USER-CONTROLLED ROUTING TO IMPLEMENT CONFIGURABLE DELAY ELEMENTS IN FPGAS

BACHELOR THESIS

Matthijs Aanen

Supervisors: Dr. Ing. D.M. Ziener, Dr. J. Pathrose Vareed, Dr. Ing. E.A.M. Klumperink

Computer Architecture for Embedded Systems Group
Faculty of Electrical Engineering, Mathematics & Computer Science
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.2 Increased Fan-Out</td>
<td>33</td>
</tr>
<tr>
<td>7.3 Inaccuracies</td>
<td>34</td>
</tr>
<tr>
<td>8 Conclusion</td>
<td>36</td>
</tr>
<tr>
<td>9 Recommendations</td>
<td>37</td>
</tr>
<tr>
<td>References</td>
<td>38</td>
</tr>
<tr>
<td>A Proof of Concept Design</td>
<td>39</td>
</tr>
<tr>
<td>B Program Source Code</td>
<td>41</td>
</tr>
<tr>
<td>B.1 Detour Routing Program</td>
<td>41</td>
</tr>
<tr>
<td>B.2 Fan-Out Routing Program</td>
<td>44</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

This chapter serves to introduce the reader to the motivation and goal of this thesis. Additionally, an outline of the thesis is appended.

1.1 Motivation

Conventionally, FPGAs are used to implement synchronous systems. This means that a clock signal and latches are used to ensure that transitions in signals occur simultaneously [1]. If unbalanced path delays occur within these systems, small unwanted transitions in signals (called glitches) can occur. They do not impact the behaviour of the system, since the clock frequency of the system will be chosen in such a way that the signals have settled before the end of a clock period. However, glitches can have an impact on the power consumption of the device.

Glitches could be captured with an external oscilloscope. However, a monitor can be used to make a snapshot of the glitch within the FPGA. Such a monitor can be implemented by having the signal fed into a tapped delay line. If the signal at each of the taps is latched at the same time, a snapshot of the signal is made. Such a configuration (shown in Figure 1.1) is also used in TDCs (Time to Digital Converters), a device that measures the time between two pulses.

![Figure 1.1: A tapped delay line with a latch on each tap. From [2].](image)

Concepts like Vernier lines and successive approximation [3] can be applied to increase the accuracy of a TDC. However, the resolution of these systems is still dependent on the accuracy of the individual delay elements.

Furthermore, high accuracy delay elements can be beneficial in the following applications:

1. **Security**
   
   Delay elements can be used to implement physically unclonable functions (PUFs) in FPGAs, based on ring oscillators. PUFs are used to verify the identity of chips, similar to how fingerprints are used to aid in the identification of persons [4].

2. **Sensors**

   As mentioned, delay elements are used in TDC sensors, used to measure time. However, since delay time in integrated circuits is dependent on two physical quantities: the supply voltage and the temperature of the device [5]. Increasing the accuracy and resolution of delay elements will in turn increase both these properties in measurements.
With the work in this thesis, the accuracy of delay elements can be improved, aiding the advancement in the above mentioned fields.

### 1.2 Assignment Description

The initial description of the assignment is stated as follows:

*The goal of this thesis is to develop and evaluate a method to construct configurable delay elements (DEs) in Xilinx FPGAs that exploit net delays through user-controlled routing. This method will involve usage of third-party CAD tools which are able to manipulate the design during different stages of the Vivado Design Flow and allow the developer to deploy their own placement and routing algorithms.*

*An evaluation of the method will be done after a characterisation of the physically implemented DEs. If time permits, and the performance is adequate, a proof of concept FPGA-based system can be developed that demonstrates their potential. Such a system would, for instance, be a TDC (Time to Digital Converter), or another system where performance is impeded by timing inaccuracies.*

The mentioned delay elements (abbreviated to DEs from now), should be implementable using the basic FPGA fabric present in modern Xilinx FPGAs and not be constrained to a single device. Additionally, a large number of them should be available so that they can be used in tapped delay arrays, for example. These requirements means that usage of special timing resources (such as input and output delay primitives) will be avoided.

It should be noted that the “user” mentioned in the assignment refers to the developer that is implementing a design in the FPGA, not the consumer that uses the product which the FPGA might end up in.

### 1.3 Thesis Outline

For this thesis, two methods, based on different mechanisms, of creating DEs have been researched. The process of designing, implementing and testing these methods is described in this report. Physically implemented DEs have been evaluated in a benchmark system. However, due to shortage of time, the methods have not been used in a functional proof of concept system. Recommendations on the further improvement of the methods will be discussed, as well as their expected potential.
Chapter 2

Theoretical Background

It is assumed that the reader is familiar with both digital and analog electronics, as well as FPGAs in general. This chapter provides the reader with the additional background information and glossary required to understand the thesis. Furthermore, the relation between the work in this thesis and previous research will be described.

2.1 Xilinx FPGA

During the thesis, experiments have been performed with devices from the manufacturer Xilinx. Xilinx provides development software for their devices, and applies a terminology that is specific to them. Both will be explained in this section.

