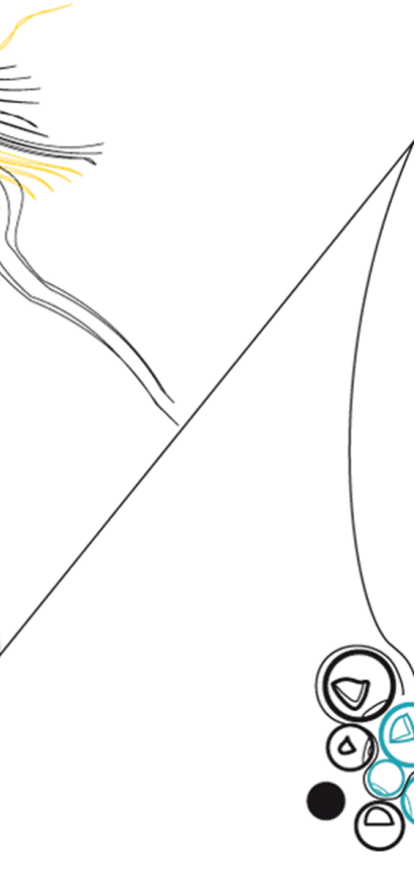
UNIVERSITY OF TWENTE.

Faculty of Electrical Engineering,
Mathematics & Computer Science

Analysis and Rerouting of Nets for Partial Reconfigurable FPGA Designs using RapidSmith2

Matthijs van Minnen

B.Sc. Thesis
July 2018



Supervisors:
dr. ing. D.M. Ziener
dr. ir. A.B.J. Kokkeler
dr. ir. R.A.R. van der Zee

Computer Architecture for
Embedded Systems Group
Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Summary

Field Programmable Gate Arrays (FPGA) are digital hardware devices that can be reprogrammed to implement a variety of tasks. In most application this is done before start-up. However, it is also possible to reconfigure during runtime. Current FPGA development platforms offer the ability to implement new functionality on a part of the hardware, otherwise known as *partial reconfiguration*. The implementation of this methodology is limited mainly because partial reconfigurable modules can only be constrained in their placement on the FPGA fabric. When these restrictions can be overcome, it is possible to implement more complex partial reconfigurable modules instead of the currently supported island configuration.

This work analyses the *routing* of partial reconfigurable modules and how this can be changed using the open-source tool RapidSmith2. This tool provides an important interface for projects created in Vivado. Furthermore, RapidSmith2 provides data structures that allow for the modification of individual cells and routes in the FPGA design. Using this tooling, it is possible to interfere in the normal design flow in order to analyse the interfaces of partial reconfigurable modules. Moreover, the tooling can be applied to identify nets crossing the border of these partial reconfigurable modules and re-route them using a simple algorithm. This has been performed and tested on a simple VHDL design to verify its operation. With this analysis, the paper provides a basic framework for the analysis of partial reconfigurable FPGA designs using RapidSmith2.

Original thesis description

Bachelor's Thesis:
"Design Flow for Creating FPGA-based Partial Reconfigurable Hardware Modules"

Student: Matthijs van Minnen
Supervision: Assoc. Prof. Dr. Daniel Ziener

Background:

Dynamic reconfiguration of FPGAs means the exchange of the FPGA configuration during runtime. *Partial dynamic reconfiguration* means that parts of the configuration can be exchanged during runtime, whereas the remainder of the configuration stays active. FPGA support for partial reconfiguration is the precondition for utilizing partial reconfiguration. However, a corresponding design flow in order to build such a system is also needed. A partial reconfigurable design is usually split into two parts: The a) *static part* is always present and only configured at power up of the system. In this part, usually the interfaces to peripheral devices, memory controllers, and the access to the configuration interface of the FPGA (e.g., the ICAP for Xilinx FPGAs) is included. The configuration of one or several b) *partial reconfigurable parts* or *areas* can be exchanged during runtime. These areas are usually embedded and surrounded by the *static part*. In these *partial reconfigurable areas*, modules and operations are implemented which can be adapted or exchanged during runtime [10].

The partial reconfigurable areas can be arranged in different configuration styles. The simplest configuration style is the *island style* which is capable to host one module exclusively per partial reconfigurable area. One drawback is the fragmentation if partial reconfigurable modules with different logic and routing utilization are used. The size of the partial reconfigurable area must be large enough to host all instances of the largest module which might result in a low utilization of the smaller modules. The negative effect of fragmentation can be reduced if the *slot* or *grid style* is used [10]. Here, the partial reconfigurable area is partitioned into slots or fields. The partial reconfigurable modules can utilize multiple slots or fields depending on the required amount of resources.

Relocation of partial reconfigurable modules means, that the same partial configuration can be loaded on different locations onto the FPGA which makes also possible to instantiate one partial reconfigurable module multiple times on the FPGA. A very flexible hardware system can be designed by combining relocation with the slot or grid configuration style. However, such a system needs sophisticated communication structures to establish the transfer of data in and out of the partial reconfigurable area and between the different reconfigurable modules.

Xilinx and Altera offer design tools for partial reconfigurable systems and sell licenses to enable this feature. Xilinx integrated the partial reconfiguration feature

in their design tool *PlanAhead* [5] and *Vivado*. Also, Altera supports partial reconfiguration for the new *Stratix V* series and integrated a partial design flow in their tools which is quite similar to the Xilinx approach [3]. However, these approaches support only an island reconfiguration style with the inclusion of static nets in the reconfigurable areas which forbids the relocation of partial reconfigurable modules.

To overcome these restrictions, the FPGA research community has introduced some partial design flows which are able to support the more advanced slot style and module relocation. A very comfortable flow for building partial reconfigurable systems is the tool *ReCoBus-Builder* [11]. This tool provides the easy generation of communication structures for bus-based and data-flow-oriented communications for the slot configuration style. The successor of *ReCoBus-Builder* is the tool *GoAhead* [2] which supports also newest Xilinx FPGA generations.

Problem statement:

In this bachelor thesis, the first steps to a novel design flow for building relocatable partial modules should be developed. Routing constraints are very important for the creation of such modules. However, Xilinx does not support routing constraints. GoAhead [2] deals with this issue by blocking routes which should not be taken by the Xilinx router. In this work, constraint-less routed modules should be analyzed in order to find nets that leave the desired area. If such nets are found, these nets have to be corrected by rerouting. This should be done by using the freely available tool "Rapidsmith2 " [7] which is able to modify placed and routed netlists by using a Java library.

The following issues should be solved:

- Get familar with the FPGA design flow and Xilinx Zynq architecture.
- Get familar with partial reconfiguration of FPGAs.
- Get familar with Rapidsmith2 [7].
- Set up a simple reconfigurable system.
- Develop an algorithm for detecting and rerouting net which leaving the desired area.
- Implement and test this algorithm by using Rapidsmith2 [7].
- Writing the thesis.

Contents

Chapter 1

Introduction

The area of the Field Programmable Gate Array (FPGA) has undergone a large growth over the last few decades which is thanks to the advantages this platform brings in the ever increasing digital generation. The market demands higher speeds at lower prices which requires dedicated hardware. Developing ASICs for every application requires long development cycles which in turn increases costs. The ever flexible FPGA platform is able to provide the basis for all the different applications and requires a reconfiguration to implement the desired task. This significantly reduces development time and cost whilst delivering high speeds as a result of the FPGA still being a hardware system.

Thanks to the flexibility in reconfiguring the FPGA it is simple to implement a large variety of functions on the platform by simply reprogramming it. The FPGA can be reconfigured to perform a completely different function. This is useful as it makes the general FPGA platform applicable in a variety of situations. By having a general FPGA platform it might occur that some of the hardware resources remain unused as it is not optimised for a specific task. This concept can be extended to several subsystems implemented *on* the FPGA. All of them take up a set amount of space, but not all subsystems will be used during all clock cycles. This problem is commonly known as fragmentation (in time) and should be avoided to improve the efficiency of the system.

A solution to this problem was derived by Xilinx in 2002 [17]. They proposed to reprogram/reconfigure the board during run time, which is known as reconfiguration. Since some portions of the board are always utilised, they will not be changed. Hence, a selected amount of area will be reconfigured; this is known as partial reconfiguration (PR). With this type of reconfiguration, modules which are not used can be replaced by other modules to increase efficiency. Besides increasing efficiency, PR could pose a solution to many other problems in FPGA development. An example here would be increasing the lifespan by moving a computationally intensive module around the FPGA and thus reducing the local heating and thus ageing of the device [1].

Both Altera and Xilinx have already implemented tools that allow for reconfiguration into their software package. For Xilinx devices this tool is implemented in the Vivado IDE. With these tools however, it is not possible to implement advanced reconfiguration structures (multiple reconfigurable slots) or to dynamically relocate modules. Community based tools, such as GoAhead [2], allow for these more complex type of structures. Unfortunately, the routing constraints required for this type of functionality are not possible in Vivado. To solve the problem, GoAhead the program simply blocks (deletes) inappropriate routes.

To allow for optimal use of the previously described designs, routes should not be blocked but instead be intelligently rerouted to still allow for the desired operation, this leads to the following question. How can the routing of partial

reconfigurable modules be analysed and altered on an FPGA device using RapidSmith2 and without blocking possible routes? This report will provide the basis for both analysis and rerouting and serve as a starting point for future research.

Before delving into a possible solution for this problem, Chapter 2 will describe the details of the FPGA design flow. Moreover, this chapter will discuss PR to a larger extend and explain the required tools: Vivado and Rapidsmith2 tools. The next Chapter 3 builds upon this by implementing algorithms that perform the rerouting. In Chapter 4 a PR project is created. The routing in this design is analysed and the performance of the algorithm is evaluated in Chapter 5.

Chapter 2

Theoretical background

The concept of PR and it's advantages were briefly discussed in the introduction. In this section, the details that are relevant for this research are explored with further depth. Given the research question, specifically PR methods and the corresponding routing thereof are of interest. These topics will be explored below and some concept are defined more properly.

2.1 FPGA design flow

Developing an FPGA application is performed using so called hardware description languages (HDL). With these languages the designer has more direct control over the hardware. The most two most common languages for programming FPGAs are VHDL and Verilog. For converting this HDL-code to a bitfile that can be uploaded to a device, multiple IDEs are available, e.g. Quartus (Altera) and Vivado (Xilinx). The programs perform a number of steps before creating the bitstream that is uploaded to the FPGA board.

