISC free software license. However, the findings and modifications to the architectures and tools in question are entirely my own. The *parse_fasm* script was in part generated with the help of ChatGPT, with my own modifications to properly parse and analyze the data generated by the tools used in this work. Much of the few initial months went to debugging the open-source toolchain, where the code was adapted to fit both the repository mentioned, as well as some requirements from Microchip Technology, who acted as an external partner for this thesis.

Acknowledgments

There are a lot of people who deserve acknowledgment over the course of writing this work. I would first like to extend my many thanks to Microchip Technology for their generous hospitality and assistance. Not at least to my external supervisors: Vitaly Marchuk, Snorre Vestli and Johannes Waagen, who showed a generous amount of patience and guidance towards my many (many) questions. I would also like to thank Eirik Hollingen, who I worked alongside during the course of this work, for the many lovely chats and discussions, as well as the many coffee breaks. It was an absolute joy working with you, and I deeply appreciate it. Lots of luck with your thesis! At my alma mater, NTNU, I would like to thank my supervisor Snorre Aunet for his help throughout this work, despite busy schedules and a heap of other students with whom I had to share your time. Thank you all.

Additionally, I would like to thank Orbit NTNU for letting me use their office, the endless amount of coffee, and the absolutely wonderful people in the organization. You have shaped much of my time here at NTNU, and I am so grateful to have had the chance to work on some pretty amazing projects, and not at least to get to know all of you. I wish you all the luck in the future, and I am so excited to see where you go from here! :orbitluv:

Finally, I would like to extend my deepest gratitude to my family and friends for their love and support. Without you, this would not be possible. I love you all <3

Abstract

This work aimed to find a routing architecture for a small FPGA integrated alongside an MCU, based on specifications given by Microchip Technology. The design needed to fit MUX-based technologies, with no tri-state buffers. It needed to be scalable to some degree, could route most designs (within reason), and needed to be area efficient (regarding the number of MUX2 gates). This work looked at 2 different routing architectures, namely the route-through and bus architecture. In order to evaluate these architectures, then 5 different Verilog test designs were used (i.e., AND4, ADD2, SR4, SR8 and SR15), with the addition of a cost estimation metric, as well as an estimated route-through and logic utilization percentage of the architecture. The results showed that the route-through architecture could route all test designs with a grid size of (4, 8), with 7 INTRA wires and 4 INTER wires per tile. The architecture had around 6x the cost of previous architecture used by Microchip. The bus architecture failed to route any design, with a cost around 4x times that of the previous architecture used by Microchip. Most of the tests performed on the route-through architecture had a higher percentage of logic utilization for a grid size of (4, 8), and a higher percentage of route-through utilization for a grid size of (8, 4). Some tests did not work for a grid size of (8, 4) due to a lack of sufficient amount of inputs. This work also looked at the relationship between the INTRA and INTER parameters for the route-through architecture. The results showed a 37% variation in the total amount of placed blocks for the various tests, primarily for a grid size of (4, 8). The question of whether block placements or routing resources run out first on the device was also discussed, where ideally, block placement should be the factor that runs out first. Several observations were made that warrant further study. Occasional routing loop errors could be mitigated by the use of automated test scripts. Further development might also benefit from using a different PnR tool, with potentially a working GUI. A new metric for blocks that have both route-through and logic functionality could help gain more insight into routing congestion issues. Additional observations were made in terms of the PWR block and device clock.

Sammendrag

Dette arbeidet hadde som mål å finne en arkitektur for ruting av en liten FPGA integrert ved en MCU, basert på spesifikasjoner gitt av Microchip Technology. Designet måtte passe med MUX-baserte teknologier, uten noe form for tri-state bufring. Designet måtte være skalerbart til en hvis grad, den måtte kunne rute de fleste design og måtte være areal-effektivt med hensyn til antall MUX2 porter. Dette arbeidet så på 2 forskjellige arkitekturer, det vil si route-through og bus-arkitekturen. For å kunne evaluere de forskjellige arkitekturene, så ble det benyttet 5 forskjellige Verilog test design (AND4, ADD2, SR4, SR8 og SR15), i tillegg til en kost-estimator, samt en prosentverdi for estimert route-through og logikk utnyttelse av arkitekturen. Resultatene viste at route-through arkitekturen kunne rute de fleste design med en størrelse på (4, 8), med 7 INTRA ledninger og 4 INTER ledninger per tile. Arkitekturen hadde ca 6x så stor kostnad som den forrige arkitekturen fra Microchip. Bus-arkitekturen klarte ikke å rute noen design, med en kostnad ca 4x så stor som den forrige arkitekturen fra Microchip. De fleste testene gjennomført på route-through arkitekturen hadde en høyere prosentandel logikk utnyttelse for en størrelse på (4, 8), samt en høyere prosentandel route-through utnyttelse for størrelsen (8, 4). Noen tester funket ikke for størrelsen (8, 4), gitt manglende antall innganger. Dette arbeidet har også sett på forholdet mellom INTRA og INTER parameterene for route-through arkitekturen. Resultatene viste en 37% variasjon i total mengde plasserte blokker for forskjellige tester, primert for størrelsen (4, 8). Spørsmålet om enten antall plasseringer eller rute-ressurser ble brukt opp først var også diskutert, hvor ideelt sett så burde antall plasseringer bli brukt opp først. Flere observasjoner ble gjort under dette arbeidet som burde være et tema for videre forskning. Sporadiske routing-løkke feil kan bli unngått ved bruk av et form for automatisert test skript. Videre utvikling kan også ta nytte av å bruke et annet PnR verktøy, med en potensielt funkende GUI. En ny måleverdi for blokker som har både route-through og logikk funksjonalitet kan være nytting for å få mere innsikt i tilfeller av kongestion av rute-ressurser. Ytterligere observasjoner ble gjort i forhold til PWR blokker og enhetsklokken.

Contents

Figures

Tables

Acronyms

API Application Programming Interface. 9, 16

FASM FPGA Assembly. 4, 6, 9, 10, 63

FOSS Free and open-source software. 4, 34, 63

FPGA Field Programmable Gate Array. iii, iv, vii, ix, x, 1–7, 9, 28

GUI Graphical User Interface. iii, iv, 28, 31

HDL Hardware Description Language. 4

I/O Input / Output. 1, 7

IB Input buffer. 9, 10

IOB Input/Output buffer. 9

IoT Internet of Things. 2

IP Intellectual property. 4

LUT Lookup Table. 2, 5, 16, 27

MCU Microcontroller unit. iii, iv, 1, 2

MUX multiplexer. iii, iv, 2, 5, 8, 16, 22, 25

OB Output buffer. 9, 10

PnR place and route. iii, iv, vii, 4, 5, 9, 11, 19–21, 23, 25–29, 31, 34, 38, 49, 65–69

RTL Register Transfer Level. xi

SoC System on a chip. 1

VPR Versatile Place and Route. 28, 31

VTR Verilog-to-Routing. 4, 28

Glossary

BEL CHIPS Alliance defines a BEL as an "abbreviation of basic logic element. A BEL can be one of 3 types, site port, logic, routing. A BEL contains 1 or more BEL pins" [2]. v, vii, x, 2, 5, 6, 9

bitstream Binary file with configuration data for logical elements on FPGA fabric [3]. 4, 6

cell CHIPS Alliance defines a cell as "a logical element of a design that contains some number of cell ports and some number of cell instances, and some number of nets" [2]. 5, 9

CLB Configurable Logic Block. Same as BEL. viii, 6, 7, 9, 10, 13, 16, 19–23, 26–31

device CHIPS Alliance defines a device as "a set of tiles and package pins" [2]. 5, 9

logic BEL CHIPS Alliance defines a logic BEL as "a placeable logic element. May be subject to 0 or more placement constraints" [2]. 5

net CHIPS Alliance defines a net as "a set of logically connected cell ports" [2]. 5

netlist A description of a circuit in terms of gates and connections, which meet the timing and power requirements of the circuit [4]. xi, 4

node CHIPS Alliance defines a node as "a set of 1 or more wires that are connected. Nodes can span multiple tiles. Nodes connect to PIPs or site pins via the wires that are part of the node" [2]. 9

PIP CHIPS Alliance defines a PIP as "an abbreviation for programmable interconnect point. A PIP provides a connection between two wires. PIPs can be unidirectional or bidirectional. Unidirectional PIPs always connect wire0 to wire1. Bidirectional PIPs can connect wire0 to wire1 or wire1 to wire0" [2]. 5–7, 16, 21, 22

routing BEL CHIPS Alliance explains that "a routing BEL connects at most 1 input BEL pin to the output BEL pin" [2]. 5

site CHIPS Alliance defines a site as "a collection of site pins, site wires and BELs" [2]. 5, 9, 10

site port BEL CHIPS Alliance explains that "a site port BEL represents a connection to a site pin contained within the parent tile of the site" [2]. 5

techmap Transformation of RTL netlist into equivalent netlist using target architecture cell mapping (i.e. cell substitution, sub-circuit substitution or gate-level technology mapping) [5]. 4

tile CHIPS Alliance defines a tile as "an instance of a tile type which contains wires and sites" [2]. 5, 9, 10

wire CHIPS Alliance defines a wire as "a piece of conductive material totally contained within a tile. Wires can be part of nodes. Wires can connect to PIPs or site pins" [2]. 9, 10

XDC Xilinx Design Constraint (XDC) file is based on Synopsis Design Constraints format (SDC). Xilinx explains that the constraints "define the requirements that must be met by the compilation flow in order for the design to be functional on the board" [6]. 11–14

Chapter 1

Introduction

The capacity and popularity of Field Programmable Gate Array (FPGA) systems have increased since they were first introduced by Xilinx in 1984. Trimberger [7] explains that FPGA technology development has gone through 3 distinct ages. In the first age, FPGA sizes were quite small, with equally as small design problems, and were generally smaller than the applications that users wanted to run on them. Most of the development was limited to manual placement of logical and physical designs. The second age saw an increase of FPGA capacity relative to Moore's law, with an increased demand for design automation. Here, area could be traded off for performance, enhanced features and ease of use. The third age saw a change in the target application, with an adaption of FPGA technology for communications infrastructure, with higher real-time performance requirements (e.g., quick development, field-upgradable architecture) and a focus on high-performance features (e.g., high-speed I/O, wide datapaths). The period also began to see an end to Dennard scaling. Trimberger [7] further explains that today, the industry has a desire for lower cost and power consumption. With increased programmability for SoC devices, user-programmable logic typically only occupies less than half the area of FPGA devices. Today, we have fewer gains from technology scaling than before, with design effort and risk being more central requirements, in part aided by more capable design tools and methods. [7]

1.1 Motivations

The motivations for this work come from the efforts of Microchip Technology to scale down a traditional FPGA architecture for upcoming Microcontroller unit (MCU) product lines. Here, the smaller, more limited FPGA fabric is tightly connected to the MCU. This design aims to make programmable logic more available to users who are not necessarily digital designers, both through simplified but sufficient architecture design and through the use of open-source toolchains for logic design, synthesis and placement/routing. Some applications include offloading tasks from the main processor to small amounts of programmable logic (e.g., non-traditional I/O operations that normally were dealt with by the use of software interrupt

routines on the main processor). It is important to note that these are low-cost products with analog components that would not see much benefit from moving to smaller technology nodes, as the nodes currently used for these products are as small as possible without facing major development issues related to technology downscaling. The size of the FPGA is limited to around 32 logical elements, based on requirements from Microchip Technology.

Previous works, such as the PIC16F13145 [8], uses much of the same concepts and will be used as reference design throughout this work. An illustration of a basic BEL can be found in Figure 2.1. A basic logical element in the PIC16 architecture consists of a Lookup Table (LUT) with some routing. The design uses 22-input MUXs per element for global routing. Note that the design is only optimized for a grid with 32 total elements, with a limited global routing network that was not meant for an island-style arrangement.

1.2 Objectives

This work aims to explore possible architectures for a small FPGA integrated alongside an MCU. The design must fit MUX-based technologies, with no tri-state buffers. The design needs to be scalable to some degree, can route most designs (within reason), and must be area efficient (with regards to the number of MUX2 gates). Due to potential intellectual property rights, the architecture exploration must be performed using open-source tools (i.e., yosys and nextpnr).

1.3 Sustainable Development Goals

In accordance to the UN Sustainable Development Goals [9], this work mainly complies with goal 9 *Industry, Innovation and Infrastructure*, but can be extended to goals 2, 3, 6, 7, 11, 12, 13, 14 and 15 due to the wide fields that apply to microchip technologies. Applications include (but are not limited to) sensors and monitoring with Internet of Things (IoT) devices (aquaculture, agriculture, pollution), medical technology, energy monitoring etc.

1.4 Structure

The rest of this work is structured as follows:

Chapter 2 *Background* covers most of the theoretical background the reader needs to know to understand the methods, results and discussion surrounding this work.

Chapter 3 *Methods* covers the methods used to produce the results used in the discussion. Section 3.1 covers the technique used to produce the cost estimation metric for each routing architecture. Section 3.2 covers the different scripts used in this work to produce the test architecture definitions, as well as the FPGA grid plot used for visual inspection and evaluation (with statistics for route-through vs logical connections). Section 3.3 covers the test designs that are synthesized for the purposes of evaluating the different routing architectures.

Chapters 4 and 5 cover the different routing architectures used in this work in more detail.

Chapter 6 *Results* lays out the results obtained from running the different test designs, as described in section 3.3, through the test architectures described in chapters 4 and 5. The results are generated from the scripts in section 3.2.

Chapter 7 *Discussion* tries to evaluate the different test architectures laid out in chapters 4 and 5 against the results laid out in chapter 6, as well as evaluating the results against the design requirements presented in section 1.2. Recommendations for future work are given in section 7.4, with a conclusion to these discussions in Chapter 8 *Conclusion*.

Chapter 2

Background

2.1 F4PGA

The F4PGA toolchain [10] is developed by Chips Alliance, which is hosted by The Linux Foundation, as a non-profit organization that develops and hosts high quality, open source hardware code (Intellectual property (IP) cores), interconnect IP (physical and logical protocols), and open source software development tools for design, verification, and more.

The goal of the F4PGA toolchain is to create a complete Free and open-source software (FOSS) FPGA toolchain, with several tools and projects to provide needed components of a full open-source end-to-end flow. [10]

The frontend and backend tools in the toolchain (i.e., *Yosys*, *nextpnr* and Verilog-to-Routing (VTR)) use certain resources as inputs, mainly the FPGA "*architecture definitions*" (documents how the FPGAs work internally) and the *interchange schema* (2.2) (logical and physical netlists)
The general flow [10] usually follows the outlined steps:

1. Prepares info about timing/resources available during the implementation stage and techmaps for synthesis tools. This info is generally given through the architecture definitions.
2. Logic synthesis is carried out by the *Yosys* framework, which translates user-provided hardware description via a Hardware Description Language (HDL) into the block and connection types available for the chosen architecture.
3. Implementation, where a place and route (PnR) tool will put individual blocks from a synthesis description into specific chip locations and create paths/routing between them. This is done through the *nextpnr* or VTR tools.
4. Design properties are translated into a set of features available for the given chip, which is saved in the FPGA Assembly (FASM) (2.5) format. This file is then translated into a bitstream that can be flashed to the given chip.

2.2 FPGA interchange format

The FPGA interchange format [2] describes the logical and physical placements within an FPGA, and aims to give a complete description of an island-type FPGA design. CHIPS Alliance [2] divides a device into 4 parts:

1. A device, which is defined as a set of tiles and package pins
2. Tiles, which contains wires and sites
3. Sites, which contains site pins, site wires and BELs.
4. BELs, which is a basic logic element. Can be one of 3 types: site port BEL, logic BEL or routing BEL. Contains 1 or more BEL pins.

2.2.1 Mapping guidelines

Nets, as defined by CHIPS Alliance [2], can be placed into 3 categories:

- Signal net, where the signal is neither a constant logical 0 or constant logical 1
- Constant logical 0: listed as "GND" type
- Constant logical 1: listed as "VCC" type

Both constant logical 0 or constant logical 1 can have multiple drivers in the device description. [2]

2.2.2 Cell placement and driver BEL pins

CHIPS Alliance [2] explains that BELs represent a placeable location for a cell, and only one cell can be placed at a given BEL. This results in the fact that the cell library and BEL design strongly affects what can be expressible by the place and route tool. [2]

2.3 BEL

Trimberger [11] explains that a BEL is composed of a LUT, multiplexers and wiring channels that connect to PIPs. If a block is unused, the output PIPs are turned off to avoid driving other signals. A BEL can implement both sequential and combinatorial logic. E.g. to build a latch, a LUT is configured with the proper truth table, and the output of the block is routed back to the input. One can also configure inputs for reset/set signals and clock inputs. These latches can be combined to form other kinds of sequential elements like D-type flip-flops. An illustration of a BEL is shown in Figure 2.1.

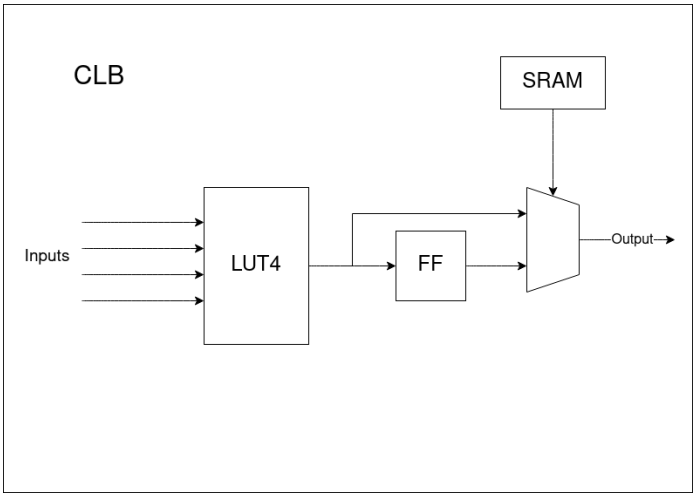


Figure 2.1: Illustration of a BEL

2.4 PIP

Trimberger [11] explains that a PIP controls the connection of wire segments in a programmable interconnect, and is typically implemented as a pass-transistor configured by a memory cell. The wire segments on each side of the transistor are connected depending on the configuration set in the memory cell.

2.5 FASM

CHIPS Alliance [12] explains that the FPGA Assembly (FASM) file format is designed to specify the bits in an FPGA bitstream that need to be set or cleared. This is implemented by enabling different "features" within the bitstream. An empty FASM file will generate a platform-specific "default" bitstream.

