Placement with Irregularities

Mini Task Report

Florian Prott and Julian Käuser
Supervisor: Jakob Wenzel M.Sc.
Submission: 09/25/2017
Institute for Computer Engineering | Computer Systems Group | Prof. Dr.-Ing. Christian Hochberger

1 Introduction

Routing and Placement are two necessary and time-consuming steps in the FPGA synthesis toolchain. Optimizations for module design in these steps speed up or enhance the FPGA design process significantly. RapidSmith[3] allows the implementation of these steps in a Java framework. It offers an API to read in Xilinx designs and FPGA descriptions, and offers researchers the possibility to approach placement and routing problems in an object-oriented way. In RapidSoC[4], a project based on RapidSmith, a re-placement method for already placed and routed modules is integrated. Although this method works for a majority of modules, some modules routing elements cannot be re-placed due to irregular structures in the FPGA fabric. In this project, the cause for these errors is examined, and an approach to fix them is introduced.

The report is structured as following. First, the exact problem description is presented. The designed approach to the problem using isoelectric planes is described. Then, the insights gained are presented. Finally, future work is proposed, and the report is concluded.

1.1 Task Description

Since the fabric of FPGAs is in general regular, the expected error source for designs at different locations are irregularities regarding the routing elements in the fabric. In the current state, the nets of the design have to be re-routed, which is time-consuming. Since we guess that only 1-2 programmable interconnect points ("PIPs") are set falsely, a detection and repair of the conflicting nets promises faster success than a new routing. The task for this project is to determine a method which finds places where the described error occurs and is able to include the missing PIPs into the net. The proposed method makes use of the interconnect description features of RapidSmith, which operate on a mask of the FPGA fabric.

1.2 Framework and Environment

This project is embedded into the RapidSmith Java environment, version a.b.c. Classes of the RapidSoC system, especially the class `MoveModulesEverywhere`, are the basis of our work. `MoveModulesEverywhere` parses existing designs from `.xdl` files into the RapidSmith environment, places the included modules on any feasible destination, and generates the corresponding `.xdl` output files. Further processing is accomplished with Xilinx ISE [1]. Based on the output of the ISE tool `xdl`, the re-placed designs can be evaluated to be functional or conflicting.

All processing developed in this project is performed in a separate Java package. Every re-placed design is extracted and examined before the output step, so that the developed methods can be applied to a RapidSmith `Design` object.

1.2.1 Naming Conventions

Throughout this report, multiple references to nets, pins, PIPs, tiles and nodes are made. These describe the physical objects on the FPGA. `Net, Pin, PIP, Tile` and `Node` describe their object representation in RapidSmith, which is mostly equally named as the physical object.

2 General Approach to the Problem

In this chapter, our approach to find and repair conflicting spots in the netlists is described.

2.1 Routing Elements in RapidSmith

Since the RapidSmith API define multiple elements involved in the routing process, a short overview over these elements is given in the following list:

- **Wire**
  A Wire is the basic connection element. It is represented as an integer due to the large number of wires on an FPGA. Wires can be hard connected to other wires, which can be determined through the `Device` class.

- **Pin**
  A Pin connects an input or output of a logic block to routing resources. It is primarily connected to one `wire`, and may be defined as output or input pin.

- **PIP**
  A PIP object describes an active connection between two wires, which are attributes to the object. If it is contained in a `net`, is is switched on. Otherwise, it remains turned off.

- **Net**
  A Net gathers a list of connected pins and active PIPs. It also has a defined source pin driving the net.

- **WireConnection**
  A WireConnection is a wrapper class for a `wire` and holds the information whether this wire is only reacheable if a PIP is turned on. This class is used to get information about the connectable wires on the FPGA.

- **Node**
  A Node is a routing object used in the router package. It can be used to describe routes of `wires` and `PIPs` in a router.

- **SinkPin**
  A SinkPin refers to a switch matrix on the FPGA. Nearly all pins' wires are connected to a switch matrix, which is an interface to the other wires. SinkPins have to be viewed if a connected pin of a wire shall be found.

- **Tile**
  A Tile is one of the checkerboard-like distributed areas on an FPGA. It holds various elements from the list above and a set of logic elements.