2.1.1 Vivado

For development on their FPGA devices, Xilinx recommends their own software suite Vivado [6]. This section describes the design flow of the software and the simulation types that can be done with it.

Design Flow

Figure 2.1 displays the different stages of the design flow in Vivado. The user starts with the design entry. This is where the system is described in a combination of HDL (Hardware Description Language) code and existing IP (Intellectual Property)(Vivado provides a library of functional blocks that can be used to speed up the design process).

After each of the main steps (design entry, synthesis, implementation and device programming), it is possible to evaluate the design in its current state, where each stage offers different methods to do this.

As with many design methods, the flow is iterative; the developer will move back and forth between different stages of development and versions of the design until he or she is satisfied with the finished system.

![Figure 2.1: Conventional development flow for Vivado.](image)

Page 6
Simulations

As seen in Figure 2.1, there are three stages of the design where simulations can be performed. These are: after design entry, after synthesis or after implementation. Vivado can perform behavioural simulations in each of these stages. However, a timing simulation can only be performed on a synthesised or implemented design. This is due to the fact that the used hardware is unknown before synthesis has been performed.

2.1.2 Hardware

The hardware elements of Xilinx FPGA can be categorised into “device objects” [7]. These objects are fixed physical parts in a Xilinx FPGA device and cannot be altered. However, most of them can be programmed, allowing a design to be implemented in the FPGA.

BEL

A BEL, short for Basic Element, is the lowest level component in the FPGA fabric (that is available to the user). BELs are located within sites. Some examples of BELs are registers, lookup tables or (de)multiplexers (for routing purposes).

The points at which wires are connected to a BEL is called a (BEL) pin.

Wire

Wires in Xilinx FPGAs are actual physical wires in the device that are used to carry signals between PIPs and pins. They only exist within a single tile.

Wires are not programmable.

Node

Nodes are sets of physically and electrically connected wires [7], that can span multiple tiles.

PIP

PIPs (Programmable Interconnect Points) are elements in the FPGA fabric that can connect two wires with each other. By adding a selection of PIPs to a net, it can be routed.

A set of PIPs with overlapping wires is called a PIP junction. Here, a net could fan-out (split up to have multiple destination pins).

Even though Xilinx is secretive about the transistor level implementation of the device objects, Vivado shows their is_buffered property. It is assumed that this property describes whether the PIP is implemented as a transmission gate, or a buffered one, such as a logical AND gate. This property and its relevance to this research is further elaborated in Chapter 7.

Switchbox

A switchbox is an object inside a tile that provides routing resources. They group together many PIP(junction)s, allowing multiple nets to be routed through them.

Site

BELs are grouped together in sites (also called slices). Sites reside within a tile and can be accessed through input and output pins on their boundary. Wires within the site connect the BELs inside with each other and the site pins.
Tile

The highest level elements in the FPGA fabric are the tiles. Tiles are arranged in a square grid (shown in Figure 2.2). A tile contains elements with the same general function, like DSP (Digital Signal Processing) or routing resources (interconnect).

Figure 2.2: A section of the FPGA fabric of a Zynq-7000 device, taken as a screen-capture from Vivado. The blue lines indicate the boundaries between different tiles (the borders of one tile are highlighted in red). The gray lines are device objects (borders).

Hierarchy

Figure 2.3 displays a simplified example tile in a Xilinx FPGA. This type of tile features two sites and a large routing BEL (switchbox).
Figure 2.3: Simplified overview of a *Combinational Logic Block* type tile in the Artix-7 FPGA Family. Please note that the type, amount and arrangement of the device objects differs from those in the actual device.

Typically, the outgoing wires on a tile like this would enter a neighbouring interconnect tile.

### 2.1.3 RapidWright

In order to gain a high degree of control over routing of designs, 3rd party software is required. A framework written in Java, called RapidWright, has been chosen. RapidWright is open source and is developed by Xilinx employees [9].

RapidWright is designed to edit design checkpoint files (.DCP) that can be generated by Vivado. This can be done at any stage of the design flow, up until bitstream generation, which can only be done by Vivado [9]. Once a design is imported into RapidWright (or when a new design is created), RapidWright can generate a new design checkpoint file, which can be opened with Vivado. Figure 2.4 shows how design checkpoint files are used as an interface between Vivado and RapidWright, and how they can be exchanged at all different stages of the flow.
2.2 Delay in FPGA

There are two mechanisms that cause a delay between cascaded circuits in FPGAs. The first one is the propagation of EM (electromagnetic) waves through a conductor, the second one is the slew rate of the voltage on a net, affected by different impedances. Both mechanisms will be introduced in this section.

2.2.1 Wave propagation

Voltages can travel through conductors as EM waves. The propagation velocity is found with the following equation [10]:

\[ v = \frac{1}{\sqrt{\epsilon \mu}} = \frac{c}{n} \]  

(2.1)

Here, \( v \) is the propagation speed of the wave, \( c \) is the speed of light in vacuum and \( n \) is the index of refraction [10]. The value of \( \epsilon \) is dependent on the permitivity of the material, \( \mu \) is dependent on the permeability of the material [11].