The following is an example of a FASM file output. Here, the different connections within an FPGA are set by enabling the feature. Line 2 shows that the *INTRA_4* wire within the CLB block is active and connected to the *INP_N_6* wire within the CLB via a PIP.

```
# Created by the FPGA Interchange FASM Generator (v0.0.18)
CLB_X1Y0.INTRA_4.INP_N_6
CLB_X1Y0.INTRA_5.FROM_SLICE0_Q_0
CLB_X1Y0.INTRA_7.INP_N_7
CLB_X1Y0.OUT_E_2.INTRA_5
CLB_X1Y0.OUT_E_6.INTRA_7
```

2.6 Island-style FPGA

Betz et al. [13] state that an island-style routing architecture for FPGAs consists of CLBs surrounded by routing channels (wire segments) on all four sides. I/O pins connect to some or all wire segments via PIPs. Interconnect PIPs (also known as *switch blocks*) connects between wire segments to form longer wires that can span the FPGA. This is illustrated in Figure 2.2.

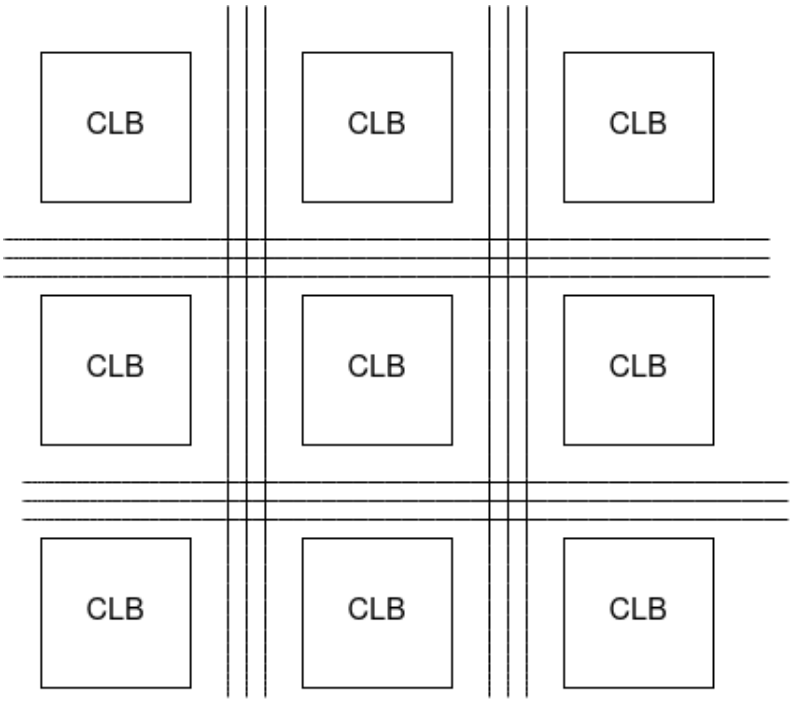


Figure 2.2: Illustration of an island-style FPGA routing architecture

Chapter 3

Methods

3.1 Cost estimation

In order to get a good basis for comparing the different architectures in this work, we need an estimator that is architecture and technology independent. Note that this estimator only gives a relative comparison between the different architectures and will need to be supplemented with other forms of data to get a full overview for comparing and evaluating different solutions.

The proposed cost estimator is based on the total number of MUX2 in the architecture, which might give an indication as to the switching and routing complexity introduced by an architecture. The list below shows the weight values used for the cost estimation, which is based on the areal size of logical elements normalized to an equivalent number of MUX2. Note that these values are based on data given by Microchip Technology. These can give a rough estimate as to the cost of different components.

Weight values:

- MUX2: 1
- DEMUX2: 1
- Config bit: 2
- Flip-flop: 3

A general cost function can be derived from looking at the construction of a MUX. In this equation, N represents the number of inputs to the MUX. In addition, a weight value of *log2(N)* must be added to represent the number of config bits that are needed to switch between the different inputs to the MUX. Note that N also needs to be normalized to a value of log2, which reflects the real-life MUX width which is implemented on a physical device (i.e. $N = 5$ needs a MUX width of 8 to realize in practice $=> N = 8$)

$$MUXN => N + floor(log2(N)) \tag{3.1}$$

3.2 Scripts

3.2.1 generate_testarch

The *python-fpga-interchange* repository from CHIPS Alliance [1] is used in this work to generate an *fpga-interchange* format description of the desired FPGA architecture. The workflow used in this work primarily focuses on the *generate_testarch.py* script. This script sets up the test architecture by generating the required site types (i.e., Input/Output buffer (IOB), power and *slice*), arranges these tiles in a grid format, connects these different tiles via nodes and wires as well as generating primitives, parameters and cell/BEL mappings for the different tiles. The script acts as a frontend that generates a Python object model, which is passed to a *capnproto* API [14] to generate serialized binary data that is passed between different programs (i.e., nextpnr and other Python scripts).

Some modifications were made to the original architecture [1] in order to have a successful run of the PnR tool. The modifications were mostly made in the functions and methods related to the IOB cells, to reflect the more simplified design currently used by Microchip as compared to the original test architecture from CHIPS Alliance. The choice was also made to remove the IOB site types in favor of a simpler scheme that only dealt with IB and OB cells. This led to a design where the first and last column of the device grid was filled with IB and OB cells respectively, and where the top and bottom rows were filled with CLBs instead of IOB cells.

In order to represent the different architectures in this work, modifications were made to the *make_tile_type*, *make_device_grid* and *make_wires_and_nodes* functions. An additional function, *make_bus_type*, was made to represent the combined bus and route-through architecture.

3.2.2 parse_fasm

The *parse_fasm* script is used in this work to process and visualize the connections made in the FASM file, which is generated after a run of the PnR tool. This script is divided into 3 main functions: *plot*, *calc_util* and *parse_fasm_file*. The entire source code can be found in the Appendix section A.4. **Note that this script is in part generated with the help of ChatGPT**.

The *parse_fasm_file* function is used to read from the FASM format file generated by the PnR tool. The data from this file is read into the *fasm_connections* dictionary, which is passed between the different functions in this script.

The *calc_util* function is primarily used to generate statistics regarding the percentage of logic blocks vs. the percentage of generated route-through blocks in the design. Here, a *logic block* is defined as a CLB tile that routes a path through the internal

SLICE site where logical operation is performed, and a *route-through* is defined as a block that is exclusively used to connect to different tiles without routing through the SLICE site. These values are generated by looking at the different sections for each tile placed in the FASM file. If a tile has a connection to a "*TO_SLICE*" or "*FROM_SLICE*" wire, and is not specifically an *IB* or *OB* tile (i.e. exclusively CLBs), then we increment the *num_logic_blocks* counter value. The *block_counts* dictionary stores the total number of lines found per block type. These values are used to get a total block count of the design, which is used to find the number of route-throughs vs. the number of logic blocks.

The *plot* function is used to visualize the design using the *matplotlib* Python library. An X-Y grid is generated based on the max values for the X and Y coordinates listed for the different tiles in the *fasm_connections* dictionary. The different tiles are illustrated as blue boxes, with red arrows illustrating the different directional connections generated between the tiles. Note that only *IB*, *OB* and *PWR* tiles are given a label in the figures generated by the script, in order to improve the readability of the figure for bigger grid sizes. The arrows are drawn by evaluating whether the current iterated element has an OUT wire pointing in the north, south, east or west direction, which is given explicitly in the name of the wire. In which case, a directional offset will be used to plot the connection.

Figure 3.1 shows an example of a grid plot generated using the *parse_fasm* file.

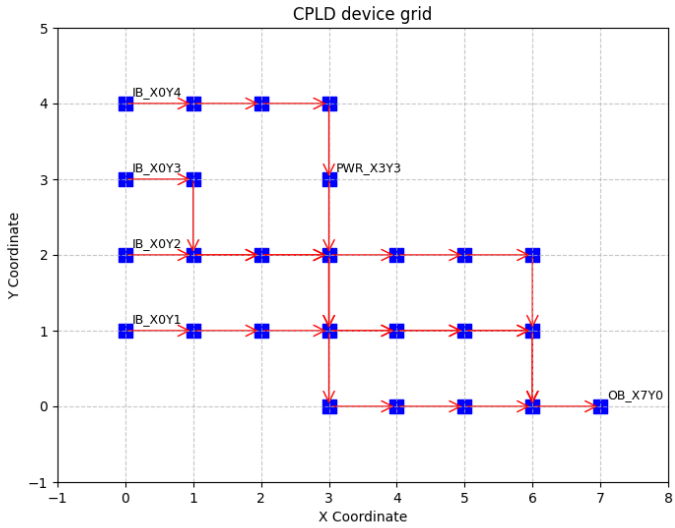


Figure 3.1: Example of a grid plot generated using the *parse_fasm* file

3.3 Test designs

This work uses 5 separate Verilog test designs to evaluate the different test architectures. This section will give a brief overview of how they function, with listings for the Verilog source code, as well as the XDC description for each test design.

3.3.1 AND4

The AND4 test design is a Verilog design of a 4-input AND gate. The AND gate utilizes 4 inputs compared to a regular 2-input design due to a bug in the *nextpnr* PnR software regarding cell mapping for gates with floating inputs. The purpose of the AND4 test is to have a simple design that works as a sort of baseline when comparing the different tests.

The source code for the design is shown below in Listing 3.1.

Code listing 3.1: Listing of AND4 test Verilog source code

```
module top (
    input  wire a,
    input  wire b,
    input  wire c,
    input  wire d,
    output wire y
);

    assign y = a & b & c & d;

endmodule
```

The XDC file for the design is shown below in Listing 3.2.

Code listing 3.2: Listing of AND4 test XDC file

```
set_property PACKAGE_PIN I_0 [get_ports a]
set_property PACKAGE_PIN I_1 [get_ports b]
set_property PACKAGE_PIN I_2 [get_ports c]
set_property PACKAGE_PIN I_3 [get_ports d]
set_property PACKAGE_PIN O_0 [get_ports y]
```

3.3.2 SR4

The SR4 test design is a Verilog design of a 4-bit Shift register. An illustration of the design is given in Figure 3.2. Note that the design distributes the shared enable signal (*en*) across all register elements. The purpose of the shift register test designs in this work is to observe how the architectures under test respond to designs with high fan-out signals, such as the enable signal (*en*).

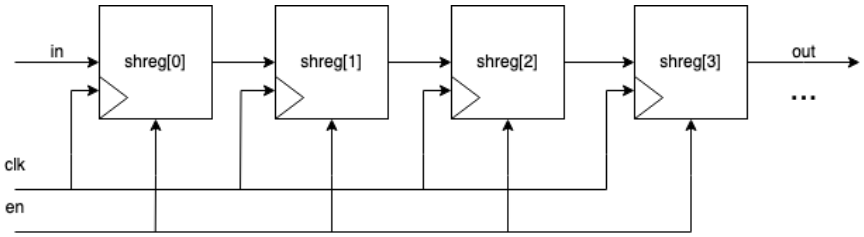


Figure 3.2: Illustration of a 4-bit shift register (SR4) test design

The source code for the design is shown below in Listing 3.3.

Code listing 3.3: Listing of SR4 test Verilog source code

```
module top (
    input clk, in, en,
    output out);

    parameter WIDTH = 4;

    reg [WIDTH-1:0] shreg;

    always @(posedge clk)
    begin
        if (en)
            shreg <= {shreg[WIDTH-2:0], in};
    end

    assign out = shreg[WIDTH-1];
endmodule
```

The XDC file for the design is shown below in Listing 3.4.

Code listing 3.4: Listing of SR4 test XDC file

```
set_property PACKAGE_PIN I_0 [get_ports clk]
set_property PACKAGE_PIN I_1 [get_ports in]
set_property PACKAGE_PIN I_2 [get_ports en]
set_property PACKAGE_PIN O_0 [get_ports out]
```

3.3.3 SR8

The SR8 test design is a Verilog design of an 8-bit Shift register. The design is identical to the SR4 test design, but with the WIDTH parameter adjusted for 8 elements. This does not affect the XDC file for the design.

3.3.4 SR15

The SR15 test design is a Verilog design of a 15-bit Shift register. The design is identical to the SR4 and SR8 test designs, but with the WIDTH parameter adjusted for 15 elements, which covers the maximum amount of placeable area for CLBs on the device. This does not affect the XDC file for the design.

3.3.5 ADD2

The ADD2 test design is a Verilog design of a 2-bit carry adder. An illustration of the design is given in Figure 3.3. The purpose of ADD2 test design is to have an application that occupies a significant amount of total block placements on the device.

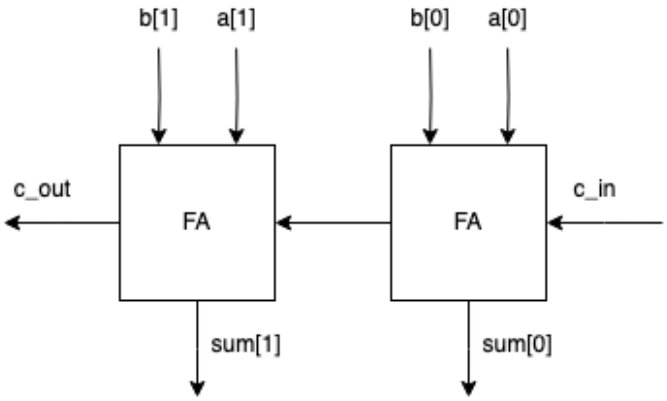


Figure 3.3: Illustration of a 2-bit carry-adder (ADD2) test design

The source code for the design is shown below in Listing 3.5.

Code listing 3.5: Listing of ADD2 test Verilog source code

```
module fulladd(
    input [1:0] a,
    input [1:0] b,
    input c_in,
    output c_out,
    output [1:0] sum
);

    assign {c_out, sum} = a + b + c_in;
endmodule
```

The XDC file for the design is shown below in Listing 3.6.

Code listing 3.6: Listing of ADD2 test XDC file

```
set_property PACKAGE_PIN I_0 [get_ports c_in]
set_property PACKAGE_PIN I_1 [get_ports a[0]]
set_property PACKAGE_PIN I_2 [get_ports a[1]]
set_property PACKAGE_PIN I_3 [get_ports b[0]]
set_property PACKAGE_PIN I_4 [get_ports b[1]]

set_property PACKAGE_PIN O_0 [get_ports c_out]
set_property PACKAGE_PIN O_1 [get_ports sum[0]]
set_property PACKAGE_PIN O_2 [get_ports sum[1]]
```

Chapter 4

Route-through architecture

The *route-through architecture* consists of 2 major parts:

1. The tile, which sets up connections between the SLICE and inputs/outputs
2. The device grid, where the individual tiles are placed and routed

The tile is illustrated in Figure 4.1. The architecture allows configuring N number of INTER-connections (input/output arrows) per direction. Inside the tile, we can configure W number of INTRA wires, which can connect to all TO, FROM, INP (input) and OUT (output) wires, where it partly acts as an internal bus. This enables the tile to route signals both directly to the SLICE, and *through* the tile, where an input from any direction can be routed to an output in any other direction.

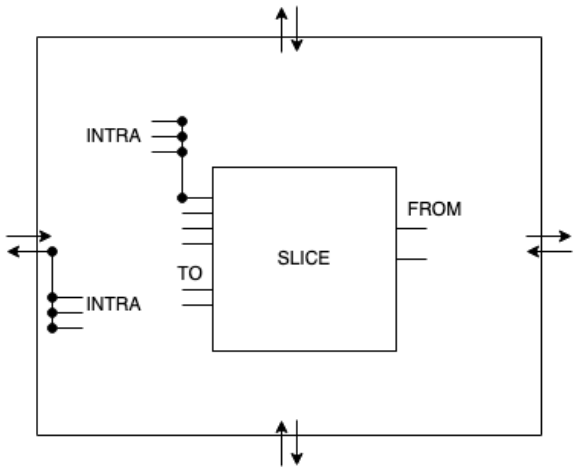


Figure 4.1: Illustration of tile architecture for the *route-through* test architecture

The SLICE contains a modified CLB (see section 2.3), which gives the option to override the LUT in favor of supplying a separate D signal which goes into the AFF flip-flop. Figure 4.2 depicts the modified SLICE, where input L0-L3 represents the inputs to the LUT, signal D is the signal which can be swapped out for the output of the LUT, and signal Q is the output of the AFF flip-flop. Both signals named O are intermediate signals between the LUT, MUX and flip-flop, respectively. The O signal in the top right corner of the figure is meant to be used as an output for different PIPs positioned between the inputs and outputs of the MUX. The mechanism for this is omitted in the context of this work.

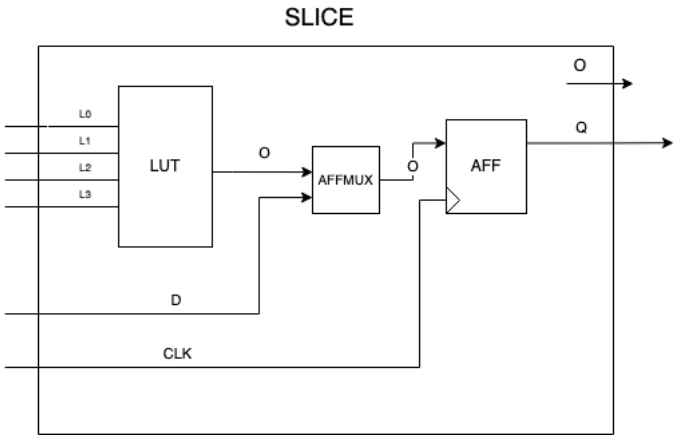


Figure 4.2: Illustration of SLICE architecture for the *route-through* test architecture

The design is in part borrowed from the CHIPS Alliance GitHub repository *pyhton-fpga-interchange* [1]. This repository provides an intermediate step between the *generate_testarch* Python script, and the *capnporoto* API that converts the Pythonic object model into a binary file (see section 3.2). The script, and hence the architecture, is modified to fit some of the technical requirements of the previous architecture from Microchip, especially regarding input and output buffers.

The architecture is arranged in an island-style format (2.6), with tiles arranged in a square. This is illustrated in Figure 4.3. Each tile can connect directly to its neighbors in each cardinal direction (north, south, east and west), with the exception of the IB and OB tiles. This contributes to the lack of long-range global routing, where signals need to be routed through the different tiles instead of routing them through a form of global network (e.g., busses). This assumes that each tile has enough INTRA and INTER-connections to enable all the required signals to be routed through.