2.2 Isoelectric Potential Search

Each design's routing in RapidSmith can be viewed as a set of `Net` objects. According to the RapidSmith documentation [2], these `Net` objects hold all used pins and pips of the net. Therefore it should be possible to reach all sink pins of a net by following wires connected to the input pin if the net is not broken. Consequently each design is broken if it is not possible to reach the output pins of an net from the input pin.

A `Net` in RapidSmith only contains `PIP` objects representing PIPs that are currently "switched on", but no `Wires`. Therefore, a net can be expanded if additional PIPs are switched on. It is theorized that it is possible to fix the broken net by switching on PIPs which reconnect the net's pins' connected wires with each other. This can only be done when the PIP will not connect the circuit to an independent third circuit.

To check whether or not a net can be traversed from it's source pin to all output pins an new measurement called `Potential` is introduced into RapidSmith. A `Potential` can be seen as the isoelectric set of wires, pins and PIPs which are connected, starting from one pin. It is never possible for a `Pin`, `PIP` or `Wire` to be in two `Potentials` at the same time. It is not possible that two non-connected elements have the same `Potential`. This concept is named after the concept of isoelectric potentials found in classical electronics. Although two electrical potentials may be at the same numeric voltage level and still be different (not isoelectric), this is not the case for the introduced potentials.

Before any further operation on a `Potential` may be performed, its spatial spread must be computed. This means that, starting from a pin, each electrically connected wire, pin and pip must be found. The search for connected elements works as described in algorithm **??**. It is also pointed graphically in Figure 2.1. Given this method, a `Potential` derived for one pin must always hold any other pin in the net if the net is routed correctly.

`Potentials` can be fused by setting PIPs. Electrically, this means that two isoelectric sets of elements are connected by a switch, and thus become one isoelectric potential. For the object representation this means that, if a PIP is set, the two `Potential` objects have to be united.

The concept of isoelectric potentials is chosen because the search for adjacent pips wire by wire is simplified this way.