However, signals usually flow overwhelmingly outside the electric conductor of a cable [11]. In practice, the propagation speed in a wire is order of 50%–99% of the speed of light [11]. This would correspond with a delay of 6.7 ps down to 3.3 ps per millimetre of wire.

The time a wave takes to travel from one end of a wire to the other is found by dividing the length of the wire by the propagation speed of the wave. This means that the delay time is directly proportional to the length of the wire.

2.2.2 Switching delay

In order to turn on or off a FET in the chip, its gate has to be charged or discharged (depending on the doping). However, the node to which this gate is attached has a capacitance, which mainly is caused by the wires in the device, and partially by the capacitance of the gate(s) itself. Since the transistors on an FPGA are used for logic, not power, small transistors are used. Therefore, it is expected that the wire capacitance dominates the capacitance of a net, not the attached transistors.
A system with a driver, wire(s) and attached logic can be modelled as an RC circuit. The time it takes for the step response of an RC filter to reach a certain voltage is linearly dependant on the value of the resistor and that of the capacitor. If the output of the RC filter is connected to the gate of a FET, reaching the threshold voltage of this transistor can be postponed by increasing the resistance and/or capacitance in the circuit, effectively increasing the delay. Figure 2.5 shows how such an RC circuit would look between two buffers, when the input signal is pulled high (the capacitor is charged from the supply voltage through the resistance of the transistor(s) of the buffer).

Figure 2.5: Circuit diagram showing how an RC filter is modelled a net that is being pulled high. The net is positioned between two buffers.

### 2.2.3 Influences

There are two additional physical quantities that also influence the delay of a signal in an FPGA, which are not dependent on the die itself, and the implemented design. These quantities are the die temperature and the internal supply voltage.

### 2.3 Previous research

In [12], efforts have been made to implement a high resolution TDC in an FPGA. In this work, a resolution of 7.4 ps was achieved by exploiting routing resources in the device (in combination with other techniques). However, there was only an indirect control over the routing, which was accomplished by constraining placement of the measurement matrix.

Figure 2.6: A set of 1024 routing paths ordered by delay time. From [12].
Figure 2.6 from [12] shows a set of 1024 routing paths ordered by delay time, for different placement constraints. The used TDC architecture in this work requires a highly linear increase in delay for the best performance. For this reason, the authors chose to use the structure resulting in line 1 (Delay Characteristic of 8x128).

However, in [12] it is mentioned that the other structures, which result in a smaller average increment in delay, would allow for a higher theoretical time resolution (the increments in time are smaller). Their measurements suggest that it is possible to route the paths with such small increments, but that they have insufficient control over the routing to implement it properly. Hence, it is expected that the performance of this TDC could be improved upon even further if a higher control over placement and routing is introduced.
Chapter 3

Analysis

In this chapter, two relevant subjects to the topic are analysed: the delay characteristics in FPGAs that can be used to solve the problem, and in what way simulations can be used.

3.1 Exploitable Delay Characteristics

In Chapter 2, the influences on delay time in FPGAs are listed. Two of these characteristics result in delay times that are directly dependant on the design that is programmed into an FPGA and are therefore candidates to be exploited in the DEs. The first is the propagation of EM waves through the conductors. The second is the delay between transitions of cascaded circuits by parasitic impedances at the interconnecting net, the output of the driving circuit and the input of the preceding circuit.

3.1.1 Wave Propagation

The propagation of EM in conductors was briefly described in Chapter 2. The wires in FPGAs are fixed. This means that the propagation speed cannot be controlled. However, routing in FPGAs is done by concatenating wires, meaning that effectively, the length of wires can be controlled. The delay that occurs due to the propagation of EM waves is then the accumulated delay of each wire.

3.1.2 Transistor Switching Delay

As explained in Chapter 2, parasitic impedances cause additional signal delay between cascaded circuits. This delay can be controlled, since FPGAs provide a large number of interconnect resources which can be used to append wires and circuits to each other.

For a single driver and net, modelled as an RC circuit, the transition time is influenced by the driver output resistance (the \( R \)), and the net or load capacitance (the \( C \)). The output resistance could be influenced by choosing a different driver for the net. For example, Xilinx FPGAs have special buffers that are used to drive high fan-out clock nets [13]. It is expected that these buffers are implemented with lower resistance MOSFETs than generic logic circuits in order to maintain a high slew-rate.

The other method would be to influence the capacitance of the RC network. This could be done by changing the load, again, using interconnect resources. Wires and circuits with input capacitance could be added to a net in order to lower the slew rate.

The model used for this thesis assumes that the resistance of wires and PIPs is insignificantly small with respect to the output resistance of the driver of a net.

3.2 Simulations

The most obvious method to make a prediction on the delay of a physically implemented DE is to perform a simulation on the design. Vivado allows the user to perform simulations during several stages of the development. Before synthesis has been done (but also after), Vivado offers a Behavioural Simulation. However, this type of simulation will not provide any information about the timing characteristics of the
design. These can only be obtained from Post-Synthesis Timing Simulation or the Post-Implementation Timing Simulation, which, obviously, can be run after synthesis and implementation respectively.