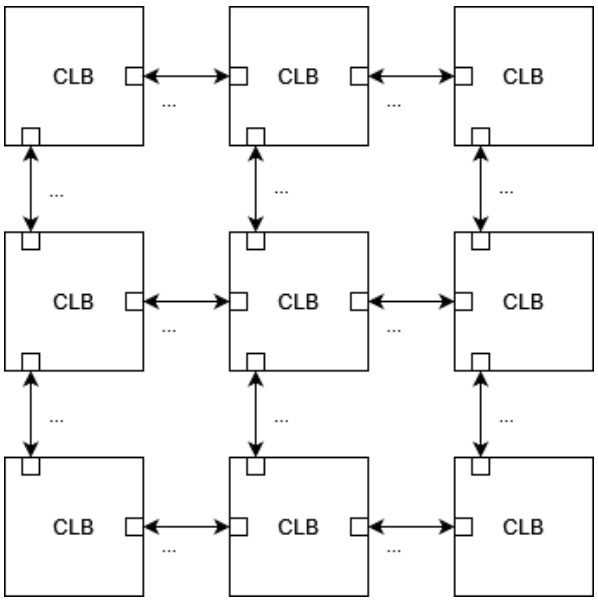


Figure 4.3: Illustration of device grid for the *route-through* test architecture

This architecture has some underlying assumptions and simplifications that are important to address:

1. Each tile has an excessive routing capability, which may introduce a more significant cost than might be needed. This must be taken into account when configuring the number of INTRA and INTER-connections per tile.
2. The INTRA wires scheme can introduce cases where input and output signals to a tile can be routed to the same type of signals (i.e., input to input, output to output). This work attempts to solve this problem with the second architecture detailed in the next chapter.
3. The clock (CLK) signals are routed as a general signal, meaning that the signals get routed through the same cardinal directional mechanism in each tile. In order for the design to fit some of the technical requirements of the previous architecture from Microchip, then the clock signals need to be routed in parallel to each individual tile, to ensure that all tiles are equally supplied with the same clock reference.

The listing for this script is located in appendix A.2.

Chapter 5

Bus architecture

The *bus architecture* consists of two major parts:

1. The tile, which sets up connections between the SLICE and input/outputs, with the addition of a BUS input/output.
2. The device grid, where the individual tiles are placed and routed, with the addition of the BUS tile which enables a common bus across individual rows/columns in the grid.

The tile is illustrated in Figure 5.1. The tile architecture is almost identical to the tile illustrated in chapter 4, Figure 4.1, with the notable exception of the BUS lines connected to the SLICE FROM wires, which enable some global routing across multiple tiles. We can also note that the outputs from the tile are connected to the SLICE FROM wires, in order to avoid the problems mentioned in chapter 4 regarding input/output signals where they potentially can be routed to signals of the same type. A solution for route-through signals is introduced by means of wires that directly connect the INP and OUT wires of the tile via route-through (RT) wires. The left-hand side of the figure shows the same INTRA wire scheme as mentioned in chapter 4. For this architecture, only the inputs to the tile and the TO wires use the INTRA wires scheme.

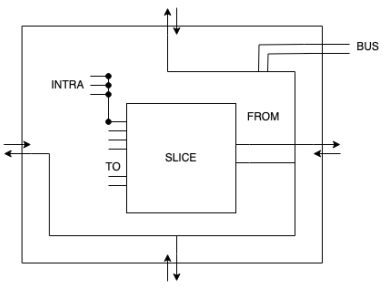


Figure 5.1: Illustration of tile architecture for the *bus* test architecture

The architecture follows the same island-style grid system as in chapter 4. In addition, BUS tiles have been added that provide a common bus for tiles in both horizontal and vertical directions, depending on what produces the best results. The common bus is primarily used to enable routing of some global signals (e.g., high fan-out signals). Each tile has an individual BUS input and output wire. The BUS tile can switch between individual outputs that get fed into the common bus, depending on what configuration is set. This scheme is illustrated in Figure 5.2.

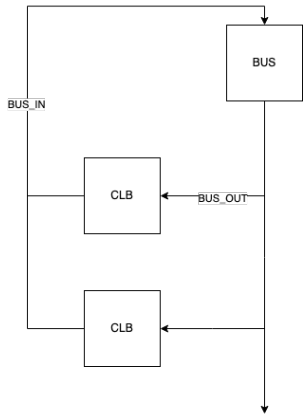


Figure 5.2: Illustration of a part of the device grid for the *bus* test architecture

This architecture has some underlying assumptions and simplifications that are important to address:

1. The grid needs an extra row/column to accommodate the BUS tile, depending on whether the design needs a vertical or horizontal bus.
2. Some route-through (RT) wires have been introduced in order to solve the problem of INP and OUT wires connecting to other wires of the same type, as described in chapter 4.
3. The BUS tiles can not switch between the different CLB tiles at run-time. The configuration of the BUS tile is exclusively decided by the PnR tool depending on the most optimal routing configuration.
4. The architecture has the same clock distribution problem as chapter 4.

The listing for this script is located in appendix A.3.

Chapter 6

Results

6.1 Place and route

The following section gives an overview of the results from the place and route (PnR) runs for the different test designs as described in section 3.3. The following grid plots generated by the *parse_fasm* script (see section 3.2) are located in appendix A.5. Note that the total number of placed blocks for each test design does not include IB and OB tiles, which results in a total number of CLB blocks equal to 16 for a grid size of (4, 8), and 24 for a grid size of (8, 4).

6.1.1 Route-through

The following list shows the results obtained from the different PnR runs for the different test designs as described in section 3.3.

- AND4: The AND4 test produced a successful run of the PnR tool given a grid size of (4, 8), with 5 INTRA wires and 2 INTER wires. This produced an estimated 87.5% of route-through tiles, where the remaining 12.5% were logic tiles. The tool placed a total of 8 blocks. This amounts to 50% of the total number of available CLBs. Note that the design did not give a successful run with the grid size of (8, 4), as the tools produced warnings of insufficient amount of input tiles in the design.
- ADD2: The ADD2 test produced a successful run of the PnR tool given a grid size of (4, 8), with 7 INTRA wires and 4 INTER wires. This produced an estimated 44.44% of route-through tiles, where the remaining 55.55% were logic tiles. The tool placed a total of 9 blocks. This amounts to ca 56% of the total number of available CLBs. Note that the design did not give a successful run with the grid size of (8, 4), as the tools produced warnings of insufficient amount of input tiles in the design.
- SR4: The SR4 test produced a successful run of the PnR tool given a grid size of (4, 8) and (8, 4), with 5 INTRA wires and 2 INTER wires. This produced an estimated 33.33-69.23% of route-through tiles, whereas the remaining 30.77-66.66% were logic tiles. The tool placed a total of 12 and

13 blocks for each relative grid size. This amounts to ca 75% and 54% of the total number of available CLBs, respectively. Note that the tools produced occasional routing loop errors across the different runs.

- SR8: The SR8 test produced a successful run of the PnR tool given a grid size of (4, 8) and (8, 4), with 6 INTRA wires and 2 and 3 INTER wires for each relative grid size. This produced an estimated 42.86-52.94% of route-through tiles, whereas the remaining 47.06-57.14% were logic tiles. The tool placed a total of 14 and 17 blocks for each relative grid size. This amounts to ca 87% and 70% of the total number of available CLBs, respectively.
- SR15: The SR15 test produced a successful run of the PnR tool given a grid size of (4, 8) and (8, 4), with 6 INTRA wires and 2 and 3 INTER wires for each relative grid size. This produced an estimated 6.25-37.5% of route-through tiles, where the remaining 62.5-93.75% were logic tiles. The tool placed a total of 16 and 24 blocks for each relative grid size. This amounts to 100% of the total number of available CLBs.

All designs are shown to work with a grid size of (4, 8), with 7 INTRA wires per TO/FROM wire, and 4 INTER wires per tile. The designs produced an estimated 6.25-87.5% route-through tiles, with 12.5-93.75% of the tiles including logic. Note that these vary per run given the random seed that is provided at that moment to the toolchain. Some tests also produced occasional routing loop errors across different runs.

6.1.2 Bus

The current version of the bus architecture does not compile for any grid size. The output of the PnR tools shows that the main router loop in *nextpnr* runs continuously, and can not manage to successfully route any design.

6.2 Cost estimation

The following section shows the cost estimates for the different test architectures evaluated in this work. The estimates are derived from the formulas and tables in section 3.1. The left-hand column of the different rows shows the name of the specific elements. The middle column shows what type of element, in terms of the number of *MUX2* in which the different parts of the design are translated into. The right-hand column shows the estimated cost of the specific element. The bottom of the table shows the total sum of the estimated costs. Note that all costs are estimated based on tile data (i.e., structure of a single tile in the design), as PIPs can only be placed at *tile level*, which limits the placement of MUX2 elements to a per tile basis.

6.2.1 Microchip design

The cost estimates for the CLB architecture in the PIC16F13145 [8] are shown in Table 6.1. Using the formula for cost estimation of an N-input MUX as shown in equation 3.1, a cost for a 22-input MUX is estimated to a value of 27. When adding the values for the DFF and DFFMUX, a total sum of 114 is estimated for the entire tile. This value will be used as a reference against the different architecture designs presented in this work.

DFF	Flip-flop	3
DFFMUX	MUX2	3
LUT0	MUX22	27
LUT1	MUX22	27
LUT2	MUX22	27
LUT3	MUX22	27
	Sum	114

Table 6.1: Microchip CLB cost estimate

6.2.2 Route-through

The cost estimates for the *route-through* architecture are shown in Table 6.2. Note that the first 10 rows show the cost for the SLICE, while the remaining rows show the cost for the design specific to the *route-through* architecture. Using the parameters for the INTRA and INTER wires given in section 6.1.1, then each TO and FROM wire will be split between 7 different connections, which would need a MUX8 to realize in practice. The INTER parameter would need a MUX4 to realize in practice. The cost of a MUX8 is estimated to a value of 11 using equation 3.1, while the cost of a MUX4 is estimated to a value of 6. The input/output and TO/FROM wires will also need a cost value for the different PIPs introduced in the *make_tile_type* function. With a number of PIPs equivalent to the INTRA and INTER parameters for each cardinal direction, then we get a cost estimate of 528 for the INP and OUT wires. For the TO and FROM wires, then each number of wire has INTRA number of PIPs attached, which gives a cost value of 88.

The total sum is estimated to 710, which is around 6.2x times the cost of the PIC16 CLB tile.

TO_CLK	MUX8	11
TO_D	MUX8	11
TO_L0	MUX8	11
TO_L1	MUX8	11
TO_L2	MUX8	11
TO_L3	MUX8	11
FROM_O	MUX8	11
FROM_Q	MUX8	11
DFF	Flip-flop	3
DFFMUX	MUX2	3
INP/OUT	2(INP/OUT)*INTRA*INTER*4(DIR)	528
TO/FROM	(6(TO)+2(FROM))*INTRA	88
	Sum	710

Table 6.2: Route-through architecture tile cost estimate

6.2.3 Bus

The cost estimates for the *bus* architecture are shown in Table 6.3. Note that the first 10 rows show the cost for the SLICE, while the remaining rows show the cost for the design specific to the *bus* architecture. As no parameter for the bus architecture produces a successful run of the PnR tool, as stated in section 6.1.2, then we need to base our estimates on parameters used in the *route-through* architecture. The cost estimation uses an INTRA value of 7, INTER value of 4 including a bus width of 2, and a number of bus ports equivalent to the height of the device (i.e., y value of 8).

Table 6.3 shows that the cost of the BUS_OUT connections is equivalent to the number of bus ports multiplied by the bus width, which gives a cost estimate of 33. The route-through wires that exclusively connect the tile inputs to the tile outputs are calculated for each cardinal direction, for both inputs and outputs, and multiplied with the INTER parameter, which gives a cost estimate of 48. The inputs produce a cost estimate for each cardinal direction, with the product between the number of INTRA and INTER wires, which gives a cost estimate of 264. The outputs follow much of the same equation, but have to account for the OUT_FROM wires compared to the number of INTRA wires, which gives a cost estimate of 48.

The total sum is estimated to 487, which is around 4.3x times the cost of the PIC16 CLB tile.

TO_CLK	MUX8	11
TO_D	MUX8	11
TO_L0	MUX8	11
TO_L1	MUX8	11
TO_L2	MUX8	11
TO_L3	MUX8	11
FROM_O	MUX8	11
FROM_Q	MUX8	11
DFF	Flip-flop	3
DFFMUX	MUX2	3
BUS_OUT	BUS_PORTS*BUS_WIDTH	33
RT	4(DIR)*2(INP/OUT)*INTER	48
INP	4(DIR)*INTER*INTRA	264
OUT	4(DIR)*INTER*2(OUT_FROM)	48
	Sum	487

Table 6.3: Bus architecture tile cost estimate

Chapter 7

Discussion

7.1 Problem statement

In order to evaluate the different test architectures, we need to determine whether or not they satisfy the design requirements laid out in section 1.2. In summary, the design needs to fit a MUX-based technology (this excludes tri-state technology), it needs to be scalable, should be able to route most designs and needs to be area-efficient. In addition, the design should be mostly limited to 32 elements to reflect the previous Microchip design, as described in section 1.1.

The different routing architectures presented in this work do not specifically require any tri-state technology to function correctly. The architectures are scalable, but have an impact on cost estimation. Comparatively, the route-through architecture has around 6.2x higher cost than the PIC architecture, while the bus design has around 4.3x higher cost than the PIC architecture. This would make the bus architecture a more viable candidate simply based on cost.

As stated in section 6.1.1, all the designs presented in this work produce a successful PnR run with a grid size of (4, 8) with 7 INTRA wires per TO/FROM wire, and 4 INTER wires per tile. Some tests (i.e., AND4 and ADD2) did not work with the grid size of (8, 4), where the tools reported that they lacked a sufficient amount of input tiles. This is to be expected as the designs in question require a significant amount of inputs compared to the relative device grid size. Note that these parameters act as the minimum amount of INTRA and INTER wires for a successful run of the PnR tool. However, these might not be optimal for all applications that the user wants to upload to the device, as the test designs used in this work are relatively simple in terms of complexity. There is also an open question regarding the effects of modifying the INTRA and INTER parameters and their relative effect on each other. This topic is further discussed in section 7.3.

7.2 Bus design

As stated in section 6.1.2, then the current version of the bus architecture does not give a successful run of the PnR tool for any grid size. The tools report that the main router loop in *nextpnr* runs continuously, and can not manage to successfully route any design. At the time of writing, the reasons for this is unknown. Some observations made when running the PnR tool point to a failure point being introduced when adding the BUS tile, as well as the OUT_FROM and RT wires to the *route-through* architecture. A leading theory is that the tools can not properly utilize the available routing resources in the BUS tile and OUT_FROM/RT wires. However, providing evidence for this theory is difficult, given the lack of insight into how the tools solve these problems.

Do note that despite the current lack of a successful implementation in this work, one might still benefit from trying to implement a similar solution in potential future work. The major benefits here are centered around future designs requiring less INTRA and INTER resources for routing signals with high fan-out (such as the enable signal present in the SR test designs).

7.3 Resource bottlenecks

7.3.1 Note on metrics

Before we can properly begin to discuss the resource usage of the architectures in question, then we first need to make a few notes regarding the metrics that we use in this work and how these are utilized. In general, it is quite difficult to measure the effectiveness of a system solely based on a single metric. Utilization, for example, only takes into account the effectiveness of the PnR tool, and how successful it has been when utilizing the available resources in the architecture. A higher route-through percentage in this case could indicate that the architecture has too few routing resources in a tile, and thus has to spread the datapath across more tiles than necessary (type of "routing congestion"). However, this depends on the size of the application that you want to run on the device. If the application takes up very little space on the device (say a few tiles near the input blocks), then a higher route-through percentage would only show the relative distance from the application to the output tiles, as more route-through blocks needs to be placed in order to reach the OB tile. Here, the percentage of logic blocks as a metric is a bit easier to interpret, as it can only indicate the number of placements on the device, which shows the size of the application that the user wants to run.

When looking more closely at a single tile, one can observe, depending on the amount of INTRA resources available, that there is no functional difference between pure logic and route-through blocks. The same route-through blocks can be used as logic blocks and vice versa, i.e., there is no functional lock in a CLB tile. This lack

of a functional difference between the types of blocks might affect the utilization statistics of a device, especially in regards to routing congestion issues as explained above.

The relationship between the INTRA and INTER parameters also has an impact on routing. Optimally, the INTRA parameter should increase the internal routing capabilities of a tile (especially for route-through signals), and the INTER parameter should increase the parallel routing capacity to neighbor tiles. Both parameters are interdependent in terms of routing, where a tile needs to have a sufficient amount of internal resources available to route signals both to the LUT and neighbor tiles, and it needs to have a sufficient amount of parallel resources available for applications that have a high amount of routing congestion. This is reflected in the CLB SLICE, which requires a minimum number of 4 INTRA wires to connect the inputs of the tile to the SLICE LUT. Additional INTRA wires are required to enable further route-through signals. Both the INTRA and INTER parameters also have an impact on the cost of the system.

7.3.2 Routing resource vs. placement

When looking at the variation in the total number of placed CLBs for each test design, we can observe a variation between 50% and 87% between the different tests (i.e., around 37% variation). This is excluding the SR15 test, which is designed to utilize the whole device to test for routing congestion. Most of the high percentage of placed blocks seems to come when using a grid size of (4, 8), where there is a shorter relative distance between the IB and OB tiles. It could be that the PnR tool needs to extend the routing of the design throughout the columns, but this does not match the relatively low percentage of route-through utilization for the different test designs with the given grid size. There is a higher percentage of route-through utilization when using the grid size (8, 4), although with a lower percentage of total placed blocks.

In an ideal situation when it comes to routing resources vs. placement, then the available amount of placed blocks should run out first, as a user application should be able to occupy the whole device while facilitating for more routing between the different blocks. This is supported by the results for the SR15 test with a grid size of (4, 8), which is designed to occupy the whole area of the device. This test has a high logic utilization percentage and a small route-through percentage (this excludes placed tiles that have route-through and logic utilization), with 6 INTRA wires and 3 INTER wires per tile. The design uses more route-through utilization when using a grid size of (8, 4), which is most likely in order to reach the OB tiles. This is visualized in the grid plots for the (4, 8) test (A.7) and for the (8, 4) test (A.8).

The ADD2 test seems to be a special case regarding routing resources. When

running a test for grid size (4, 8), then the design needs 7 INTRA wires and 4 INTER wires per tile in order to route successfully. In addition, the design uses a majority of logic utilization for only around 56% of the total amount of placed CLBs. This could be a result of routing congestion issues where the design might need more routing resources in order to spread the routing paths throughout neighboring tiles, although this sort of behavior is usually expected for designs that use almost 100% of the available amount of CLB blocks.