An overview of the FPGA design flow has been created in Figure 2.1. The steps are elaborated here. The designer starts with designing a circuit in the preferred HDL (Step 1). All the required functionality is implemented in this design. This code is the starting point of the design flow. Like with any other type of programming the code is validated to see if it actually performs what the designer intended, this is performed in the second step. If the performance is satisfactory, the code can be synthesised (e.g. converting the code to a netlist of LUTs and FFs) in Step 3. Step 4 and 5 are often mentioned together [4] but are in fact separate which can be exploited for use of PR or other more advanced techniques, such as manual floorplanning[1]. By extracting the design before placing or routing, man-

[1] "*Floorplanning is the process of choosing the best grouping and connectivity of logic in a design, and of manually placing blocks of logic in an FPGA, where the goal is to increase density, routability, or performance.*" [18] The floorplanning can be (manually) optimised to decrease the critical path and allow

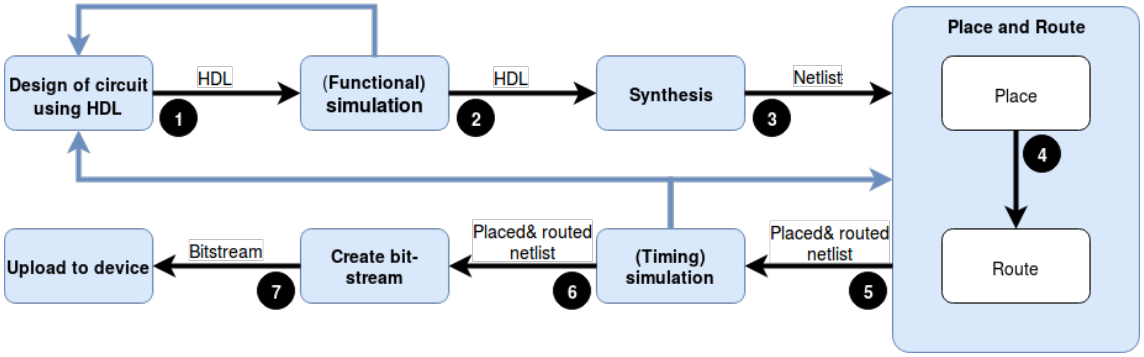


Figure 2.1: The design flow from creating the code to uploading the code on an FPGA device.

ual adjustments can be made which can help in implementing such systems. After this step is completed, another simulation is performed to test the timing of the system and to identify possible errors. When timing is not satisfactory, routing and placing could be adjusted or the initial design should be changed (see blue lines in Figure 2.1). When everything is complete, the placed and routed netlist can be converted to a bitstream which can be uploaded to the FPGA board (Step 7).

It must be noted that planning and routing is completely separate from the original design and dependent on the device it will be uploaded on. The result of synthesis will remain the same however. Hence, the designer can influence the design both through the initial code or by intervening in the placement and/or routing procedure.

2.2 Partial reconfiguration on the FPGA platform

The goal of PR is to exchange *modules* on the FPGA platform during runtime. Doing so requires the routing of I/O signals to the correct location on the FPGA. To be able to perform this, a portion of the FPGA needs to keep executing the required tasks (including the reconfiguration). This part of the FPGA is known as the *static region*. This region will also host static parts of the system, such as I/O ports as they can never be (physically) moved. Logically, the part that will host the reconfigurable modules is named *dynamic region*. This region can host a variety of different modules which can be initialised whenever required. The application of this region is limited by the configurable logic blocks (CLB) available in the hardware. Designers should take this into consideration when creating their reconfigurable modules.

2.2.1 Partial reconfiguration methodologies

Currently there are a select number of choices for PR methodologies. The simplest and most widely supported form is the *island style* reconfiguration. The PR region is surrounded by the static region (hence the name; island) and the area can exclusively host one PR module at a time. The advantage of this type of configuration is that the interfaces to the module can be standardised, simplifying the communication from the static to the dynamic area. There are however, a number of disadvantages that can be decreased by utilising a different methodology. For instance, the island should be large enough to host the largest PR module. In which case the utilisation of the available area is optimal. When a smaller module is inserted however, a large portion of area may go unused which is known as internal fragmentation. The fact that solely one module can be hosted in an island means that this unused area cannot be utilised otherwise.

To overcome these limitations, the *slot and grid style* reconfiguration methodologies were conceived [10]. The first divides the dynamic area into vertical slots of a fixed size, the latter divides the area in both horizontal and vertical direction, e.g. a 2-D grid of relocatable chunks (see Figure 2.2). With these styles, PR modules can use several slots/chunks to create the required area to host a PR

for an increase of the clock frequency.

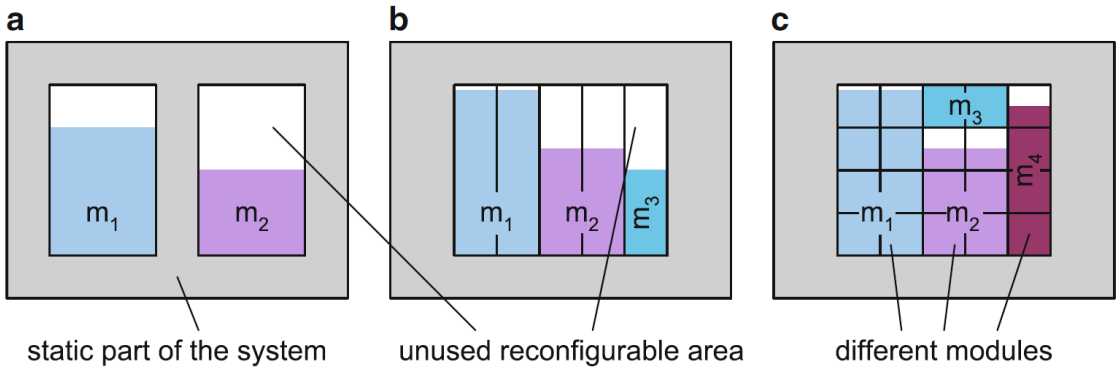


Figure 2.2: Three different PR methodologies: a) Island style, b) slot based and c) chunk based. Taken from [10].

module. If a module does not fully utilise a slot or chunk, some area goes unused, but this amount is significantly reduced when compared to the area of a larger island. Supporting these reconfiguration styles does require more effort however. Instead of communicating with a single island, several slots/chunks need to be interfaced. Next to the fact that slots/chunks need to communicate internally to perform the required task. Enabling all of this communication requires overhead which decreases the efficiency of the implementation [10].

It was stated before that slots or chunks have a fixed size. For communication with each section, a fixed amount of overhead is required. Hence, it would make sense to make larger slots/chunks. But this would reintroduce the issue of internal fragmentation. Hence, a trade-off should be found between the size and communication overhead of the slots/chunks.

2.2.2 Link between static and dynamic region

The static region should, as the name suggests, remain the same. At the same time, however, several different dynamic modules could be loaded during runtime. These should all work correctly and be interfaced with the static region. To accomplish this, a standardised interface has to be created. This is not a new idea as seen in sub-figures a) and b) in Figure 2.3. These methods utilise cells which force the router to make a connection to the edge of both the PR module and the static region. When inserted, the PR module can then interface with the static region. In the case of Figure 2.3 the older bus macros[2] or the newer proxy logic are used. The use of these cells means an additional overhead of two cells for each connection. Additionally the proxy logic cells used today are set to route-through which makes them behave like a wire. Except for the fact that they introduce a small delay. When dealing with larger streams of data this delay can become significant and is hence not desired. Figure 2.3 also proposes a new method where PR-links, or in other words wires, are used for the connection. Modern day routers are not able to create this type of PR-links, which is why a different solution must be sought for.

[2]Bus macros were required in the design for the router to have a point to attach to.

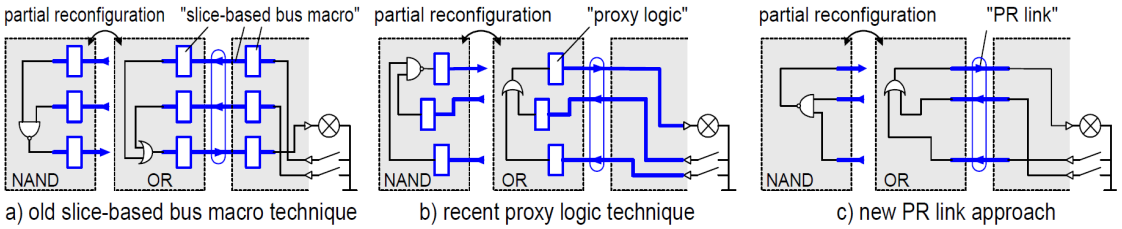


Figure 2.3: Three different methods of connecting the dynamic to the static system. Taken from [12].

2.2.3 Switching PR modules

When all prior steps have been successfully completed, PR modules must be loaded onto the FPGA board. The first module can be loaded with the static system in what is known as a full-configuration. However, the goal with PR modules is to exchange them during runtime. This can be achieved in a number of ways. Using for example the Vivado IDE (see Section 2.3) it is possible to load a partial bitstream solely containing the information for the partial module. As such the information in the RP site will be overwritten with the information of the new module. Loading this partial bitstream is done using a *JTAG interface*. Alternatively, the Zynq family (with boards such as the Zedboard; see section 2.5) provides the ability to upload a bitstream using the onboard processing system. Using this, the bitstream can be loaded into the programmable logic at any time using a *PCAP interface*. By utilising the communication between the two parts, the module can be loaded at the appropriate timing. A final option is to utilise *ICAP* to reconfigure the fabric. This method is similar to PCAP, but now the FPGA fabric itself implements a different module. This allows the FPGA to independently host partial dynamic reconfigurable projects.

2.3 The Vivado IDE

As previously mentioned, the Vivado IDE [9] is one of the most commonly used tools for performing; synthesis, placing, routing and creating the bitstream (see Figure 2.1). This IDE provides a graphical and a TCL interface. Moreover, it provides support for the island style partial configuration. Still, implementing a design requires some effort as each PR module has to be processed individually. The first step is to synthesise, place and route the static region together with a black-box that allocates the PR region. After this process is completed, the black box can be removed and a checkpoint can be made. Using this checkpoint, each PR module can then be placed and routed to provide a bitstream for each individual configuration. Since Vivado only supports island style PR, it is not directly possible to implement the slot style reconfiguration. However, it is possible to allocate the desired area such that it can be configured as a PR slot using a different method.

2.3.1 Implementing a PR island

The first step in implementing is to individually synthesise all parts of the design. This means both the static design as well as all PR modules individually. A checkpoint must be created for each synthesised project, such that they can be loaded into the design later on. Next, the static design synthesis checkpoint must be loaded together with either a PR module, or a blackbox indicating the location of the modules. This design can then be placed and routed, after which the module/blackbox can be removed to leave only the static design. A checkpoint must be created of this design because this can then be used to execute placement and routing for all the other PR modules, each time loading a different module into the static system. A bitstream can be made for each implementation independently, such that it can be uploaded to the FPGA board (see Section 2.2.3).

It must be noted that all of these steps can be combined and executed using a TCL file. This automates many of the steps but relies on the Vivado tools to properly execute all steps. In some instances it can be good to manually execute certain steps, such as allocating pblocks, to improve the overall design.

2.4 Rapidsmith2

Whereas Vivado tries to automate and abstract many processes, RapidSmith2 (RS2) [13] provides a lot of low level control (it is possible to interact with individual BELs[3] for example). As such, it is possible to change both the floorplan and the routing on a larger scale, as well as fine-tune small elements. Installing and using RS2 is simplified through the use of the techreport [15]. This document also explains how to use many of the components/functions. A small summary of the relevant information is given here for use in the report.

2.4.1 Storage types

To store the design created in Vivado, RS2 uses a number of variables. In order to properly use RS2, a good understanding of the different variables is required:

- **Device**: An overview of the platform; e.g. what sites are placed where in the fabric.
- **Design**: Provides an overview of the design created by the user; e.g. what cell types (from the cell library) are placed where in the fabric *and* how these cells are routed.
- **Cellnet**: An overview of the connections between netlists.
- **RouteTree**: A structure to store routes in, more details are given in Section 2.4.3.
- **Cell Library**: A library with all available cell types to be implemented in the design.
- **Tile**: Equal to Vivado tiles; contains a number of sites.

[3] Basic element, e.g. a LUT or flip-flop.