Depending on the type of DE, a simulation after synthesis can be useful. At this point in the design flow, there is no final information about the used BELs available yet (unless specifically constrained by the user). This means that if the design of the DE relies on placement and routing, instead of a combination of cells, this type of simulation is not expected to provide accurate results.

After implementation, the placement of cells and primitives on BELs and the routing is known. Therefore, a simulation performed at this point in the design flow is expected to yield more accurate results for all type of DE designs.
Chapter 4

Concepts

Two methods of implementing DEs, based on the delay characteristics described in Chapter 3, will be tested. Both of them are explained in this chapter.

4.1 Increasing Path-Length

One of the most obvious delay characteristic that can be exploited in a configurable DE is electric signal propagation. As mentioned in Chapter 3, the time it takes for a signal to traverse a path is influenced by the propagation speed of the electric wave (which cannot be changed without changing the die) and the length of the path (which can be manipulated).

Thus, routing a net with a detour increases the delay a signal experiences between leaving a source and reaching its destination. This concept is displayed in Figure 4.1.

As a side effect of the elongated path will be the increased capacitance of the node. This will aid in the increase of the delay time, as it will decrease the rise and fall time of the signal.

Elongating the path will require additional routing resources, but no additional BELs. This could be a disadvantage for large designs that needs those routing resources.

The lowest increment of delay is based on singular added wires, adding some capacitance and length to the net. Since the time it takes to increase or decrease the voltage at the output of an RC filter is linearly dependant on $RC$, and the time it takes for an electromagnetic wave to propagate through a conductor is linearly dependent on the length of it, the added delay is expected to be directly proportional to the number of added wires or PIPs (routing with one extra wire requires one PIP). For the RC filter model, it is assumed that the resistance of the wire is very small in comparison with the output resistance of the driver of the net. The described relation is shown in Figure 4.2.

Equation 4.1 was used to plot the linear dependency between added delay time and amount of added wires/nets, where $t_{delay}$ is the added delay time, $\alpha$ is the amount of added delay time per wire/PIP (arbitrarily set to 500 ps) and $n$ is the amount of added wires/PIPs.

![Figure 4.1: Diagram showing how delay can be introduced by elongating a net.](image-url)
Figure 4.2: Expected linear dependency between amount of added wires or PIPs and the added delay time, according to Equation 4.1. An arbitrary increase in delay time per added element of 500 ps was chosen.

\[ t_{\text{delay}} = \alpha \cdot n \]  \hspace{1cm} (4.1)

### 4.2 Increasing Fan-Out

As explained in Chapter 3, increasing the capacitance on a node will increase the rate at which the voltage on this net rises or falls. This could be achieved by increasing the amount of sink pins on this net (this number is called the fan-out). Each additional sink pin requires at least one wire to be attached to the net, which means that the capacitance of the net is increased. Figure 4.3 shows how this method can be implemented.

It is not possible to append wires that drive no pins to the net (and thus no load), because this could case reflections to occur.

Figure 4.3: A simplified diagram showing the capacitance of the added wires.

This configuration does not add a DE as an element that the signal has to pass through. Rather, it delays the arrival of a transition in the signal at any sink pin. Thus, if this type of DE is implemented in a
tapped delay line configuration, the designer should isolate each net by placing buffers in between each net.

This method potentially uses a large number of pins on BELs which cannot be used for any other purposes. However, depending on the type, a BEL can be shared between a DE and a functional cell in the design. For example, if an AND gate primitive is implemented in a 6LUT BEL, there are still 4 pins available on that BEL.

Figure 4.4 shows the expected linear increase in delay time of an individual DE as a function of the amount of added capacitive elements to the net. This linear dependency is caused by the increasing RC time, as explained in Chapter 2 and 3.

![Figure 4.4](image)

Figure 4.4: The expected linear relation between the amount of added wire capacitance and the added delay time to a net. An arbitrary increase of 500 ps delay per added element was chosen.

Equation 4.1 has been used to create this plot, with the only change being that \( n \) denotes the amount of added capacitive elements and \( \alpha \) the amount of added delay time per capacitive element.
Chapter 5

Implementation

In this chapter, the proposed concepts from Chapter 4 are implemented in a Xilinx FPGA. After an overview, the involved processes for each concept will be elaborated.

5.1 Overview

The approach that will be used to implement both concepts involves a modified version of the default Vivado design flow (shown in Figure 2.1). This adaptation can be seen in Figure 5.1.

![Figure 5.1: Modified design flow for Xilinx FPGA with Vivado and RapidWright, used to implement the two introduced concepts. (Vivado and RapidWright logos retrieved from [14] and [9], respectively.)](image)

Implementation of the two proposed concepts will be done according to the following list:

1. The two first step of the implementation is the design entry and synthesis. This will be done in Vivado and is explained in Section 5.3.
2. The main difference between the implementation of both concepts occurs after synthesis. At this stage in the design flow, a design checkpoint will be generated by Vivado. This design checkpoint is the starting point for the implementation of both proposed methods.
3. The implementations of these concepts will be done in RapidWright and involve manipulations on the netlist and the addition of constraints to the design. These processes are explained in Sections 5.4 and 5.5.
4. After generating a new design checkpoint in RapidWright, the design can be loaded back into Vivado. Here, the remaining cells and nets can be placed and routed, respectively. This is described in Section 5.6.