When it comes to the question of routing resources vs. placement, then it is quite difficult to draw any firm conclusions based on the data presented in this work. In general, it seems like available placements run out first, but there are some cases where more routing resources are required than what is strictly necessary (especially regarding the ADD2 test as mentioned above). This is also dependent on the relative grid size, where more blocks seem to be placed when using a grid size of (4, 8), with a majority percentage of logic utilization. The test designs also show a higher route-through percentage when using a grid size of (8, 4), where they most likely need more CLB blocks in order to reach an OB tile. However, more tests are needed to see how routing behaves in edge cases, as the tests used in this work are generally quite small in size (with the exception of the SR15 test design).

7.4 Future work

Several observations were made during the course of the creation of this work that warrant further study. One such observation was the occasional routing loop errors that appeared on some seeds of the test designs, as explained in section 6.1.1. These errors add more difficulty when trying to find the minimal parameters needed for routing a design, as it adds a level of uncertainty to the data. It is also optimal to find parameter values that work across all seeds, especially from a user perspective, where one would want an architecture that can route as many designs as possible. One way to test this is to use a form of automated test tool that can run different test designs (with a varying amount of total block placement) across as many different seeds as possible, while sweeping through different parameter combinations. This is especially true if the route-through architecture is chosen for further study

Some of the problems that appeared when using the chosen PnR tool might be mitigated by changing to a new tool for further studies. Shah et al. [15] explains that Versatile Place and Route (VPR) is a PnR tool that is part of the Verilog-to-Routing (VTR) open-source framework, which in part focuses on FPGA architecture exploration and research. This tool is more used in academia, and might offer more support in terms of debugging and architecture exploration compared to *nextpnr*, which worked better at the time of writing this work. Getting a working GUI for the PnR tool might also be helpful when trying to debug further routing architectures.

One tool that might help when looking at routing congestion issues (see section 7.3) might be to add a new utilization metric. The current metrics used in this work do help to gain more insight into the relationship between exclusive route-through and logic tiles, but come short in cases where we have blocks that have both route-through and logic enabled by the PnR tool. A metric that shows the percentage of such blocks in a system can be quite helpful in edge cases where most available CLB blocks have been placed by the tool, but some signals still need to be routed to neighboring tiles without the help of exclusive route-through blocks.

Some additional observations that were made when running the different test designs through the PnR tools is that the PWR block, which is supposed to connect the working design to VCC and GND respectively, seems to be included in several cases as a route-through block. In some cases, the PWR block serves a dual purpose, where it both connects the design to VCC/GND, and serves as a route-through block. This is part of the *generate_testarch* design borrowed from CHIPS Alliance [1], which is used in this work. This is a feature that might not be wanted for further designs, and might need some revisions. The same applies to the device clock, which gets routed as a standard signal (i.e., sent through an IB tile and routed through CLB tiles until reaching an OB tile). It might be more beneficial to run the clock parallel to all CLB tiles for further designs, to avoid cases where the clock signal has a lower routing priority than other signals.

Chapter 8

Conclusion

In conclusion, the route-through architecture presented in this work meets the problem statement detailed in section 1.2. The architecture is MUX-based, mostly scalable and has relatively low cost compared to the bus architecture. It can route most designs with a grid size of (4, 8), with 7 INTRA wires and 4 INTER wires per tile. Note that these are the minimal parameters needed for a successful run. However, the architecture is still around 6x as costly as the previous architecture used by Microchip (see section 1.1). Most of the tests had a higher percentage of logic utilization for a grid size of (4, 8), and a higher percentage of route-through utilization for a grid size of (8, 4). This might be due to the longer distances between the input and output tiles, where more route-through blocks must be added for a design to reach the output, especially for smaller designs placed closer to the input tiles. Some tests (i.e., AND4 and ADD2) did not work with a grid size of (8, 4), where the tools reported that the device lacked a sufficient amount of input tiles to accommodate the test designs.

The bus architecture did not work for any test design, with a cost around 4x times that of the previous architecture used by Microchip. The architecture might have a failure point when introducing the BUS tile and OUT_FROM and RT wires. However, further work can still draw some benefits from this architecture for signals with high fan-out.

In general, it is quite hard to measure the effectiveness of a system with a single metric. This work used cost, device utilization and INTRA/INTER parameters as metrics to evaluate different architectures. Some of the discussion related to this work focused on the effects of high route-through utilization vs. high logic utilization. Other parts of the discussion focused on CLB blocks that have no functional lock, especially in terms of exclusive route-through or logic functionality, which can affect further results. Routing effects resulting from changing INTRA and INTER parameters were also discussed. Here, the INTRA parameter should increase the internal routing capabilities of a tile, and the INTER parameter should increase the parallel routing capacity to neighboring tiles.

The results from running the test designs through the PnR tool showed around 37% variation in terms of the total number of placed CLB blocks, mostly using a grid size of (4, 8). This might be caused by the tools having to extend the routing path through the device column, but this is not supported by the relatively low percentage of route-through tiles present.

When it comes to the question of routing resources vs. placement, then ideally placement should be the factor that runs out first. This is supported by the findings of the SR15 test using a grid size of (4, 8). The ADD2 test is a special case, where the design needs more routing resources than necessary in order to successfully route the design. This might be due to routing congestion issues, especially when factoring in the relatively few amount of placed blocks in the design. It is quite difficult to draw any firm conclusions when it comes to the question of routing resources vs. placement. More tests are needed to see how routing behaves in edge cases, as the test designs used in this work are generally quite small in size.

During the course of this work, several observations were made that warrant further study. Occasional routing errors appeared when running several test designs. These can be mitigated by using a form of automated test tool that can run different test designs (with a varying amount of total block placement) across many different seeds, while sweeping through different parameter combinations. Further development might also benefit from using a different PnR tool like VPR, which is more supported in academia. A working GUI is also quite helpful when debugging different routing architectures. When it comes to routing resources vs. placement, adding a new metric for tiles that have both route-through and logic utilization can lead to more insight into edge cases that show routing congestion issues. Additional observations were made regarding the PWR block, which in some cases acted as a route-through block, which might not be wanted behavior for future designs. In addition, the current architectures route the device clock as a standard signal. This should be sent in parallel to all CLB tiles.

Bibliography

[1] *Python-fpga-interchange/fpga_interchange/testarch_generators/generate_testarch.py at master · chipsalliance/python-fpga-interchange*, en. [Online]. Available: https://github.com/chipsalliance/python-fpga-interchange/blob/master/fpga_interchange/testarch_generators/generate_testarch.py (visited on 05/10/2023).

[2] *Device Resources — FPGA Interchange Format 0.0-99-gc985b46 documentation*. [Online]. Available: https://fpga-interchange-schema.readthedocs.io/device_resources.html (visited on 26/09/2023).

[3] *Bitstream format — Project X-Ray 0.0-3807-g72e6371b documentation*. [Online]. Available: https://f4pga.readthedocs.io/projects/prjxray/en/latest/architecture/bitstream_format.html (visited on 29/12/2023).

[4] M. G. Jaiswal, V. S. Bendre and V. Sharma, 'Verilog Netlist Rearrangement Technique in Microwind,' in *2017 International Conference on Computing, Communication, Control and Automation (ICCUBEA)*, Aug. 2017, pp. 1–4. DOI: 10.1109/ICCUBEA.2017.8463881. [Online]. Available: https://ieeexplore.ieee.org/document/8463881 (visited on 29/12/2023).

[5] *9. Technology mapping*. [Online]. Available: https://yosyshq.readthedocs.io/projects/yosys/en/latest/CHAPTER_Techmap.html (visited on 29/12/2023).

[6] *Migrating From UCF Constraints to XDC Constraints • Vivado Design Suite User Guide: Using Constraints (UG903) • Reader • AMD Adaptive Computing Documentation Portal*. [Online]. Available: https://docs.xilinx.com/r/en-US/ug903-vivado-using-constraints/Migrating-From-UCF-Constraints-to-XDC-Constraints (visited on 12/01/2024).

[7] S. M. Trimberger, 'Three Ages of FPGAs: A Retrospective on the First Thirty Years of FPGA Technology: This Paper Reflects on How Moore's Law Has Driven the Design of FPGAs Through Three Epochs: The Age of Invention, the Age of Expansion, and the Age of Accumulation,' *IEEE Solid-State Circuits Magazine*, vol. 10, no. 2, pp. 16–29, Jun. 2018, MAG ID: 2809370239. DOI: 10.1109/mssc.2018.2822862.

[8] 'PIC16F13145 Family Full-Featured 8/14/20-Pin Microcontrollers,' en, Tech. Rep., 2023. [Online]. Available: https://ww1.microchip.com/downloads/aemDocuments/documents/MCU08/ProductDocuments/DataSheets/PIC16F13145-Family-Microcontroller-Data-Sheet-DS40002519.pdf.

[9] UN General Assembly, *Transforming our world : The 2030 Agenda for Sustainable Development*, Oct. 2015. [Online]. Available: https://documents-dds-ny.un.org/doc/UNDOC/GEN/N15/291/89/PDF/N1529189.pdf?OpenElement.

[10] *How it works — F4PGA documentation*. [Online]. Available: https://f4pga.readthedocs.io/en/latest/how.html (visited on 21/09/2023).

[11] Stephen M. Trimberger, 'Field-Programmable Gate Array Technology,' Jan. 1994, MAG ID: 1604136049.

[12] CHIPS Allience, *FPGA Assembly (FASM) — FPGA Assembly (FASM) 0.0.2-100-gffafe82 documentation*. [Online]. Available: https://fasm.readthedocs.io/en/latest/ (visited on 25/10/2023).

[13] V. Betz, J. Rose, Alexander Marquardt and A. Marquardt, 'Architecture and CAD for Deep-Submicron FPGAS,' *The Springer International Series in Engineering and Computer Science*, Mar. 1999. DOI: 10.1007/978-1-4615-5145-4.

[14] *Cap'n Proto: Introduction*. [Online]. Available: https://capnproto.org/ (visited on 13/11/2023).

[15] D. Shah, E. Hung, C. Wolf, S. Bazanski, D. Gisselquist and M. Milanović, 'Yosys+nextpnr: An Open Source Framework from Verilog to Bitstream for Commercial FPGAs,' 2019, Publisher: arXiv Version Number: 1. DOI: 10.48550/ARXIV.1903.10407. [Online]. Available: https://arxiv.org/abs/1903.10407 (visited on 23/01/2024).

Appendix A

Additional Material

A.1 Makefile

The following listing shows the makefile for the FOSS toolchain used in this work for synthesis and PnR (see section 3.2).

Code listing A.1: Makefile for FOSS toolchain

```
PYTHON   ?= $(shell which python3)
# FASMGEN  ?= fasm_clhs.py
FASMGEN  ?= fasm.py
SCHEMA   ?= fpga-interchange-schema/interchange
ARCHBUILDER ?= generate_testarch.py
# ARCHBUILDER ?= arch.py
YOSYS    ?= $(shell which yosys)
NEXTPNR  ?= $(shell which nextpnr-fpga_interchange)
BBASM    ?= nextpnr/bba/bbasm
VPR      ?= $(shell which vpr)
DEVICE   ?= cla_cpld_v1
DESIGN   ?= design
CNP_PATH ?= capnproto-java/compiler/src/main/schema

# FIXME: VPR commandline args are subject to change!
VPR_ARGS ?= \
    --arch_format fpga-interchange \
    --circuit_format fpga-interchange \
    --timing_analysis off \
    --timing_driven_clustering off \
    --echo_file on \
    --netlist_verbosity 999 \
    --pack_verbosity 999 \
    --clustering_pin_feasibility_filter off \
    --route_chan_width 100 \
    --constant_net_method route \
    --clock_modeling route \
    --check_rr_graph off \
    --generate_rr_node_overuse_report on\
    --timing_report_detail detailed\
    --write_block_usage block_usage.txt\
    --save_routing_per_iteration off

PNR_EXTRA_ARGS ?= -v
```

```
# PNR_EXTRA_ARGS ?= -v --debug
# PNR_EXTRA_ARGS ?= -v --gui

# DESIGN_DIR     = /home/app/plugin/clhs-plugin/tests
DESIGN_NAME    = $(DESIGN)
DESIGN_DIR     = /home/app/design
# DESIGN_VERILOG = $(DESIGN_DIR)/and.v
# DESIGN_XDC     = $(DESIGN_DIR)/and.xdc
DESIGN_XDC     = $(wildcard $(DESIGN_DIR)/design.xdc)
DESIGN_VERILOG = $(wildcard $(DESIGN_DIR)/design.v)
# DESIGN_VERILOG = $(wildcard $(DESIGN_DIR)/mult.v)

PNR_TOOL ?= nextpnr
BUILD    ?= build
WORK      = $(BUILD)/$(DESIGN_NAME)

IN_ENV = if [ -e env/bin/activate ]; then . env/bin/activate; fi;

# ============================================================================

all: $(WORK)/$(DESIGN_NAME).fasm parse

parse:
	$(PYTHON) parse_fasm.py

ifeq ($(PNR_TOOL),nextpnr)
arch: $(BUILD)/$(DEVICE)/chipdb.device
$(BUILD)/$(DEVICE)/chipdb.bba $(BUILD)/$(DEVICE)/chipdb.bin
endif
ifeq ($(PNR_TOOL),vpr)
arch: $(BUILD)/$(DEVICE)/chipdb.device
endif

clean:
	@rm -rf build

fullclean: clean
	@rm -rf env
	@rm -rf capnproto-c++-1.0.1

# ============================================================================

install: fullclean arch-defs

deps:
	@cd plugin/ && ./debian-deps.sh
	pip install matplotlib

fpga-interchange:
	@$(IN_ENV) pip install -e python-fpga-interchange
	@$(IN_ENV) $(PYTHON) --version; pip freeze

requirements:
	@$(IN_ENV) pip install -r requirements.txt
	# @$(IN_ENV) pip install -e python-fpga-interchange
	# @$(IN_ENV) $(PYTHON) --version; pip freeze
	$(MAKE) fpga-interchange

env:
	@$(PYTHON) -mvenv env
```

```
        $(MAKE) requirements

plugin:
        cd plugin/ && $(MAKE) clean all install

nextpnr:
        cd nextpnr && cmake . -DARCH=fpga_interchange
    -DTHREADS_HAVE_PTHREADS_ARG=TRUE; cd ..
        # cd nextpnr && cmake . -DARCH=fpga_interchange -DBUILD_GUI=ON; cd ..
        $(MAKE) -C nextpnr install
        $(MAKE) -C nextpnr
        install -D nextpnr/bba/bbasm /usr/local/bin/bbasm

capnproto:
        curl -O https://capnproto.org/capnproto-c++-1.0.1.tar.gz &&\
        tar zxf capnproto-c++-1.0.1.tar.gz &&\
        cd capnproto-c++-1.0.1 &&\
        ./configure &&\
        make -j6 check &&\
        make install

capnproto-java: capnproto
        $(MAKE) -C capnproto-java
        $(MAKE) -C capnproto-java install

arch-defs: deps capnproto capnproto-java nextpnr plugin
        $(MAKE) requirements

# ========================================================================

$(BUILD):
        @mkdir -p $@

$(BUILD)/$(DEVICE):
        @mkdir -p $@

$(BUILD)/$(DEVICE)/chipdb.device: $(ARCHBUILDER) | $(BUILD)/$(DEVICE)
        $(IN_ENV) CAPNP_PATH=$(CNP_PATH) $(PYTHON)
    $(ARCHBUILDER) --schema-dir $(SCHEMA) --out-file $@

ifeq ($(PNR_TOOL),nextpnr)

$(BUILD)/$(DEVICE)/chipdb.bba:
$(BUILD)/$(DEVICE)/chipdb.device $(DEVICE)_config.yaml | $(BUILD)/$(DEVICE)
        $(IN_ENV) CAPNP_PATH=$(CNP_PATH) $(PYTHON) -m fpga_interchange.nextpnr_emit
    --schema_dir $(SCHEMA)
    --output_dir $(BUILD)/$(DEVICE)
    --device_config $(DEVICE)_config.yaml --device $<

$(BUILD)/$(DEVICE)/chipdb.bin:
$(BUILD)/$(DEVICE)/chipdb.bba $(BBASM) | $(BUILD)/$(DEVICE)
        $(BBASM) -l --files $< $@

endif

# ========================================================================

$(WORK):
        @mkdir -p $@
```

```
$(WORK)/$(DESIGN_NAME).json: $(DESIGN_VERILOG) $(YOSYS) | $(WORK)
        $(YOSYS) -p "read_verilog␣-sv␣-I$(DESIGN_DIR)
␣␣␣␣$(DESIGN_VERILOG);␣plugin␣-i␣clhs-plugin;
␣␣␣␣synth_clhs␣-auto-top;␣clean␣-purge;␣show␣-width␣-signed
␣␣␣␣-format␣png␣-prefix␣rtlil;␣write_json␣$@;␣write_verilog
␣␣␣␣$@.v" -l $@.log

$(WORK)/$(DESIGN_NAME).netlist:
$(WORK)/$(DESIGN_NAME).json $(BUILD)/$(DEVICE)/chipdb.device | $(WORK)
        $(IN_ENV) CAPNP_PATH=$(CNP_PATH) $(PYTHON) -m
    fpga_interchange.yosys_json --schema_dir $(SCHEMA)
    --device $(BUILD)/$(DEVICE)/chipdb.device $< $@

ifeq ($(PNR_TOOL),nextpnr)

$(WORK)/$(DESIGN_NAME).phys: $(WORK)/$(DESIGN_NAME).netlist
$(DESIGN_XDC) $(BUILD)/$(DEVICE)/chipdb.bin $(NEXTPNR) |
$(WORK)
        $(NEXTPNR) -r --chipdb $(BUILD)/$(DEVICE)/chipdb.bin
    --netlist $< --xdc $(DESIGN_XDC) --phys $@ --log
    $@_nextpnr.log $(PNR_EXTRA_ARGS)

endif
ifeq ($(PNR_TOOL),vpr)

$(WORK)/$(DESIGN_NAME).phys: $(WORK)/$(DESIGN_NAME).netlist
$(BUILD)/$(DEVICE)/chipdb.device $(DESIGN_XDC) $(VPR) |
$(WORK)
        cd $(WORK) && $(VPR) $(abspath $(BUILD)/$(DEVICE)
    /chipdb.device) $(abspath $<) $(VPR_ARGS)
    $(PNR_EXTRA_ARGS) --xdc_files $(abspath $(DESIGN_XDC))
    --pack --place --route --analysis

endif

$(WORK)/$(DESIGN_NAME).phys.yaml: $(WORK)/$(DESIGN_NAME).phys | $(WORK)
        $(IN_ENV) $(PYTHON) -m fpga_interchange.convert
    --schema_dir $(SCHEMA) --schema physical --input_format
    capnp --output_format pyyaml $< $@

$(WORK)/$(DESIGN_NAME).fasm: $(BUILD)/$(DEVICE)/chipdb.device
$(WORK)/$(DESIGN_NAME).netlist $(WORK)/$(DESIGN_NAME).phys |
$(WORK)
        $(IN_ENV) $(PYTHON) $(FASMGEN) --schema_dir $(SCHEMA)
    $(BUILD)/$(DEVICE)/chipdb.device
    $(WORK)/$(DESIGN_NAME).netlist
    $(WORK)/$(DESIGN_NAME).phys $@

$(WORK)/$(DESIGN_NAME).txt: $(WORK)/$(DESIGN_NAME).fasm | $(WORK)
        $(IN_ENV) $(PYTHON) $(FASMASM) $< $@



# ============================================================================

.PHONY: fullclean clean all arch requirements nextpnr
capnproto arch-defs plugin env install fpga-interchange
```

A.2 generate_testarch (Route-through)

The following listing shows the *generate_testarch* Python script, which defines the *route-through* architecture that is used in the PnR toolchain (see section 3.2).