- **Site**: Equal to Vivado sites; contains a number of cells.
- **Cell**: Similar to Vivado cells, the available types are found in the cell library.
- **Wire**: A structure for describing a connection. More details are given in Section 2.4.2.
- RS2 provides more datastructures (e.g. BELs, PIPs, etc.), but these will not be discussed in detail as they are not relevant for the routing. These structures are mainly used for implementing low-level changes to the design.

2.4.2 Manipulation commands

Using RS2 it is possible to list all elements in the design, e.g. tiles, sites, cells, BELs, wires and PIPs. A list of all items can easily be invoked by issuing: `getTiles()`, `getSites()`, etc. Alternatively, specific elements can be searched for by for example issuing `getBEL('PAD')`. This will return all *PAD* elements on the specific tile/site.

Wires

RS2 implements a special structure for documenting wires. In essence, there are two wire types: Programmable Interconnect Point (PIP) connections and Non-PIP connections. Here Non-PIP connections are simply a wire connecting to another wire, thus forming a simple connection. The PIP connections are locations where two wires are connected using a PIP which can be programmed. The PIP connections can connect to a number of specific objects; site pins, BEL pins or a (BEL) route-through connection. The most important parts of a wire connection are the source and sink(s), these are labeled, whereas the Non-PIP connections are not [15]. The power supply and ground nets are handled separately, but do not have to be changed manually for changing the routing, so will not be discussed into more depth here.

To determine how routing is performed, more insight into the wire structure is required. RS2 identifies three major commands for retrieving information about wires [15]:

- "`mywire.getWireConnections()`: Returns a collection of all *Connection* objects whose source is "mywire". This collection can be iterated over to find all places a specific wire goes (i.e. what wires it connects to).
- `conn.isPip()`: Returns true if the wire connection "conn" is a PIP connection. Returns false otherwise.
- `conn.getSinkWire()`: Returns the sink wire of a wire connection."

The RS2 techreport goes into more detail and gives code examples for using the specific commands and identifying the different types of wire (connections).

2.4.3 Route trees

The first version of RapidSmith had no way of managing the *wire* elements. A great improvement with RS2 is the introduction of RouteTrees, which manages

these object for the user. To effectively change the routing, the structure of the route tree must be properly understood.

A *RouteTree* is a *struct* for each wire that contains about the connection it is part of. It links the source of the connection and also lists all the sinks. Moreover the *struct* contains the route from the source element to the current *RouteTree* element [15]:

- **Wire**: the actual wire the *RouteTree* is describing. This can be any of the wires described before.
- **Source**: The name of the source of this connection. Here source is of the previously described type.
- **Connection**: shows the path from the source to the current *wire*.
- **Sinks**: Shows the *sinks* that are part of the *connection*.
- **Cost**: A field for entering the cost of a *wire*. This can be useful for routing algorithms.

2.4.4 Implementing routing

RS2 converts the routing of Vivado in a three-part routing structure. The first part is the lower level routing within a site. The second part, intersite routing, connects different sites together. Once a connection has been made between sites, the third part of the routing connects the intersite routing pin to elements within the site.

If there are routes leaving the specified tiles, this must be the intersite routes. Since a connection is required between the two sites, a route must be constructed between them. It is possible to adjust the routing however, fit within the specified tiles of the module. Since this route is part of the route tree (See Section 2.4.3) the elements currently part of the tree must be removed. Subsequently, a new path must be created and the corresponding elements must be added to the routetree to once again complete the structure.

If the new route does not conflict with any of the existing routing, the design is still able to perform the original task as the sites are still connected. However, this new implementation does not contain nets that move outside the specified bounds.

Routing algorithms

In the example section RS2 provides two samples of routing algorithm. The first is very simple and named handrouter. This algorithm analyses the connections available to the *routeTree*. These options are printed and it is up to the user to select the best route. As the name suggests, a manual solution.

Alternatively, RS2 provides the A* (A-star) router which is based on the original A* algorithm created by the Stanford Research Institute as early as 1968 [8]. This algorithm is not the most efficient, but provides a simple starting point for developing new routing algorithms.

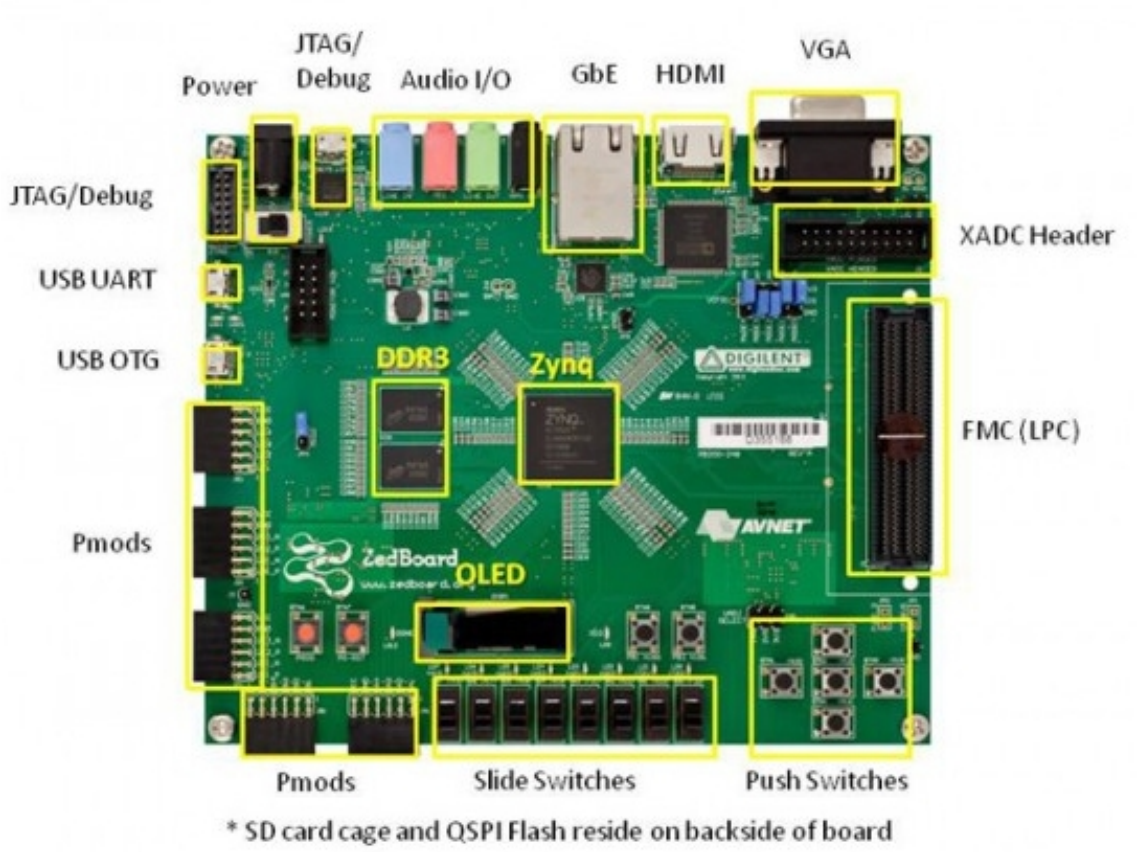


Figure 2.4: An overview of the Zedboard and it's interfaces from the Vivado software [9].

2.5 The Zedboard

To actually test the implementation on hardware, a platform to work on is required. Within the Zynq family, the Zedboard, an educational board, is available. Besides the Artix-7 FPGA with 53,200 LUT's and 106,400 FF's, this board is equipped with a dual ARM® Cortex^{TM}-A9 MPCore^{TM} which can operate up to 866MHz. Moreover, it has multiple I/O options, such as: slide switches, push buttons, LEDs and a 128x32 OLED screen [20]. These interfaces and more are shown in Figure 2.4.

Since the goal is to adapt routing on the FPGA, the ARM cores will not be used. In an actual application, these can be useful to extend the applicability of the board. The several I/O options however allow for easy debugging as they can quick visualising what is going either right or wrong.

Chapter 3

Analysing the Vivado routing

A number of elements of the created example design have to be analysed; the proxy logic used to connect the static and dynamic parts, the routing of the partial reconfigurable module and if the nets in the module leave the specified area. Once incorrect routes are detected they must be corrected, which is discussed in Section 3.3.

3.1 Proxy logic

In the 2017 version of Vivado, Xilinx presented the partition pins [19]. These pins satisfy the desire as posed in Figure 2.3 as it does not require additional physical cells which require resources and could influence timing performance. Since the interfaces between the static design and the reconfigurable modules are now handled by Vivado, it is not required to implement these using RS2. What *can* be analysed is the routing to-and-from these pins. If this is different for different modules it can have an influence on the performance between modules.

Before partition pins were implemented, an additional step had to be performed using RS2 to insert connection points between the static and dynamic region to avoid the use of proxy logic. In the list described in Chapter 4, this would have been inserted after Step 2. There the project would be exported to RS2, the desired interfaces would be inserted, after which the design would be reinserted into Vivado.

3.2 Routing analysis

The second item to be investigated is if routes go outside the boundaries set for the (PR) module. This is not meant to happen as it might have influences on the (routing of the) static system. Detecting the crossing of these boundaries is implemented using functions which are elucidated in Sections 3.2.1 and 3.2.2.

3.2.1 Specifying bounds

When determining routes that go out-of-bounds, the first step is to define what the boundaries are. The function in Listing A.1 does this using two tiles which act as two corners of a rectangle. The function then creates a `Collection<Tiles>` of all selected Tiles within the indicated rectangle. This algorithm is implemented using Java and the RS2 framework and is based on the functionality described in Section 2.4. Once a collection has been made of all selected tiles, it is possible to find the nets originating from those tiles and make a `Collection<CellNet>` out of these, which is performed in Listing A.2 These collections provide the basis for the incorrect route detection algorithm.

3.2.2 Detecting incorrect routes

Once the allowed tiles are specified and the nets originating from these tiles are found, it is possible to find all individual route trees and determine if the routes they describe are actually going outside the determined tiles. This is done in Listing 3.1. The detection of routes that leave the specified P-block is handled using the recursive function `iteratingOverRouteTree` which is called by a helper function `sinksOutsideArea`. The recursive function performs all the work and calls upon itself to traverse along the net. The function has methods for determining the different types of connections (e.g. leaf cells which connect to a site or BEL pin, or a simple wire connection) and is based on the design analyser example from RS2. The goal of this algorithm is to find routes that originate from the P-block and return to the P-block again. Other routes that leave the P-block might connect to the static region or I/O-pins and should not be removed.

The `iteratingOverRouteTree` function marks any route leaving the P-block from a site pin as a possible incorrect route. If this route then leaves the specified tiles, a flag is raised. This flag is recursively passed down, until the route reaches a leaf cell again. At this point it is evaluated on which tile the connected pin is located. If this is a tile part of the P-block, the net is added to the list of incorrectly routed nets. If this is not the case, the route is continued until the end.