Only the BELs and nets that are critical to the delay elements will be placed and routed by the external programs. This allows for less constraints in the amount of resources available to the programs.
5.2 Hardware

The device that is used for the experiments is a Xilinx Zynq-7020 system on a chip (shortened to SoC from now). This device has two ARM microprocessor cores and programmable logic that is similar to that of the Xilinx Artix-7 FPGA.

The device is mounted on the ZedBoard development board. It powers the device, but also provides many input/output interfaces and DDR3 memory for the microprocessors in the SoC [15].

![Diagram showing I/O peripherals of the ZedBoard and Zynq-7020 SoC](image)

Figure 5.2: Diagram that shows the I/O peripherals that the ZedBoard provides to the Zynq-7020 SoC. Adapted from [16].

5.3 Creating a Design Checkpoint

The first step in the implementation of both concepts is the creation of a design checkpoint that can be exported from Vivado and imported into a RapidWright program. The process of creating this checkpoint is described in this section.

5.3.1 Topology

The proposed concepts will both be applied to the same benchmark design. The design is that of a ring oscillator, shown in Figure 5.3.
During a single oscillation, a signal will propagate twice through the ring oscillator. For this reason, the total delay of the loop can be found with the following formula:

\[
 t_{\text{Loop}} = \frac{1}{2 \cdot f_{\text{Oscillation}}} \quad (5.1)
\]

With a reference total delay time in mind (e.g. that of the system without any manipulations), the measured loop delay time can be used to find the added delay per individual DE. This can be done with the following formula:

\[
 t_{\text{DE}} = \frac{t_{\text{Loop}} - t_{\text{Ref}}}{48} \quad (5.2)
\]

Here, \( t_{\text{DE}} \) is the added delay time per individual DE, \( t_{\text{Loop}} \) is the measured delay time (substituted from Equation 5.1) and \( t_{\text{Ref}} \) is the reference delay time (also substituted from Equation 5.1). There is a division by 48, since there will be 48 identically manipulated nets in the generated systems. The first and last net are not used to have the source and sink element on each manipulated net the same, allowing for an identical implementation of each DE.

A design of this topology has been chosen for the following reasons:

- The accumulated delay of the elements can easily be calculated from the (externally) measured oscillation frequency.
- The deviations in delay time between different elements is averaged out. (Individual delays cannot be measured, however.)
- The DEs are configured in a tapped delay line, which is used in TDCs and DSP [17]. This makes it similar to functional applications that make use of DEs.
- It forces the described constraint of allowing a large number of DEs to be implemented at the same time (a requirement mentioned in Chapter 1).

Due to the DEs being cascaded in this setup, not all their characteristics can be measured. (This will be elaborated further in Chapter 7.)

The used design features more elements than the ones displayed in Figure 5.3. The design was initially meant to also function as a signal monitor for glitches. An architecture was made that allows communication between the processing system and the programmable logic. Additionally, a counter has been added, allowing the system to measure its own oscillation frequency. This system is further described in Appendix A.
Due to a deficiency in time, these components of the system have not been fully implemented. They are unused in this research and do not influence the results since Vivado removed them during optimisation.

5.3.2 Design Entry

The described topology has been entered in a new Vivado project. It is made up of modules (described by VHDL code) connected in the figuration shown in figure 5.4.

Figure 5.4: Block diagram showing how different modules of the used system are interconnected.

The cascaded DEs exists within the `monitor_0` module. As mentioned earlier, this module is designed to be a tapped delay line with latches on each net. However, these latches and their outputs (data[49:0]) and set pins (start) are unused. The module named `lut2_switch_0` contains the logic for the NAND gate.

5.3.3 Constraints and Synthesis

After the design was entered into Vivado, constraints were added to force the placement of the buffer elements, and select which physical pins on the device were used for the input and output signals.

The buffers of the design are placed on 6LUT BELs in the VHDL code. Each buffer is placed in its own tile in order to provide space and routing resources for their implementation. They are positioned in a vertical row to make the process of visually evaluating the implemented DEs easy (structures or patterns will be more visible), this is displayed in Figure 5.5.
5.4 Increasing Path-Length

At this point, the design checkpoint has been generated by Vivado. This section explains the process of manipulating the generated design checkpoint file.

5.4.1 Approach

To increase the path-length of a net, the nets in between the buffer elements will be routed by a custom pathfinder that is implemented with RapidWright. This pathfinder constraints the routing resources of a net to the row that the driving buffer is in. This prevents the routes from influencing each other and allows the user to visually review the path from Vivado. In functional designs, with a proven method, the routes do not have to be constrained.