Code listing A.2: *generate_testarch* Python script for *route-through* architecture

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
#
# Copyright (C) 2020  The F4PGA Authors.
#
# Use of this source code is governed by a ISC-style
# license that can be found in the LICENSE file or at
# https://opensource.org/licenses/ISC
#
# SPDX-License-Identifier: ISC

import argparse
import math

from fpga_interchange.logical_netlist import Library, Cell, Direction, CellInstance
    , LogicalNetlist
from fpga_interchange.interchange_capnp import Interchange, CompressionFormat,
    write_capnp_file
from fpga_interchange.parameter_definitions import ParameterFormat

from fpga_interchange.testarch_generators.device_resources_builder import
    BelCategory, ConstantType
from fpga_interchange.testarch_generators.device_resources_builder import
    DeviceResources, DeviceResourcesCapnp

from fpga_interchange.testarch_generators.device_resources_builder import
    CellBelMapping, CellBelMappingEntry, Parameter, LutBel, PseudoCell

# ============================================================================


class TestArchGenerator():
    """
    Test architecture generator
    """

    def __init__(self, args):
        self.device = DeviceResources("testarch")

        self.grid_size = (8, 8)

        # Number of connections within tiles
        self.num_intra_nodes = 8
        # Number of connections between tiles
        self.num_inter_nodes = 3

        self.args = args

    def make_slice_site_type(self):
        """
        Generates a simple SLICE site type.
        """
```

```
        # The site
        site_type = self.device.add_site_type("SLICE")

        # Site pins (with BELs added automatically)
        site_type.add_pin("L0_0", Direction.Input,
                          (None, 2e-16, None, None, None, None))
        site_type.add_pin("L1_0", Direction.Input,
                          (None, 2e-16, None, None, None, None))
        site_type.add_pin("L2_0", Direction.Input,
                          (None, 2e-16, None, None, None, None))
        site_type.add_pin("L3_0", Direction.Input,
                          (None, 2e-16, None, None, None, None))
        site_type.add_pin("O_0", Direction.Output,
                          (None, 1.7, None, None, None, None))

        site_type.add_pin("D_0", Direction.Input,
                          (None, 2e-16, None, None, None, None))
        site_type.add_pin("Q_0", Direction.Output,
                          (None, 1.9, None, None, None, None))

        # Unique clock input
        site_type.add_pin("CLK", Direction.Input,
                          (None, 2e-16, None, None, None, None))

        # LUT4 BEL
        a_lut_bel = LutBel("ALUT", ["A1", 'A2', 'A3', 'A4'], 'O', 0, 15)
        site_type.add_lut_element(16, [a_lut_bel])
        bel_lut = site_type.add_bel("ALUT", "LUT4", BelCategory.LOGIC)
        bel_lut.add_pin("A1", Direction.Input)
        bel_lut.add_pin("A2", Direction.Input)
        bel_lut.add_pin("A3", Direction.Input)
        bel_lut.add_pin("A4", Direction.Input)
        bel_lut.add_pin("O", Direction.Output)

        # DFF BEL
        bel_ff = site_type.add_bel("AFF", "DFF", BelCategory.LOGIC)
        bel_ff.add_pin("C", Direction.Input)
        bel_ff.add_pin("D", Direction.Input)
        bel_ff.add_pin("Q", Direction.Output)

        if not self.args.no_ffmux:
            bel_mux = site_type.add_bel("AFFMUX", "MUX2", BelCategory.ROUTING)
            bel_mux.add_pin("I0", Direction.Input)
            bel_mux.add_pin("I1", Direction.Input)
            bel_mux.add_pin("O", Direction.Output)

        # LUT wires
        w = site_type.add_wire("L0_0_to_A1", [("L0_0", "L0_0"),
                                              ("ALUT", "A1")])
        w = site_type.add_wire("L1_0_to_A2", [("L1_0", "L1_0"),
                                              ("ALUT", "A2")])
        w = site_type.add_wire("L2_0_to_A3", [("L2_0", "L2_0"),
                                              ("ALUT", "A3")])
        w = site_type.add_wire("L3_0_to_A4", [("L3_0", "L3_0"),
                                              ("ALUT", "A4")])

        if not self.args.no_ffmux:
            w = site_type.add_wire("DIN_0", [("D_0", "D_0"), ("AFFMUX", "I1")])
```

```
        # Clock wire
        w = site_type.add_wire("CLK", [("CLK", "CLK"), ("AFF", "C")])

        w = site_type.add_wire("ALUT_O")
        w.connect_to_bel_pin("ALUT", "O")
        w.connect_to_bel_pin("O_0", "O_0")

        if not self.args.no_ffmux:
            w.connect_to_bel_pin("AFFMUX", "I0")
        else:
            w.connect_to_bel_pin("AFF", "D")

        if not self.args.no_ffmux:
            w = site_type.add_wire("AMUX_O")
            w.connect_to_bel_pin("AFFMUX", "O")
            w.connect_to_bel_pin("AFF", "D")

        w = site_type.add_wire("AFF_OUT", [("AFF", "Q"), ("Q_0", "Q_0")])

        # Site PIPs
        if not self.args.no_ffmux:
            site_type.add_pip(("AFFMUX", "I0"), ("AFFMUX", "O"),
                              (None, 5e-12, None, None, None, None))
            site_type.add_pip(("AFFMUX", "I1"), ("AFFMUX", "O"),
                              (None, 5e-12, None, None, None, None))

        for i in range(4):
            site_type.add_pip(("ALUT", f"A{i+1}"), ("ALUT", "O"),
                              (None, 5e-12, None, None, None, None))


    def make_iob_site_type(self):
        """
        Builds site types for input and output pads
        """

        # Input site type
        site_type = self.device.add_site_type("IPAD")

        bel = site_type.add_bel("IPAD", "PAD", BelCategory.LOGIC)
        bel.add_pin("P", Direction.Output)

        bel = site_type.add_bel("IB", "IB", BelCategory.LOGIC)
        bel.add_pin("P", Direction.Input)
        bel.add_pin("O", Direction.Output)

        site_type.add_pin("I", Direction.Output)

        site_type.add_wire("P_to_I", [("IB", "P"), ("IPAD", "P")])
        site_type.add_wire("P", [("I", "I"), ("IB", "O")])

        # Output site type
        site_type = self.device.add_site_type("OPAD")

        bel = site_type.add_bel("OPAD", "PAD", BelCategory.LOGIC)
        bel.add_pin("P", Direction.Input)

        bel = site_type.add_bel("OB", "OB", BelCategory.LOGIC)
        bel.add_pin("P", Direction.Output)
        bel.add_pin("I", Direction.Input)
```

```
        site_type.add_pin("O", Direction.Input)

        site_type.add_wire("P", [("OB", "P"), ("OPAD", "P")])
        site_type.add_wire("O_to_P", [("O", "O"), ("OB", "I")])

    def make_power_site_type(self):

        # The site
        site_type = self.device.add_site_type("POWER")

        # Site pins (with BELs added automatically)
        site_type.add_pin("V", Direction.Output)
        site_type.add_pin("G", Direction.Output)

        # VCC bel
        bel_vcc = site_type.add_bel("VCC", "VCC", BelCategory.LOGIC)
        bel_vcc.add_pin("V", Direction.Output)
        self.device.add_const_source(site_type.name, bel_vcc.name, 'V',
                                     ConstantType.VCC)

        # GND bel
        bel_gnd = site_type.add_bel("GND", "GND", BelCategory.LOGIC)
        bel_gnd.add_pin("G", Direction.Output)
        self.device.add_const_source(site_type.name, bel_gnd.name, 'G',
                                     ConstantType.GND)

        # Wires
        site_type.add_wire("V", [("VCC", "V"), ("V", "V")])
        site_type.add_wire("G", [("GND", "G"), ("G", "G")])

    def make_tile_type(self, tile_type_name, site_types):
        """
        Generates a simple CLB tile type
        """

        # The tile
        tile_type = self.device.add_tile_type(tile_type_name)

        # Sites and stuff
        for site_type_name in site_types:
            site_type = self.device.site_types[site_type_name]

            # Add the site
            site = tile_type.add_site(site_type.name)

            # Add tile wires for the site and site pin to tile wire mapping
            for pin in site_type.pins.values():

                if pin.direction == Direction.Input:
                    wire_name = "TO_{}_{}".format(site.ref, pin.name.upper())
                elif pin.direction == Direction.Output:
                    wire_name = "FROM_{}_{}".format(site.ref, pin.name.upper())
                else:
                    assert False

                tile_type.add_wire(wire_name, ("Tile-Site", "general"))
                site.primary_pins_to_tile_wires[pin.name] = wire_name

        if tile_type_name == "NULL":
```

```
            return

        # Add tile wires for intra nodes
        for i in range(self.num_intra_nodes):
            name = "INTRA_{}".format(i)
            tile_type.add_wire(name, ("Local", "general"))

        # Add tile wires for incoming and outgoin inter-tile connections
        for direction in ["N", "S", "E", "W"]:

            for i in range(self.num_inter_nodes):
                name = "OUT_{}_{}".format(direction, i)
                tile_type.add_wire(name, ("Interconnect", "general"))

            for i in range(self.num_inter_nodes):
                name = "INP_{}_{}".format(direction, i)
                tile_type.add_wire(name, ("Interconnect", "general"))

        # Add PIPs that connect tile wires for the site with intra wires
        wires_for_site = [w for w in tile_type.wires if w.startswith("TO_")]
        for dst_wire in wires_for_site:
            for i in range(self.num_intra_nodes):
                src_wire = "INTRA_{}".format(i)
                tile_type.add_pip(
                    src_wire, dst_wire, "intraTilePIP",
                    is_buffered21=False)

        wires_for_site = [w for w in tile_type.wires if w.startswith("FROM_")]
        for src_wire in wires_for_site:
            for i in range(self.num_intra_nodes):
                dst_wire = "INTRA_{}".format(i)
                tile_type.add_pip(
                    src_wire, dst_wire, "intraTilePIP",
                    is_buffered21=False)

        # Input tile wires to intra wires and vice-versa
        for direction in ["N", "S", "E", "W"]:
            for i in range(self.num_inter_nodes):

                src_wire = "INP_{}_{}".format(direction, i)
                for j in range(self.num_intra_nodes):
                    dst_wire = "INTRA_{}".format(j)
                    tile_type.add_pip(src_wire, dst_wire, "tilePIP")

                dst_wire = "OUT_{}_{}".format(direction, i)
                for j in range(self.num_intra_nodes):
                    src_wire = "INTRA_{}".format(j)
                    tile_type.add_pip(src_wire, dst_wire, "tilePIP")

        if tile_type_name == "PWR":
            tile_type.add_const_source(ConstantType.VCC, "FROM_POWER0_V")
            tile_type.add_const_source(ConstantType.GND, "FROM_POWER0_G")
        # TODO: const. wires


    def make_device_grid(self):
        width = self.grid_size[0] - 1
        height = self.grid_size[1] - 1

        for y in range(height + 1):
```

```
            for x in range(width + 1):
                is_0_0 = x == 0 and y == 0

                is_left = x == 0
                is_right = x == width

                is_centre = y == height // 2 and x == width // 2

                suffix = "_X{}Y{}".format(x, y)

                if is_0_0:
                    self.device.add_tile("NULL", "NULL", (x, y))
                elif is_left:
                    self.device.add_tile("IB" + suffix, "IB", (x, y))
                elif is_right:
                    self.device.add_tile("OB" + suffix, "OB", (x, y))
                elif is_centre:
                    self.device.add_tile("PWR" + suffix, "PWR", (x, y))
                else:
                    self.device.add_tile("CLB" + suffix, "CLB", (x, y))

    def make_wires_and_nodes(self):

        # Add wires for all tiles
        for tile_name in self.device.tiles_by_name:
            self.device.add_wires_for_tile(tile_name)

        # Add nodes for internal tile wires
        for tile in self.device.tiles.values():
            tile_type = self.device.tile_types[tile.type]

            for wire in tile_type.wires:
                if wire.startswith("TO_") or wire.startswith("FROM_"):
                    wire_id = self.device.get_wire_id(tile.name, wire)
                    self.device.add_node([wire_id], "toSite")
                elif wire.startswith("INTRA_"):
                    wire_id = self.device.get_wire_id(tile.name, wire)
                    self.device.add_node([wire_id], "internal")

        # Add nodes for inter-tile connections.
        def offset_loc(pos, ofs):
            return (pos[0] + ofs[0], pos[1] + ofs[1])

        for loc, tile_id in self.device.tiles_by_loc.items():
            if loc == (0, 0):
                continue
            tile = self.device.tiles[tile_id]
            tile_type = self.device.tile_types[tile.type]

            OPPOSITE = {
                "N": "S",
                "S": "N",
                "E": "W",
                "W": "E",
            }

            for direction, offset in [("N", (0, +1)), ("S", (0, -1)),
                                      ("E", (+1, 0)), ("W", (-1, 0))]:
                for i in range(self.num_inter_nodes):
                    wire_name = "INP_{}_{}".format(direction, i)
```

```
                wire_ids = [self.device.get_wire_id(tile.name, wire_name)]

                other_loc = offset_loc(loc, offset)
                if other_loc == (0, 0):
                    continue
                if other_loc[0] >= 0 and other_loc[0] < self.grid_size[0] and \
                   other_loc[1] >= 0 and other_loc[1] < self.grid_size[1]:

                    other_tile_id = self.device.tiles_by_loc[other_loc]
                    other_tile = self.device.tiles[other_tile_id]

                    if (tile_type.name.startswith("IB") and other_tile.name.
                        startswith("IB") \
                    or tile_type.name.startswith("OB") and other_tile.name.
                        startswith("OB")):
                        break
                    other_wire_name = "OUT_{}_{}".format(OPPOSITE[direction], i
                        )
                    wire_ids.append(self.device.get_wire_id(other_tile.name,
                        other_wire_name))

                self.device.add_node(wire_ids, "external")

    def make_package_data(self):

        package = self.device.add_package(self.args.package)

        ipad_id = 0
        opad_id = 0
        for site in self.device.sites.values():
            if site.type == "OPAD":
                pad_name = f"O_{opad_id}"
                opad_id += 1
            elif site.type == "IPAD":
                pad_name = f"I_{ipad_id}"
                ipad_id += 1
            else:
                continue

            package.add_pin(pad_name, site.name, site.type)

    def make_primitives_library(self):

        # Primitives library
        library = Library("primitives")
        self.device.cell_libraries["primitives"] = library

        def make_luts(max_size):
            for lut_size in range(1, max_size + 1):
                name = f"LUT{lut_size}"
                init = f"{2␣**␣lut_size}'h0"
                cell = Cell(name=name, property_map={"INIT": init})

                print(name, init)

                in_ports = list()
                for port in range(lut_size):
                    port_name = f"I{port}"
                    cell.add_port(port_name, Direction.Input)
                    in_ports.append(port_name)
```

```
                cell.add_port("O", Direction.Output)
                library.add_cell(cell)

                param = Parameter("INIT", ParameterFormat.VERILOG_HEX, init)
                self.device.add_parameter(name, param)
                self.device.add_lut_cell(name, in_ports, 'INIT')

        make_luts(4)

        def make_dffs(rst_types):
            for rst_type in rst_types:
                cell = Cell(f"DFF{rst_type}")
                cell.add_port("D", Direction.Input)
                cell.add_port("C", Direction.Input)
                cell.add_port("Q", Direction.Output)
                library.add_cell(cell)

        make_dffs([""])

        cell = Cell("IB")
        cell.add_port("O", Direction.Output)
        cell.add_port("P", Direction.Input)
        library.add_cell(cell)

        cell = Cell("OB")
        cell.add_port("I", Direction.Input)
        cell.add_port("P", Direction.Output)
        library.add_cell(cell)

        cell = Cell("VCC")
        cell.add_port("V", Direction.Output)
        library.add_cell(cell)

        cell = Cell("GND")
        cell.add_port("G", Direction.Output)
        library.add_cell(cell)

        # Macros library
        library = Library("macros")
        self.device.cell_libraries["macros"] = library

    def make_cell_bel_mappings(self):

        # TODO: Pass all the information via device.add_cell_bel_mapping()
        delay_mapping = [
            ('A1', 'O', (None, 50e-12, None, None, None, None), 'comb'),
            ('A2', 'O', (None, 50e-12, None, None, None, None), 'comb'),
            ('A3', 'O', (None, 50e-12, None, None, None, None), 'comb'),
            ('A4', 'O', (None, 50e-12, None, None, None, None), 'comb'),
        ]

        def make_lut_mapping(max_size):
            bel_pins = [f"A{pin}" for pin in range(1, max_size + 1)]
            cell_pins = [f"I{pin}" for pin in range(max_size)]

            for lut_size in range(1, max_size + 1):
                name = f"LUT{lut_size}"
                pin_map = dict(
                    zip(cell_pins[0:lut_size], bel_pins[0:lut_size]))
```