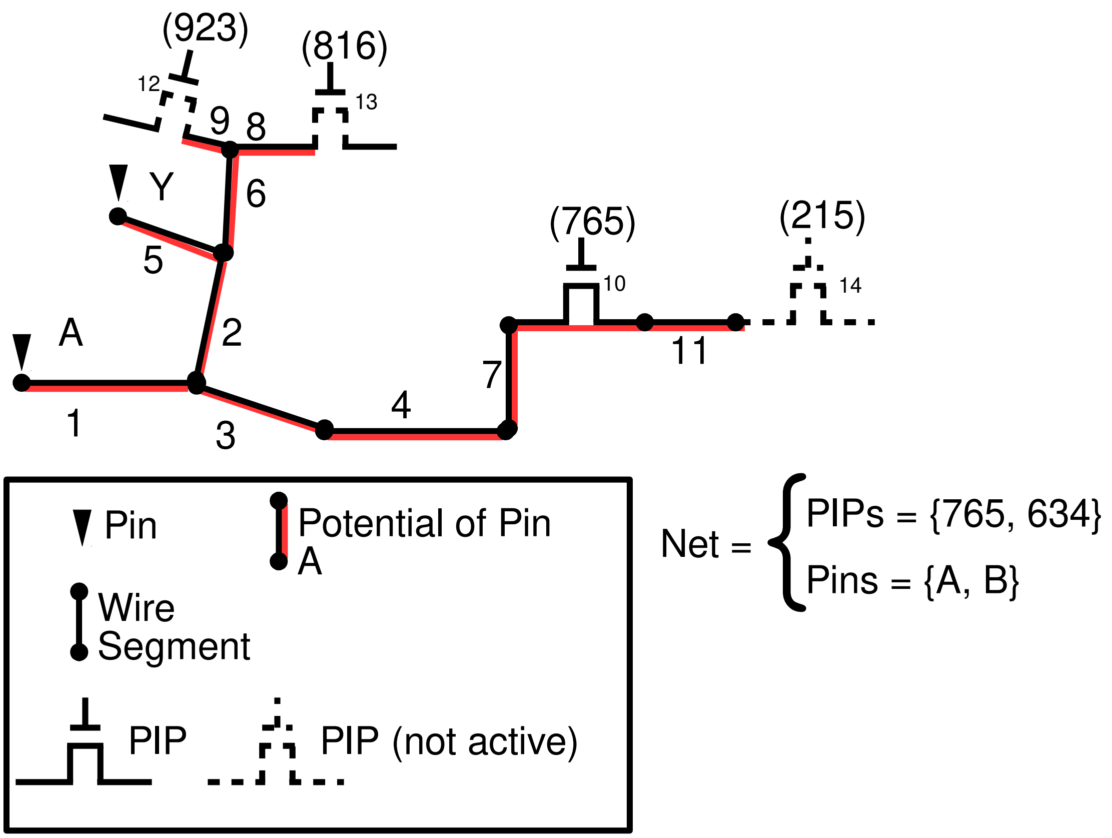

Figure 2.1: Graphical visualization of the `Potential` derivation. The wire segments and elements are enumerated in the sequence of integration. PIP IDs are in parentheses.

```
wires = {wireOf(sourcePin)};
pips = {};
pins = {sourcePin};
adjacentPIPs = {};
while new elements added do
    foreach wire i do
        foreach reacheableWire(i) do
            if !reacheable.isPIP() then
                add reacheableWire to wires;
            end
            else if pip ∈ net.pips() then
                add reacheableWire to wires;
                add pip to pips;
            end
            else
                add pip to adjacentPIPs;
            end
        end
    end
end
foreach wire i do
    add connected pin to pins;
end
```

Algorithm 1: Algorithm to determine all elements on one isoelectric potential.

2.3 Finding and Reparing broken Nets

In order to check if an net is broken the potential of each pin is calculated and compared. If the net contains more than one unique potential then the net is broken. In that case it is necessary to reconnect the two parts (potentials) of that net. If the assumption that only one or two PIPs are missing is true, this can be done by the applying the following breadth-first-search on the `Potentials` of pins A and B:

There is a priority queue *leaves* which holds only PIP-elements that do not connect to other `Potentials` besides B. Each of these elements has a parent PIP which introduced the PIP into the queue. *Leaves* is ordered by the number of parents between the PIP and Potential A in ascending order. Algorithm 2 points out the search on that priority queue.

Figure 2.2 shows the case for one non-set PIP in a net. The net specifies two pins, A and B. For both pins, the `Potential` has been determined. Clearly marked are the included wires, PIPs and other pins. PIP 215 is adjacent to both potentials. The breadth-first-search finds PIP 215 to be adjacent to both pins' potentials, so this PIP has to be set (a `PIP` instance is created for the connected `wires` and added to the net). For situations where more than one PIP has to be set, the breadth-first-search steps one level down the search tree and repeats the search on every leaf. By going one level deeper, the first PIP in the priority queue has to be set.

As one can see, the breadth-first search runtime highly depends on the number of missing PIPs in the net. For every missing PIP, the number of elements to search is multiplied by the amount of adjacent PIPs to the Potential.

add all adjacent PIPs of A to leaves;
while *leaves.size() > 0* **do**
  PIP pip = leaves.pop();
  if *Potential B contains pip* **then**
    return pip; // we can determine the connection trough the parent PIPs;
  end
  foreach *PIP p* $\in pip.getAdjacentPIPs$ **do**
    if *p!=pip.parent* **then**
      leaves.push(p); //if p is of an other potential than B we cannot add it ;
    end
  end
end

Algorithm 2: Exemplary algorithm to determine missing PIPs by breadth-first-search