It was chosen to iterate over the routetrees to find all relevant information. This way, all information is based on the current implementation. Another possibility would have been to utilise the `targetTile` variable which is used in the A* algorithm. This way, as soon as a route leaves the specified tiles, this variable can be evaluated to find if a route is correct or not. This does assume that this variable is always correctly set, which is not guaranteed by RS2.

```
1  public static Collection<SitePin> sinksOutsideArea(Collection<Tile> selectedTiles, CellNet
   selectedNet) {
2          Collection<SitePin> sinksToRoute = new ArrayList<>(); //Start a list to add sinkPins to
3          if (selectedNet.isSourced() && !selectedNet.isGNDNet() && !selectedNet.isVCCNet()
           && !selectedNet.isClkNet()) {
4                  sinksToRoute.addAll(iteratingOverRouteTree(selectedNet,
                   selectedNet.getSourceRouteTree(), true, 0, selectedTiles));
5          }
6          return sinksToRoute;
7  }
8
9  public static Collection<SitePin> iteratingOverRouteTree(CellNet n, RouteTree rt, boolean
   inside, int possible, Collection<Tile> selectedTiles) {
10         Collection<SitePin> wrongPins = new ArrayList<>();
11         if (rt == null) return wrongPins;
12
13         Collection<RouteTree> sinkTrees = rt.getSinkTrees();
14         if (rt.isLeaf()) {
15                 SitePin sp = rt.getConnectedSitePin();
16                 if (sp != null) {
17                         if (inside) {
18                                 // Inside site, so look for correct intersite route tree to leave on
19                                 for (RouteTree rt1 : n.getIntersiteRouteTreeList()) {
20                                         if (sp.getExternalWire().equals(rt1.getWire())) {
21                                                 possible = 1;
22                                                 wrongPins.addAll(iteratingOverRouteTree(n,
                                                   rt1, !inside, possible, selectedTiles));
```

```
23                                                  return wrongPins;
24                                          }
25                                  }
26                                  return wrongPins;
27                          }
28                          else
29                                  // Outside site, so just follow the route from the general
                                    routing fabric and into a site
30                                  if(possible == 2 &&
                                    selectedTiles.contains(sp.getInternalWire().getTile())) {
31                                          for(SitePin sitePin : n.getSitePins()) { //Add all the
                                            input sitePins from the net
32                                                  if(sitePin.isInput())
33                                                          wrongPins.add(sitePin);
34                                          }
35                                          return wrongPins;
36                                  }
37                                  possible = 0;
38                                  wrongPins.addAll(iteratingOverRouteTree(n,
                                    n.getSinkRouteTree(sp), inside, possible, selectedTiles));
39                                  return wrongPins;
40                  }
41          } // End of rt.isLeaf()
42
43          else {
44                  // Otherwise, if it is not a leaf route tree, then iterate across its sink trees
45                  for (Iterator<RouteTree> it = sinkTrees.iterator(); it.hasNext(); ) {
46                          RouteTree sink = it.next();
47
48                          if(!selectedTiles.contains(sink.getWire().getTile())) { //If it is in the
                            wrong tile add it to the list
49                                  possible = 2;
50                                  wrongPins.addAll(iteratingOverRouteTree(n, sink, inside,
                                    possible, selectedTiles));
51                                  return wrongPins;
52                          }
53                          wrongPins.addAll(iteratingOverRouteTree(n, sink, inside, possible,
                            selectedTiles));
54                  }
55          }
56          return wrongPins;
57  }
```

Listing 3.1: The code for identifying which nets leave the area specified by the function from the previous Section 3.2.1.

In the helper function (`sinksOutsideArea`), another check is being performed to avoid analysing nets that are either for the clock, VCC or GND since these do not follow the standard routing method and do not have to be adjusted.

In the recursive function it can be found that the leaf cells are analysed in lines 14 to 41 and the wire connections in line 45 to 54. A flag is raised if routes leave the site. If one of the wire connections is then located outside the boundaries a second flag is raised. The route is returned if a route is detected that left the boundaries and ends at a site pin within those boundaries. Once this occurs, all input pins of the net are added to `wrongPins` and they are recursively passed up such that a big collection of pins is returned to the helper function.

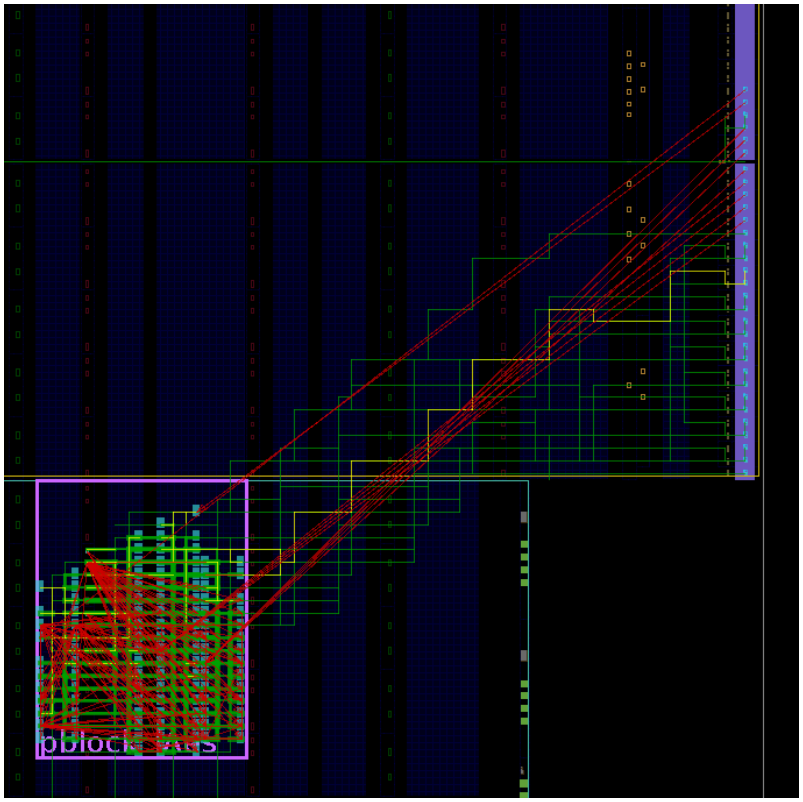


Figure 3.1: The routes in the AES design after being removed by the routes-out-of-bounds function. It can be noted that the top I/O-bank is now disconnected.

Removing selective routes

The first iteration of this function removed almost all routes leaving the P-block, since it performs a simple check if the intersite-route ever leaves the specified tiles. The result of which can be found in Figure 3.1. Since communication between static and dynamic region is required, these routes should not be removed. It can be seen how the routes to the top I/O-bank are disconnected, which is undesirable. To solve this problem, the algorithm described previously was properly implemented. Now only routes leaving *and* returning to the P-block are marked as incorrect. This yields the results found in Figure 3.2.

With the updated code, a collection is filled with pins attached to the found incorrect nets. Based on the *designAnalyser*, the recursive function iterates over all elements. For the original purpose this example function prints every element in the net. Instead, this application performs an analysis on each element to see if it is not on one of the allowed tiles, the corresponding input pins of the net are added to the collection. From Figure 3.2 it can be seen that all the out-of-bounds routes are found and removed, whereas the other routes (such as those to the static region) remain.

3.3 Alternative routing

As discussed in Section 2.4.4, RS2 provides the A* algorithm as example code. The re-routing is performed based on this algorithm. Since the goal is to avoid

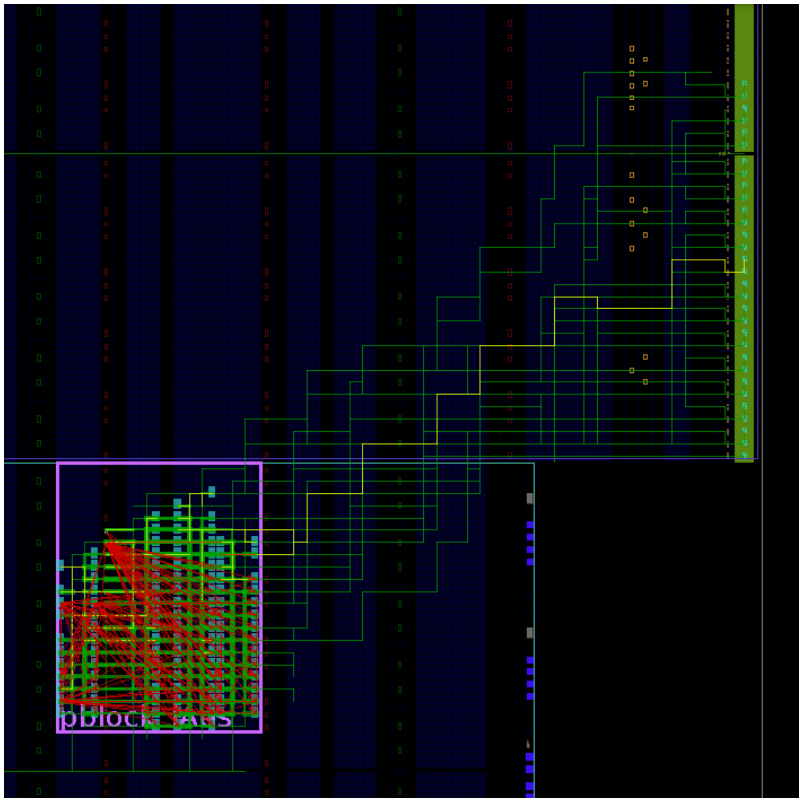


Figure 3.2: The routes in the AES design after being removed by the routes-out-of-bounds function. With the altered algorithm, only routes returning to the P-block are removed. The routes at the bottom of the P-block are routed to a site elsewhere.

going out-of-bounds, the routing algorithm was slightly adjusted to incorporate the current tile the wire is in such that it will not again create an incorrect route outside of the bounds. The new implementation of the A* `routeNet` function is shown in Listing A.3. The function receives the appropriate tile and net collections created using Listings 3.1 and 3.1 to determine which *SitePins* must be re-routed.

To get it working properly, the priority queue –which is used to select the most cost efficient route– must be initialised. This way, a `isEmpty()` call can be made to determine if a new queue must be made using the `resortPriorityQueue` function from the original RS2 algorithm.