```
            pin_map["O"] = "O"

            mapping = CellBelMapping(name)
            mapping.entries.append(
                CellBelMappingEntry(
                    site_type="SLICE",
                    bels=["ALUT"],
                    pin_map=pin_map,
                    delay_mapping=delay_mapping[0:lut_size]))

            self.device.add_cell_bel_mapping(mapping)

        make_lut_mapping(4)

        delay_mapping = [
            ('D', ('C', 'rise'), (None, 5e-12, None, None, None, None),
             'setup'),
            ('D', ('C', 'rise'), (None, 8e-12, None, None, None, None),
             'hold'),
            (('C', 'rise'), 'Q', (None, 6e-12, None, None, None, None),
             'clk2q'),
        ]

        def make_dff_mapping(rst_types):
            for rst_type in rst_types:
                mapping = CellBelMapping(f"DFF{rst_type}")
                mapping.entries.append(
                    CellBelMappingEntry(
                        site_type="SLICE",
                        bels=["AFF"],
                        pin_map={
                            "D": "D",
                            "C": "C",
                            "Q": "Q",
                        },
                        delay_mapping=delay_mapping))
                self.device.add_cell_bel_mapping(mapping)

        make_dff_mapping([""])

        def make_iob_mapping(sites, bel, pin_map):
            mapping = CellBelMapping(bel)

            for site in sites:
                mapping.entries.append(
                    CellBelMappingEntry(
                        site_type=site, bels=[bel], pin_map=pin_map))

            self.device.add_cell_bel_mapping(mapping)

        make_iob_mapping(["IPAD"],
                         "IB",
                         pin_map={
                             "O": "O",
                             "P": "P"
                         })

        make_iob_mapping(["OPAD"],
                         "OB",
                         pin_map={
```

```
                            "I": "I",
                            "P": "P"
                        })

        mapping = CellBelMapping("GND")
        self.device.add_cell_bel_mapping(mapping)

        mapping = CellBelMapping("VCC")
        self.device.add_cell_bel_mapping(mapping)

    def make_parameters(self):
        pass

    def generate(self):
        self.make_iob_site_type()
        self.make_slice_site_type()
        self.make_power_site_type()

        self.make_tile_type("CLB", ["SLICE"])
        self.make_tile_type("IB", ["IPAD"])
        self.make_tile_type("OB", ["OPAD"])
        self.make_tile_type("PWR", ["POWER"])
        self.make_tile_type("NULL", [])

        self.make_device_grid()
        self.make_wires_and_nodes()

        self.make_package_data()

        self.make_primitives_library()
        self.make_cell_bel_mappings()
        self.make_parameters()

        # Add pip imings
        # Values are taken at random, resisitance, input and output capacitance are
             chosen
        # to be samewhat inline with values calculated from skaywater PDK
        self.device.add_PIPTiming("tilePIP", 3e-16, 1e-16, 5e-10, 0.5, 4e-16)
        self.device.add_PIPTiming("intraTilePIP", 1e-16, 4e-17, 3e-10, 0.1,
                                  2e-16)

        # Add node timing
        # Value taken from skywater PDK for metal layer 1,
        # Tile-to-Tile length 30 um, internal 15 um and to site 2 um
        # Wire width of 0.14 um
        self.device.add_nodeTiming("external", 26.8, 1.14e-14)
        self.device.add_nodeTiming("internal", 13.4, 5.7e-15)
        self.device.add_nodeTiming("toSite", 1.8, 7.6e-16)

        self.device.print_stats()


# ============================================================================


def main():

    parser = argparse.ArgumentParser(description="Generates␣testarch␣FPGA")
    parser.add_argument(
        "--schema-dir",
```

```
        required=True,
        help="Path␣to␣FPGA␣interchange␣capnp␣schema␣files")
    parser.add_argument(
        "--out-file", default="test_arch.device", help="Output␣file␣name")
    parser.add_argument("--package", default="TESTPKG", help="Package␣name")
    parser.add_argument(
        "--no-ffmux",
        action="store_true",
        help=
        "Do␣not␣add␣the␣mux␣that␣selects␣FF␣input␣forcing␣it␣to␣require␣LUT-thru"
    )

    args = parser.parse_args()

    # Run the test architecture generator
    gen = TestArchGenerator(args)
    gen.generate()

    # Initialize the writer (or "serializer")
    interchange = Interchange(args.schema_dir)
    writer = DeviceResourcesCapnp(
        gen.device,
        interchange.device_resources_schema,
        interchange.logical_netlist_schema,
    )

    # Serialize
    device_resources = writer.to_capnp()
    with open(args.out_file, "wb") as fp:
        write_capnp_file(
            device_resources,
            fp)  #, compression_format=CompressionFormat.UNCOMPRESSED)


# ============================================================================

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

A.3 generate_testarch (Bus)

The following listing shows the *generate_testarch* Python script, which defines the *bus* architecture that is used in the PnR toolchain (see section 3.2).

Code listing A.3: *generate_testarch* Python script for *bus* architecture

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
#
# Copyright (C) 2020  The F4PGA Authors.
#
# Use of this source code is governed by a ISC-style
# license that can be found in the LICENSE file or at
# https://opensource.org/licenses/ISC
#
# SPDX-License-Identifier: ISC

import argparse
import math

from fpga_interchange.logical_netlist import Library, Cell, Direction, CellInstance
    , LogicalNetlist
from fpga_interchange.interchange_capnp import Interchange, CompressionFormat,
    write_capnp_file
from fpga_interchange.parameter_definitions import ParameterFormat

from fpga_interchange.testarch_generators.device_resources_builder import
    BelCategory, ConstantType
from fpga_interchange.testarch_generators.device_resources_builder import
    DeviceResources, DeviceResourcesCapnp

from fpga_interchange.testarch_generators.device_resources_builder import
    CellBelMapping, CellBelMappingEntry, Parameter, LutBel, PseudoCell

# ============================================================================


class TestArchGenerator():
    """
    Test architecture generator
    """

    def __init__(self, args):
        self.device = DeviceResources("testarch")

        self.grid_size = (9, 9)
        self.width = self.grid_size[0] - 1
        self.height = self.grid_size[1] - 1

        # Number of connections within tiles
        self.num_intra_nodes = 6
        # Number of connections between tiles
        self.num_inter_nodes = 6

        self.bus_ports = self.height
        self.bus_width = 2

        self.args = args
```

```
    def make_slice_site_type(self):
        """
        Generates a simple SLICE site type.
        """

        # The site
        site_type = self.device.add_site_type("SLICE")

        # Site pins (with BELs added automatically)
        site_type.add_pin("L0_0", Direction.Input,
                          (None, 2e-16, None, None, None, None))
        site_type.add_pin("L1_0", Direction.Input,
                          (None, 2e-16, None, None, None, None))
        site_type.add_pin("L2_0", Direction.Input,
                          (None, 2e-16, None, None, None, None))
        site_type.add_pin("L3_0", Direction.Input,
                          (None, 2e-16, None, None, None, None))
        site_type.add_pin("O_0", Direction.Output,
                          (None, 1.7, None, None, None, None))

        site_type.add_pin("D_0", Direction.Input,
                          (None, 2e-16, None, None, None, None))
        site_type.add_pin("Q_0", Direction.Output,
                          (None, 1.9, None, None, None, None))

        # Unique clock input
        site_type.add_pin("CLK", Direction.Input,
                          (None, 2e-16, None, None, None, None))

        # LUT4 BEL
        a_lut_bel = LutBel("ALUT", ["A1", 'A2', 'A3', 'A4'], 'O', 0, 15)
        site_type.add_lut_element(16, [a_lut_bel])
        bel_lut = site_type.add_bel("ALUT", "LUT4", BelCategory.LOGIC)
        bel_lut.add_pin("A1", Direction.Input)
        bel_lut.add_pin("A2", Direction.Input)
        bel_lut.add_pin("A3", Direction.Input)
        bel_lut.add_pin("A4", Direction.Input)
        bel_lut.add_pin("O", Direction.Output)

        # DFF BEL
        bel_ff = site_type.add_bel("AFF", "DFF", BelCategory.LOGIC)
        bel_ff.add_pin("C", Direction.Input)
        bel_ff.add_pin("D", Direction.Input)
        bel_ff.add_pin("Q", Direction.Output)

        if not self.args.no_ffmux:
            bel_mux = site_type.add_bel("AFFMUX", "MUX2", BelCategory.ROUTING)
            bel_mux.add_pin("I0", Direction.Input)
            bel_mux.add_pin("I1", Direction.Input)
            bel_mux.add_pin("O", Direction.Output)

        # LUT wires
        w = site_type.add_wire("L0_0_to_A1", [("L0_0", "L0_0"),
                                              ("ALUT", "A1")])
        w = site_type.add_wire("L1_0_to_A2", [("L1_0", "L1_0"),
                                              ("ALUT", "A2")])
        w = site_type.add_wire("L2_0_to_A3", [("L2_0", "L2_0"),
                                              ("ALUT", "A3")])
        w = site_type.add_wire("L3_0_to_A4", [("L3_0", "L3_0"),
```

```
                                            ("ALUT", "A4")])

        if not self.args.no_ffmux:
            w = site_type.add_wire("DIN_0", [("D_0", "D_0"), ("AFFMUX", "I1")])

        # Clock wire
        w = site_type.add_wire("CLK", [("CLK", "CLK"), ("AFF", "C")])

        w = site_type.add_wire("ALUT_O")
        w.connect_to_bel_pin("ALUT", "O")
        w.connect_to_bel_pin("O_0", "O_0")

        if not self.args.no_ffmux:
            w.connect_to_bel_pin("AFFMUX", "I0")
        else:
            w.connect_to_bel_pin("AFF", "D")

        if not self.args.no_ffmux:
            w = site_type.add_wire("AMUX_O")
            w.connect_to_bel_pin("AFFMUX", "O")
            w.connect_to_bel_pin("AFF", "D")

        w = site_type.add_wire("AFF_OUT", [("AFF", "Q"), ("Q_0", "Q_0")])

        # Site PIPs
        if not self.args.no_ffmux:
            site_type.add_pip(("AFFMUX", "I0"), ("AFFMUX", "O"),
                              (None, 5e-12, None, None, None, None))
            site_type.add_pip(("AFFMUX", "I1"), ("AFFMUX", "O"),
                              (None, 5e-12, None, None, None, None))

        for i in range(4):
            site_type.add_pip(("ALUT", f"A{i+1}"), ("ALUT", "O"),
                              (None, 5e-12, None, None, None, None))


    def make_iob_site_type(self):
        """
        Builds site types for input and output pads
        """

        # Input site type
        site_type = self.device.add_site_type("IPAD")

        bel = site_type.add_bel("IPAD", "PAD", BelCategory.LOGIC)
        bel.add_pin("P", Direction.Output)

        bel = site_type.add_bel("IB", "IB", BelCategory.LOGIC)
        bel.add_pin("P", Direction.Input)
        bel.add_pin("O", Direction.Output)

        site_type.add_pin("I", Direction.Output)

        site_type.add_wire("P_to_I", [("IB", "P"), ("IPAD", "P")])
        site_type.add_wire("P", [("I", "I"), ("IB", "O")])

        # Output site type
        site_type = self.device.add_site_type("OPAD")

        bel = site_type.add_bel("OPAD", "PAD", BelCategory.LOGIC)
```

```
        bel.add_pin("P", Direction.Input)

        bel = site_type.add_bel("OB", "OB", BelCategory.LOGIC)
        bel.add_pin("P", Direction.Output)
        bel.add_pin("I", Direction.Input)

        site_type.add_pin("O", Direction.Input)

        site_type.add_wire("P", [("OB", "P"), ("OPAD", "P")])
        site_type.add_wire("O_to_P", [("O", "O"), ("OB", "I")])

    def make_power_site_type(self):

        # The site
        site_type = self.device.add_site_type("POWER")

        # Site pins (with BELs added automatically)
        site_type.add_pin("V", Direction.Output)
        site_type.add_pin("G", Direction.Output)

        # VCC bel
        bel_vcc = site_type.add_bel("VCC", "VCC", BelCategory.LOGIC)
        bel_vcc.add_pin("V", Direction.Output)
        self.device.add_const_source(site_type.name, bel_vcc.name, 'V',
                                     ConstantType.VCC)

        # GND bel
        bel_gnd = site_type.add_bel("GND", "GND", BelCategory.LOGIC)
        bel_gnd.add_pin("G", Direction.Output)
        self.device.add_const_source(site_type.name, bel_gnd.name, 'G',
                                     ConstantType.GND)

        # Wires
        site_type.add_wire("V", [("VCC", "V"), ("V", "V")])
        site_type.add_wire("G", [("GND", "G"), ("G", "G")])

    def make_bus_type(self):
        # Input site type
        site_type = self.device.add_site_type("BUS")

        for n in range(self.bus_ports):
            for w in range(self.bus_width):
                site_type.add_pin(f"BUS_SITE_IN_{n}_{w}", Direction.Input)
                site_type.add_pin(f"BUS_SITE_OUT_{n}_{w}", Direction.Output)
                site_type.add_wire(f"BUS_{n}_{w}", [(f"BUS_SITE_IN_{n}_{w}", f"
                    BUS_SITE_IN_{n}_{w}"), (f"BUS_SITE_OUT_{n}_{w}", f"
                    BUS_SITE_OUT_{n}_{w}")])

    def make_tile_type(self, tile_type_name, site_types):
        """
        Generates a simple CLB tile type
        """

        # The tile
        tile_type = self.device.add_tile_type(tile_type_name)

        # Sites and stuff
        for site_type_name in site_types:
            site_type = self.device.site_types[site_type_name]
```

```
            # Add the site
            site = tile_type.add_site(site_type.name)

            # Add tile wires for the site and site pin to tile wire mapping
            for pin in site_type.pins.values():

                if pin.direction == Direction.Input:
                    wire_name = "TO_{}_{}".format(site.ref, pin.name.upper())
                elif pin.direction == Direction.Output:
                    wire_name = "FROM_{}_{}".format(site.ref, pin.name.upper())
                else:
                    assert False

                tile_type.add_wire(wire_name, ("Tile-Site", "general"))
                site.primary_pins_to_tile_wires[pin.name] = wire_name

        if tile_type_name == "NULL":
            return

        if tile_type_name == "BUS":

            # Add BUS interconnect wires
            for w in range(self.bus_width):
                for n in range(self.bus_ports):
                    name = f"BUS_IN_{n}_{w}"
                    tile_type.add_wire(name, ("Interconnect", "general"))

                name = f"BUS_OUT_{w}"
                tile_type.add_wire(name, ("Interconnect", "general"))

            # PIPs for BUS_OUT_n and BUS_SITE_OUT_n_w
            for w in range(self.bus_width):
                dst_wire = f"BUS_OUT_{w}"

                for n in range(self.bus_ports):
                    src_wire = f"FROM_BUS0_BUS_SITE_OUT_{n}_{w}"
                    tile_type.add_pip(
                        src_wire, dst_wire, "intraTilePIP",
                        is_buffered21=False)
            return

        # Add tile wires for intra nodes
        for i in range(self.num_intra_nodes):
            name = "INTRA_{}".format(i)
            tile_type.add_wire(name, ("Local", "general"))

        if tile_type_name == "CLB":
            # Add BUS interconnect wires
            for w in range(self.bus_width):
                name = f"BUS_IN_{w}"
                tile_type.add_wire(name, ("Interconnect", "general"))
                name = f"BUS_OUT_{w}"
                tile_type.add_wire(name, ("Interconnect", "general"))

            # PIPs for BUS and INTRA wires
            wires_for_site = [w for w in tile_type.wires if w.startswith("BUS_IN")]
            for dst_wire in wires_for_site:
                for i in range(self.num_intra_nodes):
                    src_wire = "INTRA_{}".format(i)
                    tile_type.add_pip(src_wire, dst_wire, "tilePIP")
```

```
            wires_for_site = [w for w in tile_type.wires if w.startswith("BUS_OUT")
                ]
            for src_wire in wires_for_site:
                for i in range(self.num_intra_nodes):
                    dst_wire = "INTRA_{}".format(i)
                    tile_type.add_pip(src_wire, dst_wire, "tilePIP")

        # Add tile wires for incoming and outgoin inter-tile connections
        for direction in ["N", "S", "E", "W"]:
            for i in range(self.num_inter_nodes):
                name = "OUT_{}_{}".format(direction, i)
                tile_type.add_wire(name, ("Interconnect", "general"))

            for i in range(self.num_inter_nodes):
                name = "INP_{}_{}".format(direction, i)
                tile_type.add_wire(name, ("Interconnect", "general"))

        # Add PIPs that connect tile wires for the site with intra wires
        wires_for_site = [w for w in tile_type.wires if w.startswith("TO_")]
        for dst_wire in wires_for_site:
            for i in range(self.num_intra_nodes):
                src_wire = "INTRA_{}".format(i)
                tile_type.add_pip(
                    src_wire, dst_wire, "intraTilePIP",
                    is_buffered21=False)

        wires_for_site = [w for w in tile_type.wires if w.startswith("FROM_")]
        for src_wire in wires_for_site:
            dst_wire = "OUT_{}".format(src_wire)
            tile_type.add_wire(dst_wire, ("Local", "general"))
            tile_type.add_pip(
                src_wire, dst_wire, "intraTilePIP",
                is_buffered21=False)

        # Input tile wires to intra wires and vice-versa
        for direction in ["N", "S", "E", "W"]:

            # Route-through
            for i in range(self.num_inter_nodes):
                src_wire = "INP_{}_{}".format(direction, i)
                tile_type.add_pip(src_wire, "RT", "tilePIP")

                dst_wire = "OUT_{}_{}".format(direction, i)
                tile_type.add_pip("RT", dst_wire, "tilePIP")

            for i in range(self.num_inter_nodes):
                src_wire = "INP_{}_{}".format(direction, i)
                for j in range(self.num_intra_nodes):
                    dst_wire = "INTRA_{}".format(j)
                    tile_type.add_pip(src_wire, dst_wire, "tilePIP")

                wires_for_site = [w for w in tile_type.wires if w.startswith("
                    OUT_FROM_")]
                for src_wire in wires_for_site:
                    dst_wire = "OUT_{}_{}".format(direction, i)
                    tile_type.add_pip(src_wire, dst_wire, "tilePIP")

        if tile_type_name == "PWR":
            tile_type.add_const_source(ConstantType.VCC, "FROM_POWER0_V")
```