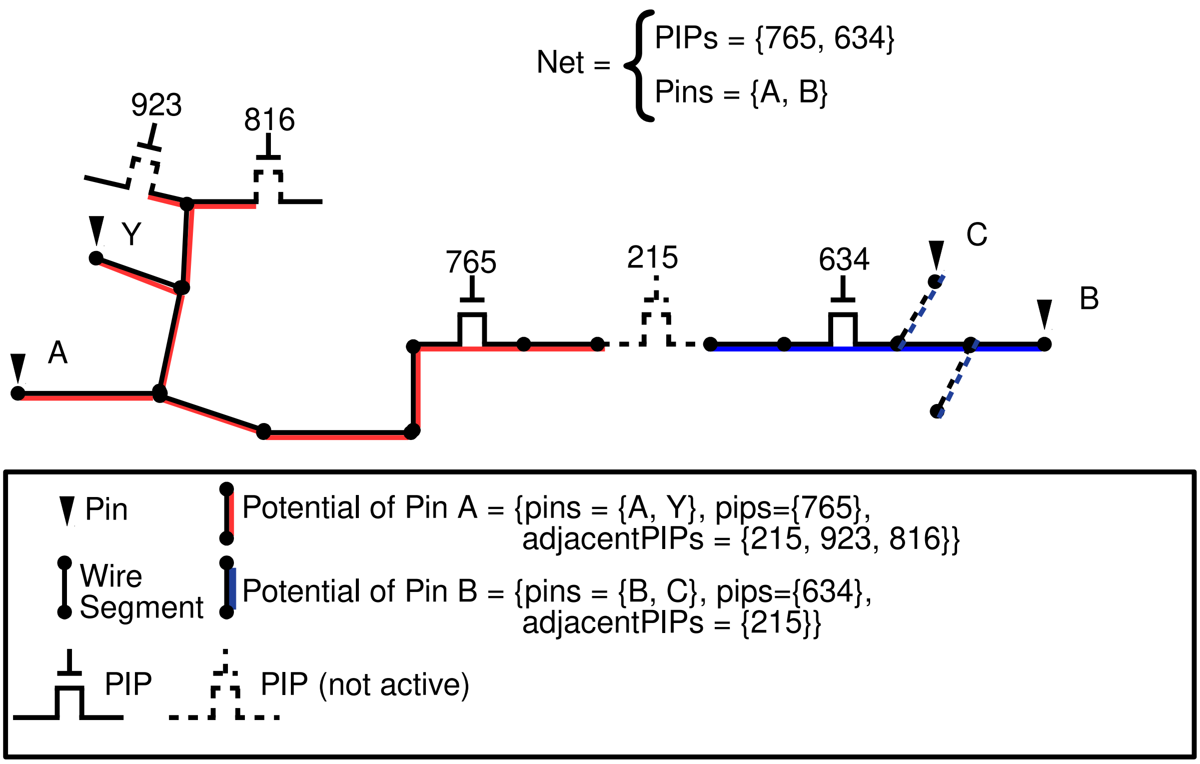

Figure 2.2: A net with two `Potentials` which have to be connected.

3 Insights

Although the approach described in chapter 2 seems promising, testing it for various example nets (conflicting and functional in terms of the ISE `xdl` output) did not lead to the expected results. Therefore, we examined the reasons, implementation and other possible error sources.

3.1 Hand Routing

The RapidSmith framework has some built-in example classes, which are designed to help the user understand the way RapidSmith works. One of these classes is the `HandRouter`, which allows the routing of a net through a console. It displays reacheable wires at the next iteration and adds PIPs as the route is chosen. We used and adapted the `HandRouter` to manually examine broken nets. Working on complete `.xdl`-files only, we adapted the router to work on single nets, and inserted it after the potential derivation.

Using the HandRouter reveals several error sources, but also that each connection reacheable from each pin is correctly integrated in the pin's `Potential` instance. Therefore, we could prove that, based on the given nets, the concept of isoelectric potentials is implemented correctly. Nevertheless, the errors still occurred.

Further analysis of the reacheable routing elements from every pin revealed that the assumption of only 1 or two missing PIPs between the isoelectric potentials of the sink and source pin evaluated to be wrong. In the most examined cases, at least four or more PIPs not specified by the net have to be additionally switched on. While some of the wire segments where additional PIPs are necessary only feature one or two switcheable PIPs, others had a much longer list.

Therefore, we decided not to apply the breadth-first-search and propose to perform a re-routing of the net. Any router has to solve the problem of too many possible connections on the route to take. The breadth-first-search is suitable for search trees of one or two levels, but it is probably bad compared to a distinct router able of finding routes with a larger number of PIPs.