This concept is implemented by choosing a wire that has to be included in the path. First, the program finds a route from the driving BEL of the net to that wire. Then, a path from that wire to the destination BEL is added to the net. In this manner, the net takes a detour to its destination.

5.4.2 RapidWright

Figure 5.6 shows how the concept is implemented in the Java program with the RapidWright framework. The tile that is used for the detour route is found by taking the tile of the source BEL, and iteratively taking the next interconnect tile while keeping count.
This program leaves much margin for optimisation. Due to the restrictions in time, no efforts were made to reduce the computational complexity of the algorithm.

5.5 Increasing Fan-Out

5.5.1 Approach

The 6LUT BELs are readily available throughout the fabric of the FPGA. For this reason, the input pins of these elements are selected to serve as the sink pins that will be added to a net. Since the input impedance of each individual pin on a 6LUT is unknown, the choice has been made to add pins in sets of 6 (all the input pins on a 6LUT). This should allow for a uniform resolution in added capacitance.

The implementation method starts after synthesis of the RO design with the tapped delay line. Each DE cell in the delay line contains a LUT1 primitive which will function as the dividing buffer, creating 50 separated nets. All of these nets could have their fan-out increased. However, to keep the layout of the design free from being cluttered, the first and last nets in the delay line will not be touched, meaning that there will be 48 manipulated nets in total. Figure 5.7 shows the resulting setup.

Figure 5.6: Simplified diagram that shows how the developed RapidWright program works. Words in monospace font refer to objects of a class introduced by RapidWright.
Figure 5.7: Diagram of the ring oscillator using increased fan-out to decrease the oscillation frequency. Both the NAND and LUT1 (buffer) primitives are implemented with 6LUT BELs.

The influence on the capacitance per added LUT6 cell can be controlled by choosing the distance at which the cell is placed from the original net. Placing the cell close to the original net would cause the length of the wire to the BEL to be short, thus giving it a small capacitance. Placing the cell on a LUT that is further away would have the adverse effect.

5.5.2 RapidWright

After the synthesised design is used to generate a design checkpoint file, the information can be loaded into the program. The naming convention from the synthesised design then allows the program to find all the DE cells. It will then append a configurable number of 6LUT BELs to this cell and connect all their input pins to the output net of the DE cell.

An overview of the program is provided in Figure 5.8.

Figure 5.8: Simplified diagram of the fan-out increasing Java program. Words in monospace font refer to objects of a class introduced by RapidWright.
5.6 Placement, Routing & Bitstream Generation

A TCL script was used to convert the generated design checkpoints into bitstreams that can be used to program the FPGA. The script performs the following tasks for each generated checkpoint:

- Assign a location to the unplaced cells in the design (place_design)
- Optimise the design with opt_design (some unused cells in the design are removed)
- Route the remaining nets (route_design)
- Generate the bitstream file (write_bitstream)

The cells and nets that have been placed and routed in the RapidWright program are not touched by Vivado. This is due to the constraints added by the program instructing Vivado not to do so.

5.7 Simulations & Measurement Setup

Efforts have been made to perform timing simulations on the finalised design checkpoints. However, since, by default, Vivado imports design checkpoint without a project, timing simulations were not available. Due to the restrictions in time and the earlier discovery that simulations did not yield accurate results, the decision was made to discontinue any attempts to solve this problem.

In order to measure the oscillation frequency of the system, programmed on a physical device, an oscilloscope (Teledyne Lecroy HDO6054A-MS) was used. With a probe, the output voltage of the pin on the ZedBoard was measured. A spectogram with highlighted peaks was used to read out the frequencies. The setup is shown in Figure 5.9.

Figure 5.9: A photograph of the measurement setup, featuring the ZedBoard which is hooked up to a HDO6054A-MS oscilloscope.

Measurements have been done at room temperature, and each set of measurements was done in rapid succession in order to prevent the temperature of the die from influencing the results. However, programming the FPGA and logging the measured frequency was done manually. Furthermore, the assumption was made that the differences in power consumption of programmed designs are insignificantly small.

The Zynq-7020 was powered by the voltage regulator on the ZedBoard, which in turn was powered by the AC/DC adapter shipped with the ZedBoard. The assumption was made that the supply voltage was stable enough to not influence the results were not influenced by it.
Chapter 6

Results

The implemented concepts from Chapter produced a large number of design checkpoint files and measured frequencies. In this chapter, this data will be presented. The data will also be compiled into information that can be used to evaluate the performance of the DEs and their method of implementation.

6.1 Simulations

One of the proposed methods to evaluate the oscillation frequency of a design is to make use of the timing simulations that Vivado can perform. For different design variants, a timing simulation has been performed at the two available moments in the design flow: after synthesis and after implementation. The results can be found in Figure 6.1.

![Figure 6.1: Simulated and measured oscillation frequencies for different variants of ring oscillators. Simulated after synthesis and implementation and measured on a programmed device (Zynq-7020).](image)

It can be observed from this graph that there is a large deviation between the simulated and measured values (often in the order of 100%). In addition, there is no apparent correlation between the simulation results and the measurements. Due to these errors, the simulation results are deemed invaluable during the evaluation of oscillation frequencies. During the remainder of this research, only the measured oscillation frequencies will be taken into account.