```
            tile_type.add_const_source(ConstantType.GND, "FROM_POWER0_G")

    def print_grid(self):
        for loc, tile_id in self.device.tiles_by_loc.items():
            tile = self.device.tiles[tile_id]

            print(str(loc) + "␣" + tile.name, end="␣")
            if (loc[0] == 8):
                print("")

    def make_device_grid(self):

        for y in range(self.height + 1):
            for x in range(self.width + 1):
                is_vert = y == self.height
                is_horiz = x == self.width
                is_0_0 = x == 0 and y == 0
                is_left = x == 0
                is_right = x == self.width - 1
                is_centre = y == (self.height - 1) // 2 and x == (self.width - 1)
                    // 2
                # is_bus = (x, y) in [(2, 8), (5, 8), (7, 8)]
                # is_bus = (x, y) in [(2, 8), (8, 6)]
                is_bus = (x, y) in [(2, 8)]

                suffix = "_X{}Y{}".format(x, y)

                if is_0_0:
                    self.device.add_tile("NULL", "NULL", (x, y))
                elif is_left and not is_vert:
                    self.device.add_tile("IB" + suffix, "IB", (x, y))
                elif is_right and not is_vert:
                    self.device.add_tile("OB" + suffix, "OB", (x, y))
                elif is_centre:
                    self.device.add_tile("PWR" + suffix, "PWR", (x, y))
                # elif is_bus:
                #     self.device.add_tile("BUS" + suffix, "BUS", (x, y))
                elif not (is_vert or is_horiz):
                    self.device.add_tile("CLB" + suffix, "CLB", (x, y))
                else:
                    self.device.add_tile("PWR" + suffix, "PWR", (x, y))

        self.print_grid()

    def make_route_through(self):
        # Add nodes for inter-tile connections.
        def offset_loc(pos, ofs):
            return (pos[0] + ofs[0], pos[1] + ofs[1])

        for loc, tile_id in self.device.tiles_by_loc.items():
            if loc == (0, 0):
                continue
            tile = self.device.tiles[tile_id]
            tile_type = self.device.tile_types[tile.type]
            if tile_type.name.startswith("BUS"):
                break

            OPPOSITE = {
                "N": "S",
                "S": "N",
```

```
                "E": "W",
                "W": "E",
            }

            for direction, offset in [("N", (0, +1)), ("S", (0, -1)),
                                      ("E", (+1, 0)), ("W", (-1, 0))]:
                for i in range(self.num_inter_nodes):

                    wire_name = "INP_{}_{}".format(direction, i)
                    wire_ids = [self.device.get_wire_id(tile.name, wire_name)]

                    other_loc = offset_loc(loc, offset)
                    if other_loc == (0, 0):
                        continue
                    if other_loc[0] >= 0 and other_loc[0] < self.width and \
                        other_loc[1] >= 0 and other_loc[1] < self.height:

                        other_tile_id = self.device.tiles_by_loc[other_loc]
                        other_tile = self.device.tiles[other_tile_id]
                        if other_tile.name.startswith("BUS"):
                            break
                        other_wire_name = "OUT_{}_{}".format(OPPOSITE[direction], i
                            )
                        wire_ids.append(self.device.get_wire_id(other_tile.name,
                            other_wire_name))

                        self.device.add_node(wire_ids, "external")

    def make_bus_vert(self, loc, tile_id):
        tile = self.device.tiles[tile_id]

        # BUS_IN
        # Seperate node per bus channel
        for w in range(self.bus_width):
            for n in range(0, loc[1]): # Move up left column

                bus_in = f"BUS_IN_{n}_{w}"
                bus_inputs = [self.device.get_wire_id(tile.name, bus_in)]

                other_loc = (loc[0] - 1, n) # Iterate through left column pos
                if other_loc == (0, 0):
                    continue
                if other_loc[0] >= 0 and other_loc[0] < self.width and \
                    other_loc[1] >= 0 and other_loc[1] < self.height:

                    other_tile_id = self.device.tiles_by_loc[other_loc]
                    other_tile = self.device.tiles[other_tile_id]
                    other_bus_in = f"BUS_IN_{w}"

                    # We done goofed up
                    assert(self.device.tile_types[other_tile.type].name.startswith(
                        "CLB"))

                    bus_inputs.append(self.device.get_wire_id(other_tile.name,
                        other_bus_in))

                self.device.add_node(bus_inputs, "external")

        # BUS OUT
        # Connect bus output to all tiles with same node
```

```
        for w in range(self.bus_width):
            bus_output = f"BUS_OUT_{w}"
            bus_outputs = [self.device.get_wire_id(tile.name, bus_output)]

            for n in range(0, loc[1]): # Move up left column

                other_loc = (loc[0] - 1, n)
                if other_loc == (0, 0):
                    continue
                if other_loc[0] >= 0 and other_loc[0] < self.width and \
                    other_loc[1] >= 0 and other_loc[1] < self.height:

                    other_tile_id = self.device.tiles_by_loc[other_loc]
                    other_tile = self.device.tiles[other_tile_id]

                    bus_outputs.append(self.device.get_wire_id(other_tile.name,
                        bus_output))

            self.device.add_node(bus_outputs, "external")

    def make_bus_horiz(self, loc, tile_id):
        tile = self.device.tiles[tile_id]
        tile_type = self.device.tile_types[tile.type]

        if tile_type.name.startswith("BUS"):

            # BUS_IN
            # Seperate node per bus channel
            for w in range(self.bus_width):
                for n in range(0, loc[0]): # Move through above row

                    bus_in = f"BUS_IN_{n}_{w}"
                    bus_inputs = [self.device.get_wire_id(tile.name, bus_in)]

                    other_loc = (n, loc[1] + 1) # Moving through top yes very good
                    if other_loc == (0, 0):
                        continue
                    if other_loc[0] >= 1 and other_loc[0] < self.width - 1 and \
                        other_loc[1] >= 0 and other_loc[1] < self.height:

                        other_tile_id = self.device.tiles_by_loc[other_loc]
                        other_tile = self.device.tiles[other_tile_id]
                        other_bus_in = f"BUS_IN_{w}"

                        # We done goofed up
                        assert(self.device.tile_types[other_tile.type].name.
                            startswith("CLB"))

                        bus_inputs.append(self.device.get_wire_id(other_tile.name,
                            other_bus_in))

                    self.device.add_node(bus_inputs, "external")

            # BUS OUT
            # Connect bus output to all tiles with same node
            for w in range(self.bus_width):
                bus_output = f"BUS_OUT_{w}"
                bus_outputs = [self.device.get_wire_id(tile.name, bus_output)]

                for n in range(0, loc[1]): # Move up left column
```

```
                other_loc = (n, loc[1] + 1) # Moving through top yes very good
                if other_loc == (0, 0):
                    continue
                if other_loc[0] >= 1 and other_loc[0] < self.width - 1 and \
                    other_loc[1] >= 0 and other_loc[1] < self.height:

                    other_tile_id = self.device.tiles_by_loc[other_loc]
                    other_tile = self.device.tiles[other_tile_id]

                    bus_outputs.append(self.device.get_wire_id(other_tile.name,
                         bus_output))

            self.device.add_node(bus_outputs, "external")

    def make_wires_and_nodes(self):

        # Add wires for all tiles
        for tile_name in self.device.tiles_by_name:
            self.device.add_wires_for_tile(tile_name)

        # Add nodes for internal tile wires
        for tile in self.device.tiles.values():
            tile_type = self.device.tile_types[tile.type]

            for wire in tile_type.wires:
                if wire.startswith("TO_") or wire.startswith("FROM_"):
                    wire_id = self.device.get_wire_id(tile.name, wire)
                    self.device.add_node([wire_id], "toSite")
                elif wire.startswith("INTRA_"):
                    wire_id = self.device.get_wire_id(tile.name, wire)
                    self.device.add_node([wire_id], "internal")

        self.make_route_through()

        for loc, tile_id in self.device.tiles_by_loc.items():
            tile = self.device.tiles[tile_id]
            tile_type = self.device.tile_types[tile.type]

            if tile_type.name.startswith("BUS"):
                if (loc[1] == self.height) and (loc[0] < self.width):
                    self.make_bus_vert(loc, tile_id)
                elif (loc[1] <  self.height) and (loc[0] == self.width):
                    self.make_bus_horiz(loc, tile_id)

    def make_package_data(self):

        package = self.device.add_package(self.args.package)

        ipad_id = 0
        opad_id = 0
        for site in self.device.sites.values():
            if site.type == "OPAD":
                pad_name = f"O_{opad_id}"
                opad_id += 1
            elif site.type == "IPAD":
                pad_name = f"I_{ipad_id}"
                ipad_id += 1
            else:
                continue
```

```
            package.add_pin(pad_name, site.name, site.type)

    def make_primitives_library(self):

        # Primitives library
        library = Library("primitives")
        self.device.cell_libraries["primitives"] = library

        def make_luts(max_size):
            for lut_size in range(1, max_size + 1):
                name = f"LUT{lut_size}"
                init = f"{2␣**␣lut_size}'h0"
                cell = Cell(name=name, property_map={"INIT": init})

                print(name, init)

                in_ports = list()
                for port in range(lut_size):
                    port_name = f"I{port}"
                    cell.add_port(port_name, Direction.Input)
                    in_ports.append(port_name)

                cell.add_port("O", Direction.Output)
                library.add_cell(cell)

                param = Parameter("INIT", ParameterFormat.VERILOG_HEX, init)
                self.device.add_parameter(name, param)
                self.device.add_lut_cell(name, in_ports, 'INIT')

        make_luts(4)

        def make_dffs(rst_types):
            for rst_type in rst_types:
                cell = Cell(f"DFF{rst_type}")
                cell.add_port("D", Direction.Input)
                cell.add_port("C", Direction.Input)
                cell.add_port("Q", Direction.Output)
                library.add_cell(cell)

        make_dffs([""])

        cell = Cell("IB")
        cell.add_port("O", Direction.Output)
        cell.add_port("P", Direction.Input)
        library.add_cell(cell)

        cell = Cell("OB")
        cell.add_port("I", Direction.Input)
        cell.add_port("P", Direction.Output)
        library.add_cell(cell)

        cell = Cell("VCC")
        cell.add_port("V", Direction.Output)
        library.add_cell(cell)

        cell = Cell("GND")
        cell.add_port("G", Direction.Output)
        library.add_cell(cell)
```

```
        # Macros library
        library = Library("macros")
        self.device.cell_libraries["macros"] = library

    def make_cell_bel_mappings(self):

        # TODO: Pass all the information via device.add_cell_bel_mapping()
        delay_mapping = [
            ('A1', 'O', (None, 50e-12, None, None, None, None), 'comb'),
            ('A2', 'O', (None, 50e-12, None, None, None, None), 'comb'),
            ('A3', 'O', (None, 50e-12, None, None, None, None), 'comb'),
            ('A4', 'O', (None, 50e-12, None, None, None, None), 'comb'),
        ]

        def make_lut_mapping(max_size):
            bel_pins = [f"A{pin}" for pin in range(1, max_size + 1)]
            cell_pins = [f"I{pin}" for pin in range(max_size)]

            for lut_size in range(1, max_size + 1):
                name = f"LUT{lut_size}"
                pin_map = dict(
                    zip(cell_pins[0:lut_size], bel_pins[0:lut_size]))
                pin_map["O"] = "O"

                mapping = CellBelMapping(name)
                mapping.entries.append(
                    CellBelMappingEntry(
                        site_type="SLICE",
                        bels=["ALUT"],
                        pin_map=pin_map,
                        delay_mapping=delay_mapping[0:lut_size]))

                self.device.add_cell_bel_mapping(mapping)

        make_lut_mapping(4)

        delay_mapping = [
            ('D', ('C', 'rise'), (None, 5e-12, None, None, None, None),
             'setup'),
            ('D', ('C', 'rise'), (None, 8e-12, None, None, None, None),
             'hold'),
            (('C', 'rise'), 'Q', (None, 6e-12, None, None, None, None),
             'clk2q'),
        ]

        def make_dff_mapping(rst_types):
            for rst_type in rst_types:
                mapping = CellBelMapping(f"DFF{rst_type}")
                mapping.entries.append(
                    CellBelMappingEntry(
                        site_type="SLICE",
                        bels=["AFF"],
                        pin_map={
                            "D": "D",
                            "C": "C",
                            "Q": "Q",
                        },
                        delay_mapping=delay_mapping))
                self.device.add_cell_bel_mapping(mapping)
```

```
        make_dff_mapping([""])

        def make_iob_mapping(sites, bel, pin_map):
            mapping = CellBelMapping(bel)

            for site in sites:
                mapping.entries.append(
                    CellBelMappingEntry(
                        site_type=site, bels=[bel], pin_map=pin_map))

            self.device.add_cell_bel_mapping(mapping)

        make_iob_mapping(["IPAD"],
                         "IB",
                         pin_map={
                             "O": "O",
                             "P": "P"
                         })

        make_iob_mapping(["OPAD"],
                         "OB",
                         pin_map={
                             "I": "I",
                             "P": "P"
                         })

        mapping = CellBelMapping("GND")
        self.device.add_cell_bel_mapping(mapping)

        mapping = CellBelMapping("VCC")
        self.device.add_cell_bel_mapping(mapping)

    def make_parameters(self):
        pass

    def generate(self):
        self.make_iob_site_type()
        self.make_slice_site_type()
        self.make_power_site_type()
        self.make_bus_type()

        self.make_tile_type("CLB", ["SLICE"])
        self.make_tile_type("BUS", ["BUS"])
        self.make_tile_type("IB", ["IPAD"])
        self.make_tile_type("OB", ["OPAD"])
        self.make_tile_type("PWR", ["POWER"])
        self.make_tile_type("NULL", [])

        self.make_device_grid()
        self.make_wires_and_nodes()

        self.make_package_data()

        self.make_primitives_library()
        self.make_cell_bel_mappings()
        self.make_parameters()

        # Add pip imings
        # Values are taken at random, resisitance, input and output capacitance are
              chosen
```

```
        # to be samewhat inline with values calculated from skaywater PDK
        self.device.add_PIPTiming("tilePIP", 3e-16, 1e-16, 5e-10, 0.5, 4e-16)
        self.device.add_PIPTiming("intraTilePIP", 1e-16, 4e-17, 3e-10, 0.1,
                                  2e-16)

        # Add node timing
        # Value taken from skywater PDK for metal layer 1,
        # Tile-to-Tile length 30 um, internal 15 um and to site 2 um
        # Wire width of 0.14 um
        self.device.add_nodeTiming("external", 26.8, 1.14e-14)
        self.device.add_nodeTiming("internal", 13.4, 5.7e-15)
        self.device.add_nodeTiming("toSite", 1.8, 7.6e-16)

        self.device.print_stats()


# ============================================================================


def main():

    parser = argparse.ArgumentParser(description="Generates␣testarch␣FPGA")
    parser.add_argument(
        "--schema-dir",
        required=True,
        help="Path␣to␣FPGA␣interchange␣capnp␣schema␣files")
    parser.add_argument(
        "--out-file", default="test_arch.device", help="Output␣file␣name")
    parser.add_argument("--package", default="TESTPKG", help="Package␣name")
    parser.add_argument(
        "--no-ffmux",
        action="store_true",
        help=
        "Do␣not␣add␣the␣mux␣that␣selects␣FF␣input␣forcing␣it␣to␣require␣LUT-thru"
    )

    args = parser.parse_args()

    # Run the test architecture generator
    gen = TestArchGenerator(args)
    gen.generate()

    # Initialize the writer (or "serializer")
    interchange = Interchange(args.schema_dir)
    writer = DeviceResourcesCapnp(
        gen.device,
        interchange.device_resources_schema,
        interchange.logical_netlist_schema,
    )

    # Serialize
    device_resources = writer.to_capnp()
    with open(args.out_file, "wb") as fp:
        write_capnp_file(
            device_resources,
            fp)  #, compression_format=CompressionFormat.UNCOMPRESSED)


# ============================================================================
```

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

A.4 parse_fasm

The following listing shows the *parse_fasm* Python script, which is used to parse and illustrate the FASM data generated by the FOSS toolchain (see section 3.2).

Code listing A.4: *parse_fasm* Python script

```
import re
import matplotlib.pyplot as plt
import matplotlib.patches as patches

def parse_fasm_file(file_path):
    with open(file_path, "r") as f:
        fasm_content = f.read()

    pattern = r"([A-Za-z0-9_]+)\.([A-Za-z0-9_]+)\.([A-Za-z0-9_]+)"
    matches = re.findall(pattern, fasm_content)

    fasm_connections = []
    for match in matches:
        module_name, pin_name, net_name = match
        connection = {
            "module": module_name,
            "pin": pin_name,
            "net": net_name
        }
        fasm_connections.append(connection)

    fasm_connections = fasm_connections[1:]

    return fasm_connections


fasm_connections = parse_fasm_file("build/design/design.fasm")

# Num. total placed blocks
# % logic blocks
# % route-throughs (except IO blocks)

def calc_util(fasm_connections):
    block_counts = {}  # Dictionary to store total lines per block
    num_logic_blocks = 0
    logic_match = False

    for connection in fasm_connections:
        module_name = connection["module"]
        net_name = connection["net"]
        pin_name = connection["pin"]
        # if net_name.startswith("INTRA"):
        block_match = re.match(r'[A-Z]+_X(\d+)Y(\d+)', module_name)
        # I have seen this block before...
        if block_match and not (module_name.startswith("IB") or module_name.
            startswith("OB")):
```

```
            block_number = (int(block_match.group(1)), int(block_match.group(2)))
            if block_number not in block_counts:
                block_counts[block_number] = 1
                logic_match = False

            # Logic v route-throughs. IOBs are an exception
            # if (net_name.startswith("TO_SLICE") or net_name.startswith("
                FROM_SLICE") \
            # or pin_name.startswith("TO_SLICE") or pin_name.startswith("FROM_SLICE
                ")) \
            # and (("IB" not in module_name) or ("OB" not in module_name)) \
            # and not logic_match:
            if (net_name.startswith("ALUT") and not logic_match):
                num_logic_blocks += 1
                logic_match = True

    # The rest should be route-throughs
    total_blocks = len(block_counts)
    num_route_throughs = total_blocks - num_logic_blocks

    # breakpoint()
    print("Total num of blocks: ", total_blocks)
    print("Num of logic: ", num_logic_blocks)
    print("Num of route-throughs: ", num_route_throughs)
    print("% logic: ", (num_logic_blocks/total_blocks)*100, "\t % route-throughs: "
        , (num_route_throughs/total_blocks)*100)

def plot():
# Extract module names from fasm_connections
    module_names = list(set(connection["module"] for connection in fasm_connections
        ))

# Extract X and Y coordinates from module names using regular expressions
    x_coordinates = [int(re.search(r'X(\d+)', name).group(1)) for name in
        module_names]
    y_coordinates = [int(re.search(r'Y(\d+)', name).group(1)) for name in
        module_names]

# Create a grid based on X-Y coordinates
    plt.figure(figsize=(8, 6))
    plt.scatter(x_coordinates, y_coordinates, color='b', marker='s', s=100)
    plt.grid(True, linestyle='--', alpha=0.7)

# Label modules with their names
    for i, name in enumerate(module_names):
        plt.text(x_coordinates[i] + 0.1, y_coordinates[i] + 0.1, name, fontsize=9)

# Add connections as arrows between modules based on fasm_connections
    for connection in fasm_connections:
        src_module = connection["module"]
        pin_or_net_1 = connection["pin"]
        pin_or_net_2 = connection["net"]
        if src_module in module_names:
            src_idx = module_names.index(src_module)
            src_x, src_y = x_coordinates[src_idx], y_coordinates[src_idx]