3.2 Possible Reasons for Errors

The number of missing PIPs leads us to the assumption that the errors in re-placed designs or nets are not completely caused by small irregularities in the FPGA fabric. RapidSmith features a coordinate-based location description of nearly all elements of the FPGA. While this is useful for the majority of applications, some details may be hidden in this abstraction. Not every cartesian coordinate perfectly translates into the actual position of the element on the FPGA, so that regular structures in terms of the cartesian coordinates do not necessarily imply regular structures in the fabric. Hence, the assumption of conflicting re-placed nets caused by slight irregularities could not be proven.

In addition to the larger-than-expected irregularities, there are other error sources in RapidSmith. We cannot determine the exact source, but we have to assume that certain information on the FPGA are not implemented correctly. This assumption is made due to the fact even for functional nets (in terms of the `xdl` conversion), some of the corresponding RapidSmith `Nets` were not routed correctly (i.e. the source was reacheable from all sink pins).

3.3 Outcome

Within this project, a method to find missing PIPs in re-placed FPGA designs is developed. Its working principle is based on a breadth-first-search. According to the assumption that only very few PIPs are missing in the nets because of slight irregularities in the FPGA fabric, a breadth-first search is applicable.

Reviewing the results proves the assumption of only few missing PIPs wrong. The analysis of error sources can be done with the built-in tools of RapidSmith. We suggest the use of a dedicated router to repair the corrupted nets, because it is suited well for connecting elements with a higher amount of interconnect points between them. Of course, our approach to the problem is not the only possible; therefore, other solutions may produce better results.

4 Conclusion

The goal of this project was to determine and repair nets which are corrupted after a module re-placement due to slight irregularities in the FPGA fabric. This goal could partially be met. The analysis of corrupted nets is functional; although there is a method to fix the errors, we suggest a re-routing of the nets, because our solution is not applicable for the encountered error sources.

Our approach is based on isoelectric potentials, on which a breadth-first search finds missing interconnect elements. As pointed out in chapters 2 and 3, the basic approach has correct results. Nevertheless, the errors encountered in the conflicting nets have shown to be of a kind which is not applicable to the breadth-first-search as solution.

In conclusion it seems that there is currently no possibility to determine the exact properties of an net with RapidSmith based on the `.xdl` files. Corrupted nets can be found, but inconsistencies with the further processing complicate the possibility to distinguish between functional and corrupted nets. During our experiments with the RapidSmith hand router we started to suspect that our database might be to small. Most nets, even those of not broken designs e.g. designs that can be used without a problem by the ISE tool-chain, contain no wires or other elements connecting the pins with each other. The few Nets that are connected are usually of a rather simplistic manner. It should be noted that there is the slim possibility of bugs within RapidSmith. While we did not encounter any evidence for bugs we must note that the RapidSmith version used in this project showed some differences to the publicly available version on http://rapidsmith.sourceforge.net/. While most of these differences seem to be improvements, for example the wireEnumerator was upgraded by one version, it should also be noted that the GUI seems to be missing files.

All things considered, we recommend to use the algorithmic approach of this work with a bigger dataset or an alternative tool. For the meanwhile, we recommend to use a re-routing in case of error, because it is probably better than the suggested approach.

5 Bibliography

[1] Xilinx ise integrated synthesis environment, 10 2013.

[2] Jaren Lamprecht Philip Lundrigan Brent Nelson Brad Hutchings Christopher Lavin, Marc Padilla and Michael Wirthlin. A library for low level manipulation of partially placed and routed fpga designs. Technical report, NSF Center for High Performance Reconfigurable Computing (CHREC), Department of Electrical and Computer Engineering, Brigham Young University, http://rapidsmith.sourceforge.net/doc/TechReportAndDocumentation.pdf?format=raw, 1 2014.

[3] Christopher Lavin, Marc Padilla, Philip Lundrigan, Brent Nelson, and Brad Hutchings. Rapid prototyping tools for fpga designs: Rapidsmith. In *Field-Programmable Technology (FPT), 2010 International Conference on*, pages 353–356. IEEE, 2010.

[4] Jakob Wenzel and Christian Hochberger. Rapidsoc: short turnaround creation of fpga based socs. In *Rapid System Prototyping (RSP), 2016 International Symposium on*, pages 1–7. IEEE, 2016.