3.4 Wrapper

To call all previously described functions in the correct succession, a class named `routeOutOfBounds_Example` is used. It is given in Listing 3.2. This function first determines the correct tiles and nets using the functions from Listings A.1 and A.2. Subsequently it calls the function from Listing 3.1 to determine which routes go out of bounds. It was found that one should be very careful which nets to unroute, otherwise errors will occur when exporting the adjusted design. Hence, this function only removes routes if there are incorrectly routed pins. Since those will also be rerouted by the algorithm from Listing A.3. After all the alterations have been made to the design, statistics are printed to represent the quality of the work performed.

```
1  public static void main(String[] args) throws IOException {
2          double startTime = System.nanoTime(); //This is the time at which the analysis is
           started
3          // load the device and design
4          String checkpoint = "/home/matthijs/AES3.3.rscp";
5          System.out.println("Loading Device and Design...");
6          VivadoCheckpoint vcp = VivadoInterface.loadRSCP(checkpoint);
7          CellDesign design = vcp.getDesign();
8          Device device = vcp.getDevice();
9          CellLibrary libCells = vcp.getLibCells();
10
11         // loading reverse wire connections
12         device.loadExtendedInfo();
13
14         //Creating variables which can hold both the routetree and the statistics
15         Results results = new Results();
16         int reroutedRoutes = 0;
17         int correctRoutes = 0;
18         int wrongRoutes = 0;
19         int errorRoutes = 0;
20
21         // Routing net
22         System.out.println("Re-routing Nets...");
23         RouteOutOfBounds router = new RouteOutOfBounds();
24
25         //Find the selected Tiles and nets originating from those Tiles
26         Collection<Tile> areaTiles = RouteOutOfBounds.selectingArea(device,
           device.getTile("CLBLM_L_X36Y28"), device.getTile("CLBLM_R_X43Y49"));
           //Collection of allowed Tiles
27         Collection<CellNet> areaNets = RouteOutOfBounds.selectingNets(areaTiles, design);
           //Collection of allowed nets (those leaving from the allowed Tiles)
28
29         for(CellNet net : areaNets) {
30                 System.out.println("\tCurrently working on net: " + net.toString());
31
32                 // Find the pins that need to be routed for the net
33                 Iterator<SitePin> sinksToRoute =
                   RouteOutOfBounds.sinksOutsideArea(areaTiles, net).iterator(); //Iterator for
                   the sinks that must be re-routed
34
35                 if(sinksToRoute.hasNext()) {
36                         net.unrouteIntersite();
37                         results = router.routeNet(areaTiles, areaNets, sinksToRoute, net);
38                 }
39
40                 if(results.routeTree != null) {
41                         net.unrouteIntersite();
42                         net.addIntersiteRouteTree(results.routeTree);
43                 }
44
45                 //Update the statistical values
46                 reroutedRoutes += results.reroutedRoutes;
47             correctRoutes += results.correctRoutes;
48             wrongRoutes += results.wrongRoutes;
49             errorRoutes += results.errorRoutes;
50         }
51
52         // Displaying results
```

```
53          System.out.println("Done!");
54          double estimatedTime = (System.nanoTime() − startTime)/1000000000;
55          System.out.println("This took " + estimatedTime + " seconds");
56
57          System.out.println(reroutedRoutes + " pins were rerouted of which " + correctRoutes +
            " were correct, " + wrongRoutes + " could not be rerouted and " + errorRoutes + "
            caused an error.");
58          float percentage = (correctRoutes∗100f)/reroutedRoutes;
59          System.out.println("Therefore, " + percentage + "% is correct.");
60
61          // Re−evaluate if routes are correct by applying the sinksOutsideArea function again
62          int counter = 0;
63          for(CellNet net : areaNets) { //Iterate over all nets again and see if wires leave area
64                  if(!net.isClkNet() && !net.isGNDNet() && !net.isVCCNet()) {
65                      for (Iterator<SitePin> sinksToRoute =
                        RouteOutOfBounds.sinksOutsideArea(areaTiles, net).iterator();
                        sinksToRoute.hasNext();){
66                          SitePin sink = sinksToRoute.next();
67                          if(sink != null)
68                                  counter++;
69                      }
70                  }
71          }
72      System.out.println("\nThe routes−out−of−bounds function found that " + counter + "
        routes are still incorrect.");
73
74          // Evaluate all nets
75          int numrouted= 0;
76          for (CellNet n: design.getNets())
77                  if (n.getIntersiteRouteTreeList()!= null)
78                          numrouted++;
79          System.out.println("The design has: " + design.getNets().size() + " nets, " + numrouted +
            " of them are routed.");
80
81          //Export the altered design
82          System.out.println("\nExporting now...");
83          VivadoInterface.writeTCP("/home/matthijs/Documents/temp.tcp", design, device,
            libCells);
84          System.out.println("Done!");
```

Listing 3.2: The code for importing a Vivado design and calling the appropriate functions to reroute nets that go outside the specified boundaries.

Chapter 4

Example applications

The general goal is to adapt the routing of the FPGA to make it more coherent. Doing this requires a number of steps which are listed below. This list does assume that a PR design is ready and synthesised and only goes through the steps that follow after synthesis.

1. Load the static design into Vivado.
2. Allocate black box(es) for the PR module(s).
3. Perform the usual design flow steps; place and route (e.g. 4 & 5 from Figure 2.1).
4. Perform the same steps for all PR modules in order to get checkpoints for all of them. As described in Section 2.4.
5. Apply *pr_verify* in order to validate all PR implementations.
6. Export to RS2 to evaluate the routing of the module and possibly adjust it.
7. Convert the project back to Vivado and create the final bitstream (step 7 from Figure 2.1).

Analysing and testing the implementation of adjusted routing requires an example program on which this analysis can be applied. This will need to be a PR-system so, this requires a static system able to host a PR module. Two projects were utilised. On the one hand an AES core [6] and on the other a project that can create random numbers using one module and test for primality with another. The latter will be created to work with larger (32 bit) prime numbers, which will require a significant amount of logic. Both the module for prime verification as well as the random number generator module will utilise the same interface structure. The exact code implementation for the prime verification can be found in Appendix B.

4.1 AES core

In contrast to many other applications, the verification of the re-routing requires projects that are large, such that there are a number of routes (that leave the module). Hence, a 128-bit AES core was sourced to serve as an application. The project is subdivided into smaller sections. Unfortunately, all of these sections do not contain much logic. So, instead, the entire core is implemented as a module fitting within a simple wrapper function. The resource utilisation is smaller when compared to the prime verification (see section 4.2) with *738 LUTs and 327 FFs*.

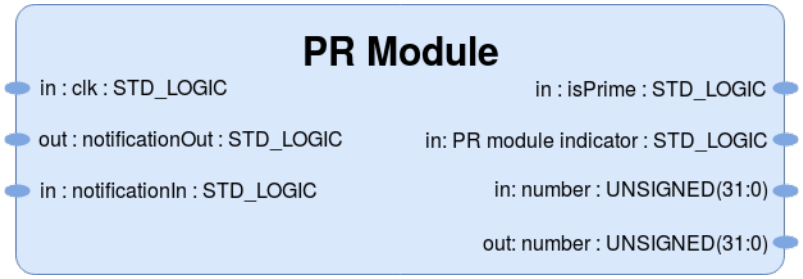


Figure 4.1: The interfaces of the partial reconfigurable modules.

4.2 Prime verification

To allow for the switching between PR modules, a standardised interface is required. In this way, each module connects to the same interfaces and the routing of the static system can be simplified. For the created system (see Appendix B for the source code), a number of interfaces are defined for the PR modules as defined in Figure 4.1.

The difference between the two modules is that one should send and the other receive the 32-bit random number. VHDL offers `'inout'` interfaces. These are however, not supported by the FPGA fabric as this does not support bidirectional communication. Hence, Vivado needs to implement a route for either direction which is not efficient in terms of gates and is error prone. Instead, it was chosen to implement two interfaces for `'in'` and `'out'` respectively. The *isPrime* interface indicates if the the number is in fact prime, or not. The *PR module indicator* is a simple boolean that indicates to the static system which module is currently loaded. This makes sure that the static system will only communicate with the module if the correct one is loaded. The other interfaces are for correct communication between the static and dynamic region in order to properly time all events.

4.2.1 Final implementation

When applying the code from Appendix B as a PR project, the code can be fully synthesised and implemented. The total resource costs for this project are presented in Table 4.1. Since this is a simple example program, the resources take up a fraction of the Zedboard's available gates (see Section 2.5), but more than the AES core. The P-Block has been arbitrarily placed in the fabric, but does (as Vivado requires) span the height of an entire clock region.

An attempt was made to implement this project as a static design. Unfortunately, Vivado simplifies the design to 2 LUTs and 5 FFs, which is not suitable for this application.

Table 4.1: An overview of the resource requirements of the individual parts of the example design.

	LUT	FF
Static design	833	376
Prime verification module	325	33
Number generator module	366	60

Chapter 5

Evaluation

5.1 Prime verification project

Creating the example project provided a proper tutorial to understanding both the Vivado tooling, as well as partial reconfiguration. Using this it was easier to understand how PR worked and what difficulties would arise from working on projects with PR modules.

During the implementation of the example project, a few problems occurred, many of which were solvable. Except for the prime verification module. The applied method (which is inefficient, but this ensures larger amounts of logic) for determining primality requires the FPGA to divide the random number by all number up to the square root of the number itself. For a 32-bit number this requires at most $\sqrt{2^{32}} = 65536$ divisions. With this number of iterations on a for-loop, the design did not synthesise correctly. After trial and error, it was determined that using a 27-bit number is still eligible for prime verification with $\sqrt{2^{27}} \approx 11586$ iterations of the for-loop. This was implemented and the random number was adjusted in the final code too by discarding the top bits.

5.2 Routing analysis

The algorithms posed in Chapter 3 were applied to the AES core project. Initially this yielded no results as Vivado was able to format all routes within the partial reconfigurable module. An attempt was made to resize the P-block allocated for the module to force routes to go outside. Unfortunately, the block has a minimum size depending on the amount of logic it must fit. If the size is decreased further. With this limitation it was not possible to create routes that move out of the bounds.

To circumvent this problem, a different VHDL project was sourced. This AES core contained a little less logic, but could also not be forced to route outside the P-block. After more tinkering it was observed that a static implementation restricted to a specified area does create routes leaving that area, similar to the desired situation where routes leave the partial reconfigurable module[1]. This could be the starting point for the re-routing algorithm. For the prime verification project a static implementation proved troublesome as Vivado simplifies the design to 2 LUTs and 5 FFs which is not enough logic to perform any kind of analysis on.

Moving forward, the AES core is used for verification. With this project it was possible to apply the algorithm described in Chapter 3. Using the class from Listing 3.2 the design is imported and the appropriate functions are called. This piece of code requires the user to manually specify the corner tiles of the PR area and

[1]This also circumvents the need for the expensive Vivado partial reconfiguration license, as it is now possible to create it using the standard Vivado package and RapidSmith2.

Table 5.1: An overview of the result obtained on the AES core project.

	AES core
Correct routes	12490
Incorrect routes	241
Erroneous routes	0
Total routes	12731
Percentage correct	98.11%
Execute time	8.88 seconds

the file location of the RSCP checkpoint. These can differ depending on the use-case and between projects. With the correct names set, the AES core project was imported into RS2 and the code was run. The results are summarised in Table 5.1.

From the results in Table 5.1 it can be observed that there is a small number of routes that could not be rerouted. Since the available fabric is limited within the P-block, it can occur that some routes cannot be routed within the boundaries. Hence, they are not unrouted and the design will keep some imperfections. It can be seen that these routes make up 1.89% of the total routes. Figure 5.1 illustrates how there are still some nets fanning out from the P-block. There are still some red nets which correspond to the incorrect routes from Table 5.1.