            # Determine the wire name from pin_or_net_1 or pin_or_net_2 (whichever
                starts with "INTRA" or has a direction)
            if pin_or_net_1.startswith("INP") or pin_or_net_1.startswith("OUT"):
                wire_name = pin_or_net_1
```

```
            if pin_or_net_2.startswith("INP") or pin_or_net_2.startswith("OUT"):
                wire_name = pin_or_net_2
            else:
                # If neither pin nor net starts with "INTRA", treat them as wires
                wire_name = pin_or_net_1  # You can also use pin_or_net_2, as they
                    are interchangeable

            if "OUT_N" in wire_name:
                dest_x, dest_y = src_x, src_y + 1
            elif "OUT_S" in wire_name:
                dest_x, dest_y = src_x, src_y - 1
            elif "OUT_E" in wire_name:
                dest_x, dest_y = src_x + 1, src_y
            elif "OUT_W" in wire_name:
                dest_x, dest_y = src_x - 1, src_y
            else:
                continue
            if (dest_x, dest_y) == (0, 0): # Ignore routing to NULL tiles
                continue

            arrow = patches.FancyArrowPatch((src_x, src_y), (dest_x, dest_y), color
                ='red', arrowstyle='->', mutation_scale=20)
            plt.gca().add_patch(arrow)
            # - MUX based bus

# Set plot limits and labels
    plt.xlim(-1, max(x_coordinates) + 1)
    plt.ylim(-1., max(y_coordinates) + 1)
    plt.xlabel('X Coordinate')
    plt.ylabel('Y Coordinate')
    plt.title("CPLD device grid")

# Display the grid
# plt.show()
    plt.savefig("arch.png")


plot()
calc_util(fasm_connections)
```

A.5 Grid plots

The following figures are generated after the different PnR runs from section 6.1.1 using the test designs as described in section 3.3.

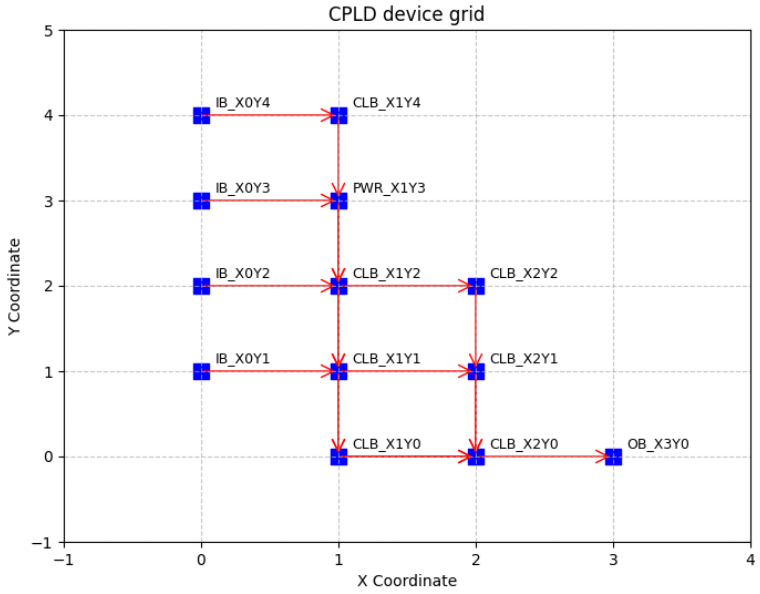


Figure A.1: Grid plot for PnR run of AND4 test for route-through architecture with grid size (4, 8)

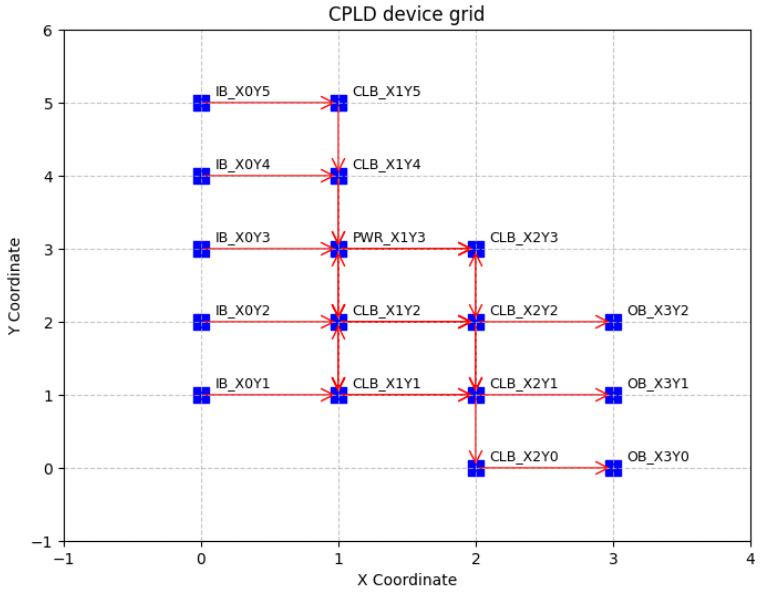


Figure A.2: Grid plot for PnR run of ADD2 test for route-through architecture with grid size (4, 8)

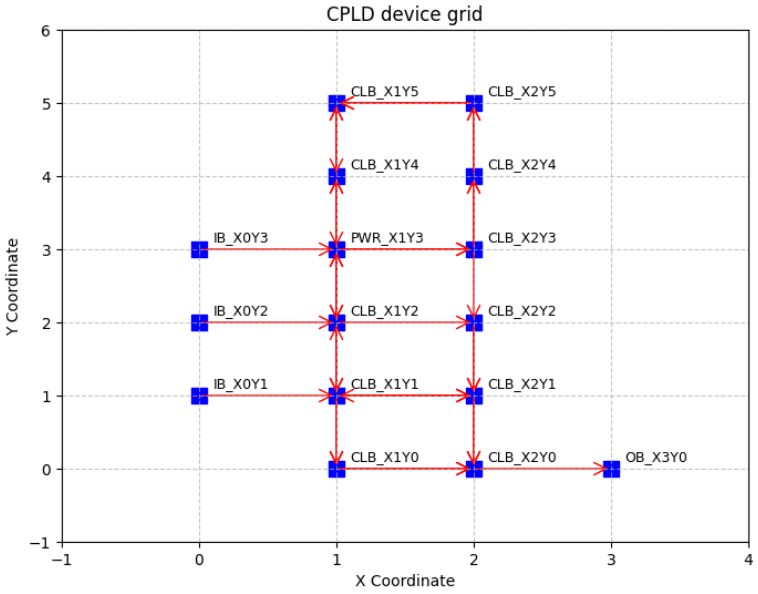


Figure A.3: Grid plot for PnR run of SR4 test for route-through architecture with grid size (4, 8)

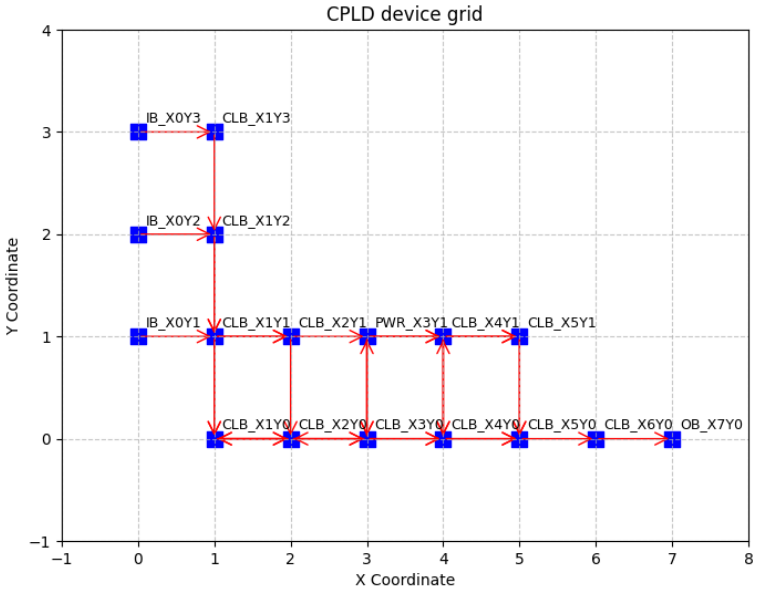


Figure A.4: Grid plot for PnR run of SR4 test for route-through architecture with grid size (8, 4)

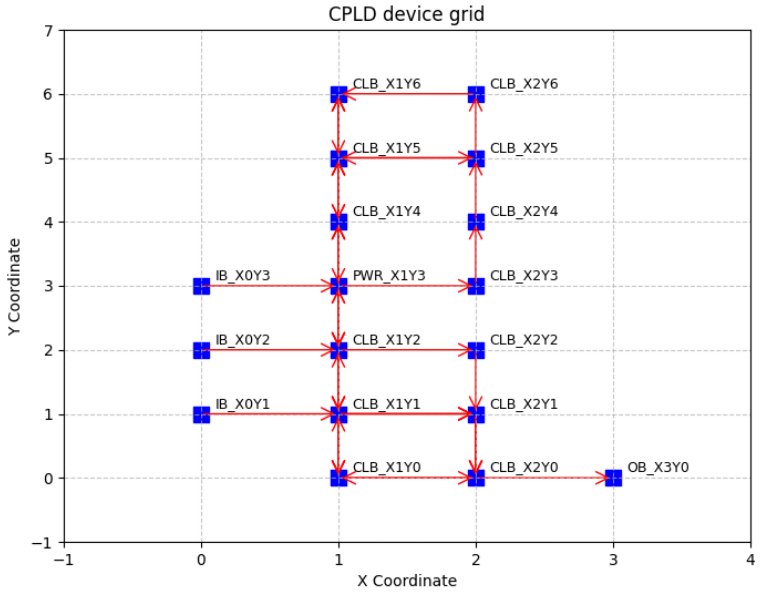


Figure A.5: Grid plot for PnR run of SR8 test for route-through architecture with grid size (4, 8)

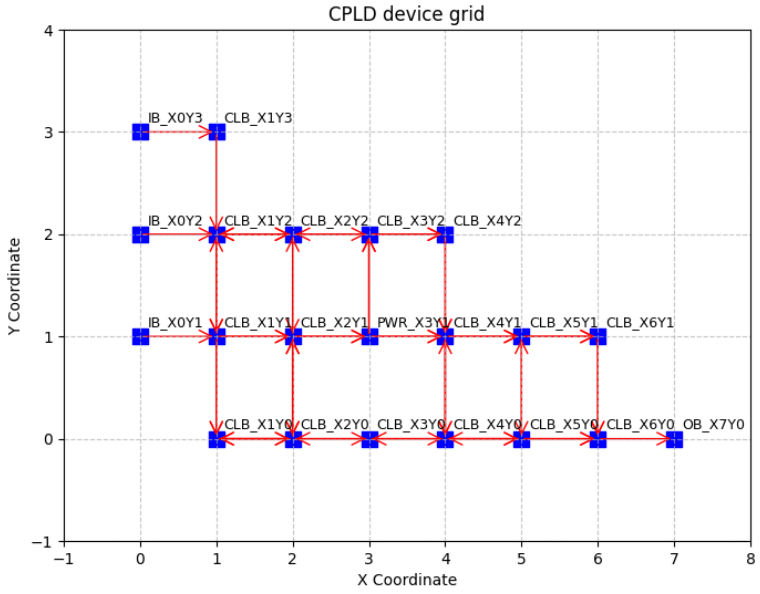


Figure A.6: Grid plot for PnR run of SR8 test for route-through architecture with grid size (8, 4)

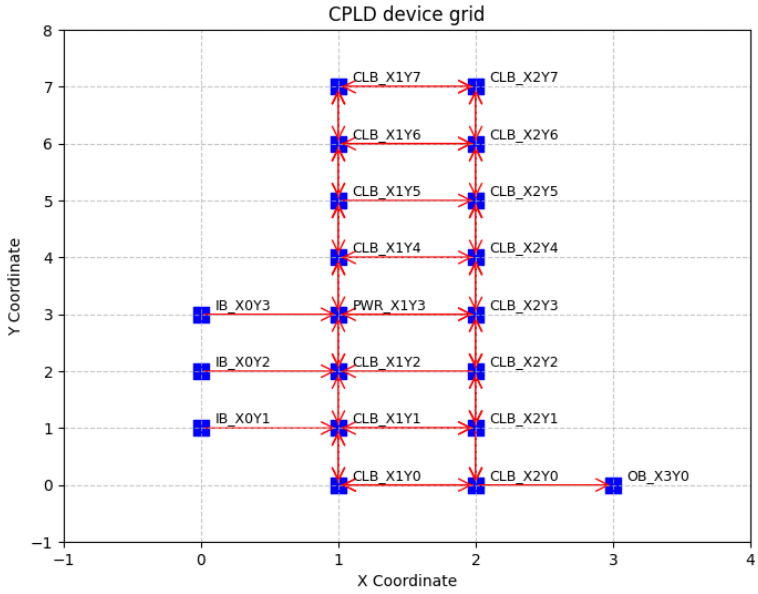


Figure A.7: Grid plot for PnR run of SR15 test for route-through architecture with grid size (4, 8)

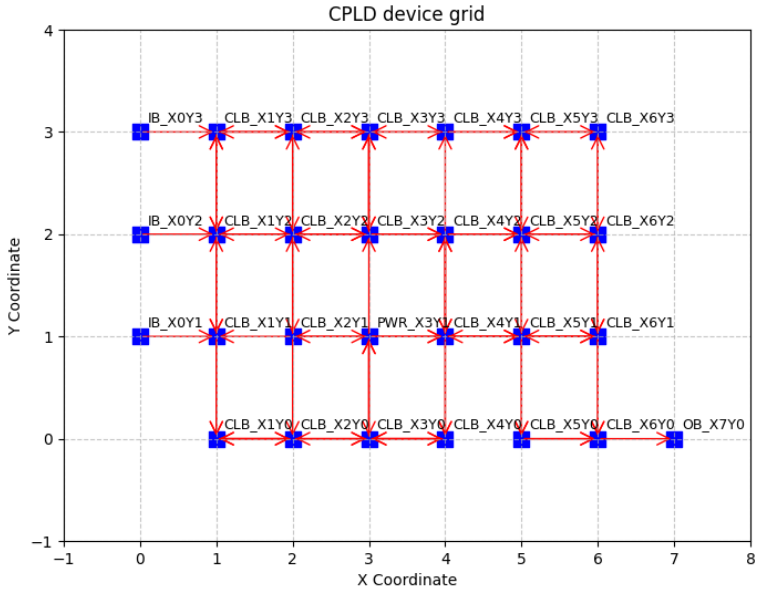
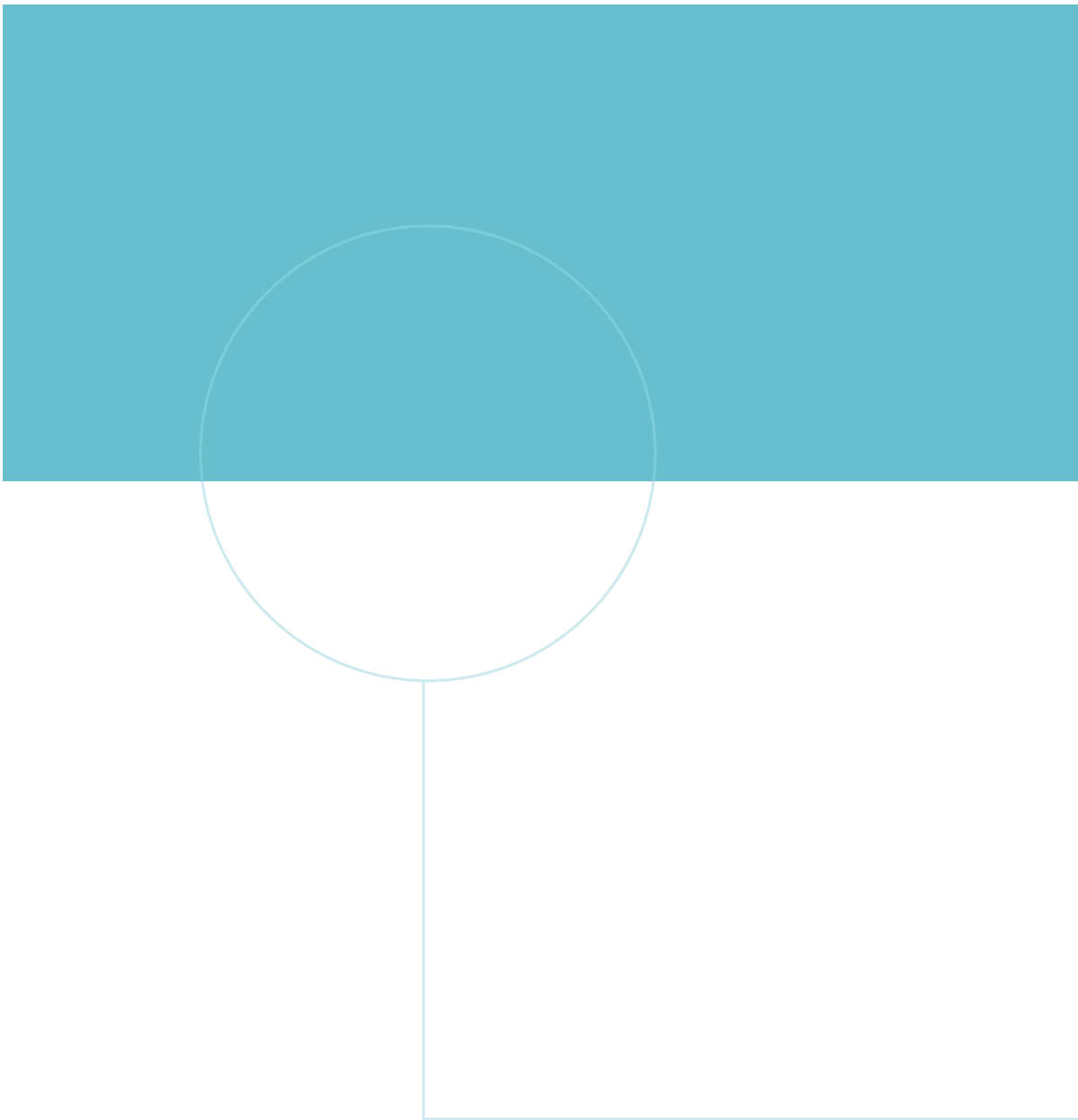


Figure A.8: Grid plot for PnR run of SR15 test for route-through architecture with grid size (8, 4)



NTNU

Kunnskap for en bedre verden