6.2 Increased Path-Length

The developed Java program has been used to generate a number of design checkpoints with elongated paths. These checkpoints have been implemented with Vivado, as described in Chapter.
of these designs were made and used to program the FPGA. In this section, the generated design checkpoints and the corresponding frequency measurements will be discussed.

6.2.1 Generated Design Checkpoint

To have a quick insight in how the program performs, a special variant was run that sweeps over the length of the path for each DE. This iteration of the program will not be used to obtain frequency measurements. The resulting design is shown in Figure 6.2.

Figure 6.2: An edited screenshot from Vivado, showing the nets (yellow) that are routed by the Rapid-Wright program. The requested amount of spanned tiles is incremented once every 2 DEs. This means that the amount of requested amount of offset tiles ranges from 0 to 23. Please note that this design is not used for measurements, but only for (visual) inspection. Additionally, even though there are vertical lines, there are no overlapping nets. These occur because some of the used wires have pins in neighbouring rows. The green arrows indicate the length of the paths used to span the requested amount of interconnect tiles. The red rectangle contains the routed detour path of a single DE.

At first glance, the router performs well. The nets seem to increase in length linearly and only rarely leave their rows (the reason of which has not been determined). Additionally, there are no overlapping nets.

However, a flaw in the program surfaces when the first pair of lengthened paths is inspected (those with the shortest length). The route exits the originating tile, only to return to it, before going to the destination tile (seen in figure 6.3). Going to the destination tile directly, which was the intended behaviour, would have resulting in a higher oscillation frequency, and a more linear increase in delay.

Additionally, there is a gap in the array of nets, occurring when the requested offset is 16 tiles (see Figure 6.3B). The nets are not routed by the program, and will therefore be routed by Vivado, most likely resulting in a lower delay than desired.
Figure 6.3: Two screenshots that show a section of a routed design checkpoint. A is the resulting routing when the requested tile offset is equal to zero. B is the routed checkpoint when the requested tiles in the detour is equal to 16. In this case, the program had failed to do so. It was therefore routed by Vivado.

It should be noted that the design checkpoint in Figure 6.2 is not used for measurements. It is merely a sweep of the parameter for visual inspections within one design. Figure 6.4 is one of the used design checkpoints for measurements. In this generated design, the amount of spanned tiles requested from the RapidWright program is 26.
Figure 6.4: Two combined screenshots of the by RapidWright generated design checkpoint where the nets are routed with a detour of 26 tiles. The smaller framed picture is a zoomed in section of the larger image. Even though the zoomed out network shows horizontal wires, the wires from different rows are not directly connected.

6.2.2 Measurements

The oscillation frequency has been measured for each amount of spanned tiles. These results can be seen in Figure 6.5. The measurements have been performed on two separate devices of the same model (Zynq-7020 on a ZedBoard), in order to gain insight on the accuracy of the delay times.

![Oscillation Frequency Graph](image)

Figure 6.5: Oscillation frequency of the RO as a function of the amount of requested spanned tiles from the program.

As expected, the frequency peaks when the router is constrained to use a wire in the 16th interconnect tile.
The oscillation period at zero requested tiles was used as a point of reference in order to compute the added delay for each amount of tiles used in the detour. The resulting plot can be found in Figure 6.6.

Figure 6.6: Added delay per individual element, with the request for 0 tiles as reference. The measurements were done on two of the same devices (Zynq-7020 on a ZedBoard).

Because each generated design checkpoint used for measurements contains 48 delay elements, the amount of PIPs used for each net and the corresponding delay time is not known. The average is known however. Figure 6.7 shows the added delay from Figure 6.6 however as a function of the average amount of PIPs used to route the detour nets. A negative offset in amount of PIPs was added to find the amount of "added" PIPs. The resulting graph shows the expected linear relation between added PIPs or wires and delay time. Two linear trend lines have been added, both forced to intersect the origin, showing to what degree the measurements fit the expectation from Chapter 4.

Figure 6.7: A graph that shows the added delay time as a function of the average amount of PIPs added to a DE. The amount of initial PIPs has been set to that of the nets with no additionally requested tiles. For both datasets, a trendline has been added conforming to Equation 4.1.

When not constraining the trend line to the origin, and removing the problematic measurements from the data set (where the requested amount of tiles are 0 or 16), trend lines show an increase in delay time (parameter $\alpha$ in Equation 4.1) of 0.0878 ns and 0.0935 ns per added PIP for FPGA #1 and FPGA #2 respectively.

The difference between computed delay times of both FPGAs is plotted in Figure 6.8. It ranges from -0.932 ps to 88.9 ps and, too, seems to be linearly increasing.
Figure 6.8: A graph showing the difference between the delay times from both set of measurements in Figure 6.7. It is computed by subtracting the measurements from FPGA #1 from that of FPGA #2.