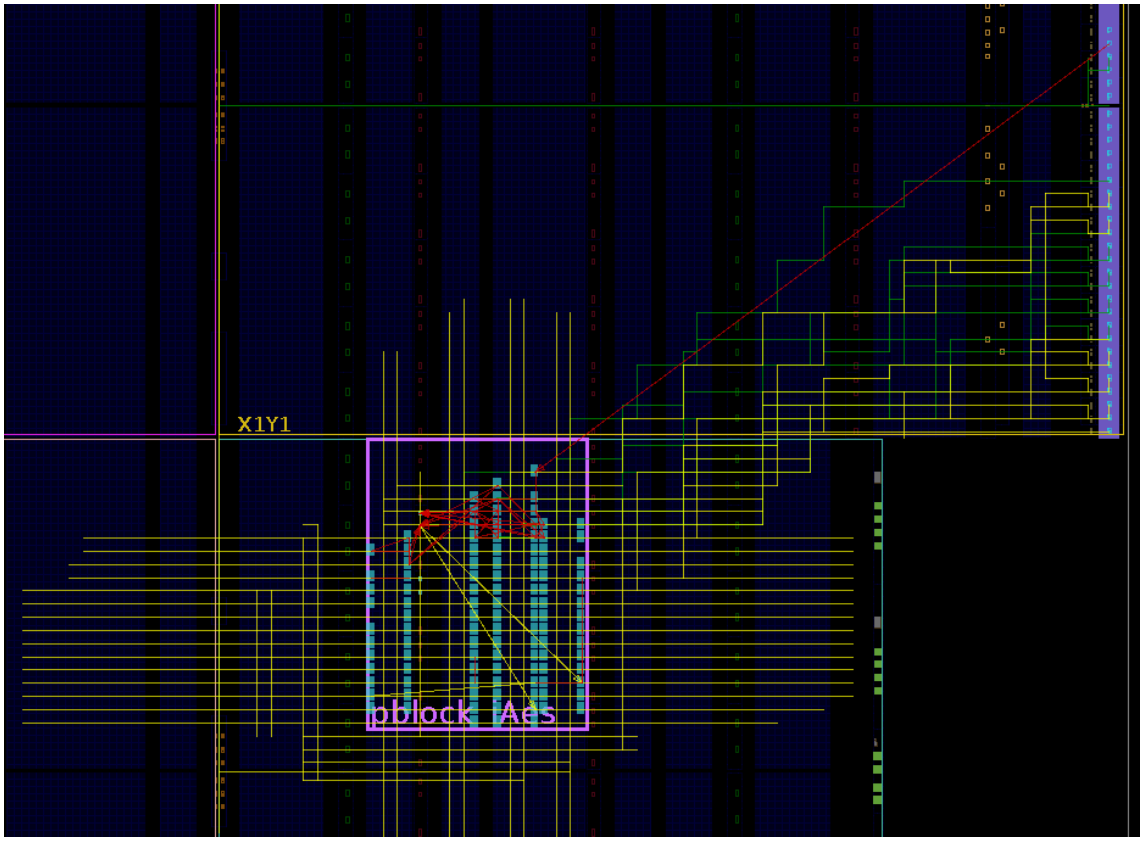


Figure 5.1: An overview of the corrected routing of the AES core project in Vivado.

Chapter 6

Discussion

This thesis required a thorough understanding of many different aspects regarding the FPGA. Aspects that are not part of the Electrical Engineering curriculum. Naturally, learning and understanding new material is integral part of a Bachelor Assignment. However, learning how to operate the software programs Vivado and RapidSmith2 required a significant portion of time which limited the progress that could be made on actual research.

In order to adapt routing on a very low level, RapidSmith2 was used. Since this program is currently still under heavy development, there are still some bugs. Some of which are solved by software updates, because RS2 is updated regularly. Unfortunately, some issues still remained, and getting RS2 compatible with the available Zedboard proved to be very troublesome.

A significant portion of time was spent on fixing the numerous problems that arose during installation of the Zedboard with RS2. After numerous attempts of creating the required configuration files, the program was downgraded to version 1.1.1. This version did not allow for the creation of the configuration files, but was able to recognise the Zedboard. All of this thanks to the help of Gerhard Mlady, who helped with solving many of the issues with RS2. Moreover, he was able to provide the required configuration files which I was not able to create myself through RS2. After numerous tests, it turns out that the Zedboard is still not fully supported by RS2 yet. Many of the sites are improperly named during file creation which results in error prompts when the file is imported. Solving this would require going through hundreds of pin names and replacing them with a name that is supported by RS2.

It is likely that problems such as these are solved in the future. For this research however, this meant the Zedboard could not be utilised for the creation of the Vivado project. This also eliminated the testing option of uploading the adjusted project to the Zedboard to verify if the operation is the same as before.

The choice for using the Zedboard was made as it was physically available. This allowed for the option to test the example project when initially making it, as well as after the re-routing process. The latter would be a feasible method to verify if the functionality indeed remains the same. Since it was not possible to use the Zedboard in RS2, an alternative was found in an Artix7 (xc7a100t-csg324) FPGA. This board was not available, but is supported by default in RS2. Therefore it is at least possible to apply the discussed algorithms and verify if they operate correctly. Using the Artix7 FPGA made the usage of RS2 trivial, but all testing has to be in the digital domain and cannot be produced on an actual FPGA board. Seeing that Gerhard Mlady is able to use the RS2 tooling together with the Zedboard which would suggest that it is possible to use it, when given more time.

When working on the routing algorithm in particular, many exceptions oc-

curred in the example code supplied by RS2. For example, the resizing of the `PriorityQueue` did not properly work. This meant that only a handful of routes could be rerouted before running into exception which voided the rest of the routes. Moreover, a problem occurred when trying to read the wire and connection of each `RouteTree` as some had no wires connected, so returned `null`. With this value, retrieving a connection type threw another exception.

These kind of errors forced me to analyse the existing routing algorithm in more detail than I initially hoped, which also took more time than initially intended. This did give me a proper insight into the structure of these functions and RS2. The work did lead to a working solution where all routes can evaluated by implementing a smarter requirement for when nets actually have to be rerouted. Due to limitations in the fabric, the A* router is not always able to find an alternative route, but this is in a minority of the cases at just 1.89%.

Chapter 7

Conclusion

This research has proven that it is feasible to utilise RS2 to analyse and adjust the routing. It was possible to apply the developed algorithm to the AES project for verification. The prime verification module was created to host enough logic to test applications created using RS2, which can be found to be true when comparing the resources of the two designs. It was found that routes only leave the allocated area if the design is implemented statically, whereas the routes remain inside a partial reconfigurable module. It was not possible to implement the prime verification project as a static design, so the AES core was used for all verification processes. Despite not being used for verification, the prime verification project has proven that the default Vivado routing for partial reconfigurable modules routes within the module itself, which is what is desired for these modules. The fact that routes remain within the partial reconfigurable modules proves hopeful for further development of PR projects including more advanced reconfiguration styles such as the slot based design.

The routing analysis is performed accurately, all incorrect routes (those that leave the partial reconfigurable module and do not route to outside the module) are found and can effectively be removed as is shown in Figure 3.2. Subsequently, an A* algorithm is employed to reroute the nets that have been unrouted. Due to the limited amount of logic within the specified area it is sometimes not possible for the simple A* algorithm to find a route that adheres to the requirements. As an effect, 1.89% of the routes cannot be properly re-routed. By reducing the number of incorrect routes the disadvantageous effects of the PR modules on the static fabric can be significantly reduced.

It can be found that all problems regarding the default RS2 have been solved as every single route can be processed and no errors occur. So, what can be taken from this report is the foundation for the analysis of PR projects using Vivado which provides a fully functional algorithm for determining incorrect nets and is able to re-route 98.11% of those routes on a small example application. It was found that despite the support for the Zynq family by RS2, the Zedboard is not yet fully supported. It has been proven that RS2 provides support for other families.

Chapter 8

Recommendations

In this work, the first steps towards alternative routing in RS2 are made. It must be noted that the methods here are not the most efficient or fastest methods and can most certainly be improved. A small portion of 1.89% of the routes cannot be properly rerouted. This number should be improved and most preferably be reduced to 0% to provide the desired outcome. The example A* algorithm, created by RS2 is known not to be the most efficient algorithm [8]. When improving the routing it is thus advised to implement a different algorithm with better performance that is also able to fit all routes within the required area.

Additionally, the currently implemented method requires more testing, preferably on an actual hardware device as was attempted with the Zedboard. This would verify if the routing procedure is actually effective and can directly show if the re-routed nets still deliver the same performance. When the design is imported into Vivado, it is also possible to perform a timing analysis and contrast it to the original design. This way, the effect of the re-routing can be evaluated. It would make sense that timing performance is affected since fitting routes within the partial reconfigurable module does not deliver the most time effective routes.

With the detection of out-of-bounds routes, the analysis of routing around PR modules is not complete. When analysing the routing within PR modules, it is also important to verify if each module uses the same routing towards the partition pins (or the older proxy logic cells). If these routes differ depending on the module, this could lead to differences in the performance. Also, it would be interesting to study the effect of shrinking the size of the modules and which effects this has on both the routes inside the module moving out-of-bounds, as well as the effect on these routes to the partition pins.

After these effect have been studied, a next step can be taken to research PR projects implementing slot based designs. Since they need routing between the slots, this will bring a more challenges for the routing. Proper analyis of the routes around these slots can be beneficial for the correct implementation of slot based designs.

Bibliography

[1] J. Angermeier, D. Ziener, M. Glaß, and J. Teich. Stress-aware module placement on reconfigurable devices. In *2011 21st International Conference on Field Programmable Logic and Applications*, pages 277–281, Sept 2011.

[2] Christian Beckhoff, Dirk Koch, and Jim Torresen. Go Ahead: A Partial Reconfiguration Framework. In *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*, pages 37–44, 5 2012.

[3] Mark Bourgeault. Altera's partial reconfiguration flow. *Technical Report*, 2011.

[4] Shih-Chun Chen and Yao-Wen Chang. Fpga placement and routing. In *Computer-Aided Design (ICCAD), 2017 IEEE/ACM International Conference*, pages 914–921, 13–16 November 2017.

[5] Nij Dorairaj, Eric Shiflet, and Mark Goosman. PlanAhead Software as a Platform for Partial Reconfiguration. *Xilinx XCELL Journal, Art*, 55:68–71, 2005.

[6] Jerzy Gbur. Aes_128_192_256. *Wroclaw University of Science and Technology*, May 2006.

[7] Travis Haroldsen, Brent Nelson, and Brad Hutchings. Rapidsmith 2: A framework for bel-level cad exploration on xilinx fpgas. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '15, pages 66–69, New York, NY, USA, 2015. ACM.

[8] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, July 1968.

[9] Xilinx Inc. Vivado® design suite, 2017.2.

[10] Dirk Koch. *Partial Reconfiguration on FPGAs: Architectures, Tools and Applications*, volume 153. Springer, 2012.

[11] Dirk Koch, Christian Beckhoff, and Jürgen Teich. ReCoBus-Builder - a Novel Tool and Technique to Build Statically and Dynamically Reconfigurable Systems for FPGAs. In *Proceedings of International Conference on Field-Programmable Logic and Applications (FPL 08)*, pages 119–124, Heidelberg, Germany, September 2008.

[12] Dirk Koch, Jim Torresen, Christian Beckhoff, Daniel Ziener, Christopher Dennl, Volker Breuer, Jürgen Teich, Michael Feilen, and Walter Stechele. Partial reconfiguration on fpgas in practice – tools and applications. In *Proceedings of the 2012 Architecture of Computing Systems (ARCS'12)*, pages 297–319, February 2012.

[13] C. Lavin, M. Padilla, P. Lundrigan, B. Nelson, and B. Hutchings. Rapid prototyping tools for fpga designs: Rapidsmith. In *Field-Programmable Technology (FPT), 2010 International Conference on*, pages 353–356. IEEE, 2010.

[14] Michael Mattioli. Driving the oled display on the zedboard. https://github.com/mmattioli/ZedBoard-OLED, May 2017.

[15] Brent Nelson, Thomas Townsend, and Travis Haroldsen. *RAPIDSMITH2 - A Library for Low-level Manipulation of Vivado Designs at the Cell/BEL Level - Technical Report and Documentation*. February 2018.

[16] Stan Ng. Binary to bcd. https://www.quora.com/How-do-I-convert-an-8-bit-binary-number-to-BCD-in-VHDL/answer/Stan-Ng-2, June 2017.

[17] Stephen M. Trimberger, Richard A. Carberry, Robert Anders Johnson, and Jennifer Wong. Method of time multiplexing a programmable logic device, Nov 2002. US 5646545A.

[18] Xilinx. *Floorplanning Methodology Guide*, page 9. Springer, May 2010.

[19] Xilinx. *Vivado Design Suite User Guide Partial Reconfiguration (UG909)*. April 2017.

[20] Xilinx. Zynq-7000 all programmable soc data sheet: Overview. https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf, June 2017.

Appendix A

Re-routing algorithm code

In Section 3.2, a description has been given of a number of functions. The code to realise the described behaviour is presented here.

Specifying tiles

```
1  public static Collection<Tile> selectingArea(Device device, Tile bottomLeft, Tile topRight) { //
   This function finds all the nets leaving the specified (PR) area
2          Collection<Tile> allTiles = device.getTiles();
3          Collection<Tile> selectedTiles = new ArrayList<>();
4
5          int[] coordinateBottomLeft =
           {bottomLeft.getTileXCoordinate(),bottomLeft.getTileYCoordinate()};
6          int[] coordinateTopRight =
           {topRight.getTileXCoordinate(),topRight.getTileYCoordinate()};
7
8          for(Tile tile : allTiles) { //Populate selectedTiles with all tiles within the boundaries
9                  int[] tempCoordinate = {tile.getTileXCoordinate(), tile.getTileYCoordinate()};
10                 if(tempCoordinate[0] <= coordinateTopRight[0] &&  //See if X-coordinate is
                   smaller that the topRight coordinate
11                     tempCoordinate[0] >= coordinateBottomLeft[0] && //See if X-coordinate
                       is larger that the bottomLeft coordinate
12                     tempCoordinate[1] <= coordinateTopRight[1] &&  //See if Y-coordinate is
                       smaller that the topRight coordinate
13                     tempCoordinate[1] >= coordinateBottomLeft[1]) { //See if Y-coordinate is
                       larger that the bottomLeft coordinate
14                         selectedTiles.add(tile);
15                 }
16         }
17         return selectedTiles;
18 }
```

Listing A.1: The code for specifying the tiles which are part of the region based on a bottom left and top right coordinate.

Specifying nets

```
1  public static Collection<CellNet> selectingNets(Collection<Tile> selectedTiles, CellDesign
   design) {
2          Collection<CellNet> selectedNets = new ArrayList<>(); //A list for nets originating
           from the selectedTiles
3          for (CellNet n : design.getNets()) { //Populate the selectedNets collection
4                  if(n.isSourced() && !n.isGNDNet() && !n.isVCCNet() && !n.isClkNet()) {
5                          for(SitePin p : n.getSourceSitePins()) {
6                                  if(selectedTiles.contains(p.getSite().getTile()))
7                                          selectedNets.add(n);
8                          }
9                  }
10         }
```

```
11          return selectedNets;
12  }
```

Listing A.2: The code for specifying which nets originate from the collection of tiles derived in Listing A.1.

Adapted A* routing algorithm

```
1   public Results routeNet(Collection<Tile> areaTiles, Collection<CellNet> areaNets,
    Iterator<SitePin> sinksToRoute, CellNet net) {
2           // Initialize the route
3           RouteTree start = initializeRoute(net); //The routetree to be returned
4           Set<RouteTree> terminals = new HashSet<>();
5
6           //Initialize return object
7           Results results = new Results();
8           results.routeTree = new RouteTree(null);
9           results.reroutedRoutes = 0;
10          results.correctRoutes = 0;
11          results.wrongRoutes = 0;
12          results.errorRoutes = 0;
13
14          assert sinksToRoute.hasNext() : "There are no CellNet objects to be re-routed for this
            net. This means the detection of incorrect routes went wrong!";
15
16          // Iterate over each sink SitePin in the net, and find a valid route to it.
17          SKIP_SINK :
18          while(sinksToRoute.hasNext()) {
19                  // initialize the target wire, and priority queue
20                  SitePin sink = sinksToRoute.next();
21                  Wire targetWire = getTargetSinkWire(sink);
22                  targetTile = targetWire.getTile();
23                  resortPriorityQueue(start);
24
25                  // Notify user which pin we are working on
26                  System.out.println("\t\tActually re-routing sink: " + sink);
27
28                  // Update the number of routes being worked on for the statistics
29                  results.reroutedRoutes++;
30
31                  // Variables for while-loop
32                  boolean routeFound = false;
33                  int count = 0;
34
35                  while (!routeFound) { //This loop actually builds the routing data structure
36                          count++;
37                          if(count > 50000) { // max = 1000000
38                                  System.out.println("\t\t\tCould not find an alternative route,
                                    so skipping this!");
39                                  results.wrongRoutes++;
40                                  break SKIP_SINK;
41                          }
42                          // Grab the lowest cost route from the queue
43                          RouteTree current = priorityQueue.poll();
44
45                          // Get a set of sink wires from the current RouteTree that already exist
                            in the queue
```

```
46                          Set<Wire> existingBranches =
                            usedConnectionMap.getOrDefault(current, new HashSet<Wire>());
47
48                          // Search all connections for the wire of the current RouteTree
49                          try { // In case the wire does not have a connection, the exception is
                            caught
50                                  for (Connection connection :
                                    current.getWire().getWireConnections()) {
51                                          Wire sinkWire = connection.getSinkWire();
52
53                                          // Solution has been found
54                                          if (sinkWire.equals(targetWire)) {
55                                                  RouteTree sinkTree =
                                                    current.addConnection(connection);
56                                                  sinkTree = finializeRoute(sinkTree);
57                                                  terminals.add(sinkTree);
58                                                  routeFound = true;
59                                                  results.correctRoutes++;
60                                                  break;
61                                          }
62                                          // Only create and add a new RouteTree object if it
                                            doesn't already exist in the queue
63                                          if (!existingBranches.contains(sinkWire) &&
                                            areaTiles.contains(sinkWire.getTile())) {
64                                                  RouteTree sinkTree =
                                                    current.addConnection(connection);
65                                                  sinkTree.setCost(current.getCost() + 1);
66                                                  priorityQueue.add(sinkTree);
67                                                  existingBranches.add(sinkWire);
68                                          }
69                                  }
70                          }
71                          catch(NullPointerException ex) {
72                                  System.out.println("\t\t\tCould not find wire connected to
                                    this pin!");
73                                  results.errorRoutes++;
74                                  break SKIP_SINK;
75                          }
76                          usedConnectionMap.put(current, existingBranches);
77                  }
78                  // prune RouteTree objects not used in the final solution. This is not very
                    efficient...
79                  start.prune(terminals);
80                  results.routeTree = start;
81          }
82          return results;
83  }
```

Listing A.3: The code for re-routing the incorrect nets based on the original A* code by RapidSmith2 [13] (this listing shows only the altered function `routeNet`)

Appendix B

VHDL code

As described in Chapter 4, a sample VHDL code is required to work with. The source code of both the static system as well as the two PR modules are stated here. It must be noted that the prime verification algorithm is most certainly not the best or most efficient algorithm, but for the sake of this example it is more than sufficient.

For the creation of a random 32-bit number, a long (8000 bits) `std_logic_vector` with random bits is used. An index increases every clock cycle. Because of the random timing of requesting a number, the index will have a 'random' value. Using the index, a 32 bits can be retrieved from the `std_logic_vector`. When the index is within the last 32 bits of the first 32 bits of the `std_logic_vector` will be returned. The possibility that this happens is $\frac{1}{250}$ so reasonably small. The final implementation can be found in Listing B.2.

For the conversion of this 32-bit number to a format that can be printed on the OLED on the Zedboard (BCD), the solution by Stan Ng was used [16]. This is implemented as a function in the static design (Listing B.1).

To print the number on the Zedboard's OLED screen; the Digitlent OLED library modified for Zedboard by Michael Mattioli was used [14]. It must be noted that such an OLED screen is not available for the Artix7 board which was used as an alternative to the Zedboard.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4
5  entity Zedboard is
6      Port (
7              clk : in STD_LOGIC;
8              reset : in STD_LOGIC;
9              nextState : in STD_LOGIC;
10             displayLed : out STD_LOGIC;
11             primeLed : out STD_LOGIC;
12             generateLed : out STD_LOGIC;
13             LD7 : out std_logic;
14             oled_sdin : out std_logic;
15             oled_sclk : out std_logic;
16             oled_dc : out std_logic;
17             oled_res : out std_logic;
18             oled_vbat : out std_logic;
19             oled_vdd : out std_logic
20         );
21 end Zedboard;
22
23 architecture Behavioral of Zedboard is
24
25 FUNCTION number2bcd (in_binary:UNSIGNED(31 DOWNTO 0)) RETURN std_logic_vector IS
26
27 variable concatVector : std_logic_vector (39 downto 0) := (others => '0');
```

```
28
29  variable s_digit_0 : unsigned( 3 downto 0):= "0000";
30  variable s_digit_1 : unsigned( 3 downto 0):= "0000";
31  variable s_digit_2 : unsigned( 3 downto 0):= "0000";
32  variable s_digit_3 : unsigned( 3 downto 0):= "0000";
33  variable s_digit_4 : unsigned( 3 downto 0):= "0000";
34  variable s_digit_5 : unsigned( 3 downto 0):= "0000";
35  variable s_digit_6 : unsigned( 3 downto 0):= "0000";
36  variable s_digit_7 : unsigned( 3 downto 0):= "0000";
37  variable s_digit_8 : unsigned( 3 downto 0):= "0000";
38  variable s_digit_9 : unsigned( 3 downto 0):= "0000";
39
40  BEGIN
41          for i in 31 downto 0 loop
42                  if (s_digit_9 >= 5) then s_digit_9 := s_digit_9 + 3; end if;
43                  if (s_digit_8 >= 5) then s_digit_8 := s_digit_8 + 3; end if;
44                  if (s_digit_7 >= 5) then s_digit_7 := s_digit_7 + 3; end if;
45                  if (s_digit_6 >= 5) then s_digit_6 := s_digit_6 + 3; end if;
46                  if (s_digit_5 >= 5) then s_digit_5 := s_digit_5 + 3; end if;
47                  if (s_digit_4 >= 5) then s_digit_4 := s_digit_4 + 3; end if;
48                  if (s_digit_3 >= 5) then s_digit_3 := s_digit_3 + 3; end if;
49                  if (s_digit_2 >= 5) then s_digit_2 := s_digit_2 + 3; end if;
50                  if (s_digit_1 >= 5) then s_digit_1 := s_digit_1 + 3; end if;
51                  if (s_digit_0 >= 5) then s_digit_0 := s_digit_0 + 3; end if;
52                  s_digit_9 := s_digit_9 sll 1; s_digit_9(0) := s_digit_8(3);
53                  s_digit_8 := s_digit_8 sll 1; s_digit_8(0) := s_digit_7(3);
54                  s_digit_7 := s_digit_7 sll 1; s_digit_7(0) := s_digit_6(3);
55                  s_digit_6 := s_digit_6 sll 1; s_digit_6(0) := s_digit_5(3);
56                  s_digit_5 := s_digit_5 sll 1; s_digit_5(0) := s_digit_4(3);
57                  s_digit_4 := s_digit_4 sll 1; s_digit_4(0) := s_digit_3(3);
58                  s_digit_3 := s_digit_3 sll 1; s_digit_3(0) := s_digit_2(3);
59                  s_digit_2 := s_digit_2 sll 1; s_digit_2(0) := s_digit_1(3);
60                  s_digit_1 := s_digit_1 sll 1; s_digit_1(0) := s_digit_0(3);
61                  s_digit_0 := s_digit_0 sll 1; s_digit_0(0) := in_binary(i);
62          end loop;
63
64          concatVector(3 downto 0) := std_logic_vector(s_digit_0);
65          concatVector(7 downto 4) := std_logic_vector(s_digit_1);
66          concatVector(11 downto 8) := std_logic_vector(s_digit_2);
67          concatVector(15 downto 12) := std_logic_vector(s_digit_3);
68          concatVector(19 downto 16) := std_logic_vector(s_digit_4);
69          concatVector(23 downto 20) := std_logic_vector(s_digit_5);
70          concatVector(27 downto 24) := std_logic_vector(s_digit_6);
71          concatVector(31 downto 28) := std_logic_vector(s_digit_7);
72          concatVector(35 downto 32) := std_logic_vector(s_digit_8);
73          concatVector(39 downto 36) := std_logic_vector(s_digit_9);
74
75          RETURN concatVector;
76  END number2bcd;
77
78      -- Component declarations for OLED
79      component oled_init is
80          port ( clk : in std_logic;
81                  rst : in std_logic;
82                  en : in std_logic;
83                  sdout : out std_logic;
84                  oled_sclk : out std_logic;
85                  oled_dc : out std_logic;
```

```
86                  oled_res : out std_logic;
87                  oled_vbat : out std_logic;
88                  oled_vdd : out std_logic;
89                  fin : out std_logic);
90      end component;
91
92      component oled_ex is
93          port ( clk : in std_logic;
94                  rst : in std_logic;
95                  en : in std_logic;
96                  bcd : in std_logic_vector(39 downto 0);
97                  sdout : out std_logic;
98                  oled_sclk : out std_logic;
99                  oled_dc : out std_logic;
100                 fin : out std_logic);
101     end component;
102
103     component rModule_primegenerate is
104         port( clk : in STD_LOGIC;
105                 prime : out STD_LOGIC;
106                 notificationIn : in STD_LOGIC;
107                 notificationOut : out STD_LOGIC;
108                 nIn : in UNSIGNED(31 downto 0);
109                 nOut : out UNSIGNED(31 downto 0);
110                 prModule : out STD_LOGIC);
111     end component;
112
113     -- OLED FSM
114     type states is (Idle, OledInitialize, OledExample, Done);
115
116     -- OLED variables
117     signal current_state : states := Idle;
118
119     signal init_en : std_logic := '0';
120     signal init_done : std_logic;
121     signal init_sdata : std_logic := '0';
122     signal init_spi_clk : std_logic;
123     signal init_dc : std_logic := '0';
124
125     signal example_en : std_logic := '0';
126     signal example_sdata : std_logic := '0';
127     signal example_spi_clk : std_logic;
128     signal example_dc : std_logic;
129     signal example_done : std_logic := '0';
130
131     -- Static design FSM
132     type control is (createRandomNumber, checkPrime, displayData); --Define FSM
133
134     -- Signal containing the desired bcd codes
135     signal bcd : STD_LOGIC_VECTOR(39 downto 0) := x"0000000000";
136
137     signal notificationIn : STD_LOGIC := '0';
138     signal notificationOut : STD_LOGIC := '0';
139     signal prModule : STD_LOGIC;
140     signal numberIn : UNSIGNED(31 downto 0) := (others => '0');
141     signal numberOut : UNSIGNED(31 downto 0) := (others => '0');
142     signal prime : STD_LOGIC := '0';
143
```

```
144  begin
145
146      -- OLED portmaps
147      Initialize: oled_init port map (clk => clk,
148                                      rst => reset,
149                                      en => init_en,
150                                      sdout => init_sdata,
151                                      oled_sclk => init_spi_clk,
152                                      oled_dc => init_dc,
153                                      oled_res => oled_res,
154                                      oled_vbat => oled_vbat,
155                                      oled_vdd => oled_vdd,
156                                      fin => init_done
157                                     );
158
159      Example: oled_ex port map ( clk => clk,
160                                  rst => reset,
161                                  en => example_en,
162                                  bcd => bcd,
163                                  sdout => example_sdata,
164                                  oled_sclk => example_spi_clk,
165                                  oled_dc => example_dc,
166                                  fin => example_done
167                                 );
168
169   verifyPrimePR : rModule_primegenerate port map (
170                clk => clk,
171                prime => prime,
172                notificationOut => notificationIn,
173                notificationIn => notificationOut,
174                nOut => numberIn,
175                nIn => numberOut,
176                prModule => prModule
177           );
178
179      -- MUXes to indicate which outputs are routed out depending on which block is enabled
180      oled_sdin <= init_sdata when current_state = OledInitialize else example_sdata;
181      oled_sclk <= init_spi_clk when current_state = OledInitialize else example_spi_clk;
182      oled_dc <= init_dc when current_state = OledInitialize else example_dc;
183      -- End output MUXes
184
185      -- MUXes that enable blocks when in the proper states
186      init_en <= '1' when current_state = OledInitialize else '0';
187      example_en <= '1' when current_state = OledExample else '0';
188      -- End enable MUXes
189
190  process(clk)
191
192  variable random : UNSIGNED(31 downto 0) := numberIn;
193  variable state : control := createRandomNumber; --Create FSM and default at start (creating a
     random number)
194  variable primeLocal : STD_LOGIC := '0';
195  variable temp : unsigned(31 downto 0) := x"0012D687";
196
197  begin
198
199  if rising_edge(clk) then
200
```

```
201     if reset = '1' then
202         current_state <= Idle;
203     elsif nextState = '1' then
204         current_state <= OledExample;
205     else
206         case current_state is
207             when Idle =>
208                 current_state <= OledInitialize;
209             -- Go through the initialization sequence
210             when OledInitialize =>
211                 if init_done = '1' then
212                     current_state <= OledExample;
213                 end if;
214             -- Do example and do nothing when finished
215             when OledExample =>
216                 if example_done = '1' then
217                     current_state <= OledExample;
218                 end if;
219             -- Do nthing
220             when Done =>
221                 current_state <= Done;
222             when others =>
223                 current_state <= Idle;
224         end case;
225     end if;
226
227     case state is
228         when createRandomNumber => --Send signal to randomNumberGenerator module
            to create a new number.
229             generateLed <= '1';
230             primeLed <= '0';
231             displayLed <= '0';
232             LD7 <= '0';
233
234             if prModule = '0' then -- Only run this code when the correct module is loaded
235                 notificationOut <= '1';
236                 if notificationIn = '1' then
237                     notificationOut <= '0';
238                     random := numberIn;
239
240                     -- Display the number
241                     bcd <= number2bcd(random);
242                     current_state <= OledExample;
243
244                     -- Random number received, move to next step
245                     state := checkPrime;
246                 end if;
247             end if;
248
249         when checkPrime => --Send the previously created number to the checkPrime
            module to see if number is prime.
250             generateLed <= '0';
251             primeLed <= '1';
252             displayLed <= '0';
253             LD7 <= '0';
254
255             if prModule = '1' then -- Only run this code when the correct module is loaded
256
```

```
257                     numberOut <= random; -- Send the number to the prime module
258                     notificationOut <= '1';
259                     if notificationIn = '1' then
260                         if prime = '1' then
261                             primeLocal := '1';
262                         else
263                             primeLocal := '0';
264                         end if;
265                         notificationOut <= '0';
266                         state := displayData; -- All data is available --> print it
267                     end if;
268 
269                 end if;
270 
271             when displayData => --Display both the number and if it is a prime or not. (Can be
                run regardless of module)
272                 generateLed <= '0';
273                 primeLed <= '0';
274                 displayLed <= '1';
275 
276                 -- Display the number
277                 bcd <= number2bcd(random);
278                 current_state <= OledExample;
279 
280                 --Set led[7] to the right state to match prime y/n
281                 LD7 <= prime;
282 
283                 -- Only continue after button press
284                 if reset = '1' then
285                     random := (others => '0');
286                     primeLocal := '0';
287                     state := createRandomNumber; --Do the entire cycle again
288                 end if;
289         end case;
290     end if;
291     end process;
292 
293     end Behavioral;
```

Listing B.1: The VHDL code used for the static part of the PR system.

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3   use ieee.numeric_std.all;
4 
5   entity rModule_primegenerate is
6 
7   port (
8             clk:   in std_logic;
9             prime:   out std_logic;
10            notificationIn:  in std_logic;
11            notificationOut: out std_logic;
12            nOut:   out UNSIGNED(31 downto 0);
13            nIn:    in UNSIGNED(31 downto 0);
14            prModule :   out std_logic
15    );
16 
17  end entity;
18 
```

```
19  architecture behavioral of rModule_primegenerate is
20
21      constant bits : std_logic_vector(7999 downto 0) := x"[long array of hexadecimal bits here]";
22      constant bitsLength : integer := 8000;
23      signal arrayIndex : integer := 0;
24      signal number : std_logic_vector(31 downto 0) := (others => '0');
25
26  begin
27      -- Drive output signal.
28  -- nOut <= unsigned(number);
29      nOut <= unsigned(number);
30      prModule <= '0';
31      prime <= '0';
32
33      -- Synchronous process.
34      process (clk) is
35      begin
36          if rising_edge(clk) then
37              notificationOut <= '0';
38              arrayIndex <= arrayIndex + 1;
39              if arrayIndex = bitsLength then
40                  arrayIndex <= 0;
41              end if;
42          if notificationIn = '1' then -- If a number is requested, return 32 bits starting from
            arrayIndex
43              if arrayIndex >= bitsLength - 27 then
44                              number(26 downto 0) <= bits(26 downto 0);
45                  else
46                  number(26 downto 0) <= bits(arrayIndex + 26 downto arrayIndex);
47              end if;
48
49              notificationOut <= '1';
50              end if;
51          end if; --End clock
52      end process;
53  end behavioral;
```

Listing B.2: The VHDL code for the module that generates 32-bit random numbers.

```
1   library IEEE;
2   use IEEE.STD_LOGIC_1164.ALL;
3   use IEEE.NUMERIC_STD.ALL;
4
5   entity rModule_primegenerate is
6       port(  clk : in STD_LOGIC;
7                      prime : out STD_LOGIC;
8        notificationIn : in STD_LOGIC;
9       notificationOut : out STD_LOGIC;
10                              nIn : in UNSIGNED(31 downto 0);
11                             nOut : out UNSIGNED(31 downto 0); --Dummy interface
12                    prModule : out STD_LOGIC
13          );
14  end rModule_primegenerate;
15
16  architecture Behavioral of rModule_primegenerate is
17
18  signal number : integer := 0;
```

```
19
20  begin
21
22  nOut <= (others => '0');
23  prModule <= '1';
24
25  process(clk)
26  begin
27
28      if rising_edge(clk) then
29          if (notificationIn = '1') then
30              number <= to_integer(nIn);
31
32              if number = 1 then
33                      prime <= '0';
34              else
35                  for i in 2 to 11586 loop --46341 is sqrt of 2^31
36                      if number mod i=0 then
37                          prime <= '0';
38                      else
39                          prime <= '1';
40                      end if;
41                  end loop;
42              end if;
43              notificationOut <= '1';
44          end if;
45      end if; -- End clock
46  end process;
47
48  end Behavioral;
```

Listing B.3: The VHDL code for the prime verification module.