6.3 Increased Fan-Out

6.3.1 Generated Design Checkpoint

Since it was not possible to choose the placement of the added 6LUTs, this was done by Vivado after the checkpoint was loaded back into the suite. Vivado’s placement algorithm placed these BELs close to the buffers in the delay line. Figure 6.9 shows the design after routing and placement with 150 added 6LUTs.

Figure 6.9: Device view after the design has been fully placed and routed by Vivado. The amount of added 6LUT BELs is 150.
6.3.2 Measurements

The frequency measurements have been performed on the FPGA (Zynq-7020 on a ZedBoard). Due to the unexpected low degree of linearity, a second set of measurements was performed on a different device (FPGA #2), of the same model, in order to gain more insights. The interval between amount of added 6LUTs in each checkpoint was increased in a few steps. The resulting plot is found in Figure 6.10.

![Figure 6.10: Oscillation frequency of the RO as a function of the amount of added 6LUTs to each net.](image)

Using Equation 5.1 and 5.2 these frequency measurements have been compiled into the corresponding added delay times per DE, displayed in Figure 6.11. The delay time at zero added LUTs is taken as the reference point.

![Figure 6.11: Additional delay time per DE, for a different amount of added LUT BELs. Measurements have been performed on two devices of the same model (Zynq-7020 on a ZedBoard). A line with the difference between the two FPGAs has been added.](image)

The linearity of this curve leaves much to be desired. After a sharp rise in delay time, the growth comes to an end. The difference in delay time between the two FPGAs seems to be constant and has an average of 1.1 ps.
Chapter 7

Discussion

In this chapter, the validity of the results presented in this thesis are examined. Problems in the developed Java software are described as well as factors that could have affected the measurement results.

7.1 Increased Path-Length

After inspecting the generated design checkpoints of the implemented path elongating program, some bugs in the program surfaced. The first problem occurs with the net routed with the smallest offset. This problem is an oversight in the program; instead of trying to route the net to the first encountered interconnect tile, it contains an additional wire in the originating tile itself (see Figure 6.3A). If this net would have used a wire in the first neighbouring interconnect, the resulting oscillation frequency is expected to be higher due to the path being shorter.

A second problem occurred when the program is requested to route the net with an offset of 16 tiles. The program failed to route these nets (see Figure 6.3B). As a result, Vivado routes these nets, causing a large peak in the measured frequencies.

The reason for this failure is a shortcoming in the program implemented with RapidWright. When it is selecting an offset tile to route the net through, it counts the amount of tiles that are prefixed with the characters "INT.". This way, the program would only use regular interconnect tiles for the routing. However, at an offset of 16, the router encounters an interconnect tile that does not feature a switchbox, but serves as an interface to external devices. This tile does not contain any wires that are suitable for the path, causing the router to fail. The program was able to route all other nets.

7.2 Increased Fan-Out

During implementation of the increased fan-out concept did not appear fruitful. It was expected that, even without the control over placement, a correlation between the amount of added BELs and the delay time was present (over the whole range).

It was discovered after the measurements that PIPs have a property that could be the reason for this unexpected behaviour. This property describes whether or not the PIP is buffered. If a PIP is implemented as a transmission gate (which was assumed during the experiment), as displayed in Figure 7.1 the impedance that the starting node experiences at the PIP is that of the attached node. The PIP is then modelled as a switch.
Figure 7.1: A transmission gate circuit, used to electrically connect two nodes (A and B). If EN is high, the nodes are connected, otherwise not.

However, if the PIP is buffered, the attached node will be driven by the buffer, depending on the voltage at the starting node. The two notes are isolated in the sense that the impedance experienced by the starting node is that of the buffer, which is not influenced by the impedance of the node on the other side. This means that added capacitance on the attached node will not accumulate with that of the starting node. The two scenarios are both displayed in Figure 7.2.

Figure 7.2: Circuit diagram A) shows a simple model of the delayed net if a PIP would consist of a transmission gate. Circuit B) shows the same model, except if the PIP was buffered, and implemented with a logical AND gate. In this case, the capacitance behind the AND gate would not affect the node before the gate.

7.3 Inaccuracies

Temperature and core voltage can have a significant impact on the frequency of ring oscillators in FPGAs [5]. Due to the short time span of this thesis, these variables were not controlled during experiments.

Even though the experiments were done in rapid succession, and attempts have been made to prevent the device from heating up or cooling down during experiments, temperature fluctuations could have
affected measurements. This could be caused by changing in ambient temperature, or by designs having different power consumption.

It should also be noted that within-die variations (caused during production or ageing of the device), and thus absolute placement of the design, has an influence on ring oscillator frequencies [18]. This was not accounted for during the experiments. The delay line buffers locations were fixed during the experiments in order to reduce sweeping parameters.

Intra-die variations have been briefly investigated by performing the path-length measurements on two devices. This number of devices, however, results in too little data to compile into meaningful information about accuracy.
Chapter 8

Conclusion

Two concepts of DE have been implemented and evaluated. The first one, which makes use of path length in order to increase delay, shows to be working. The problems of the software are known, and can be solved easily. However, the model used for the used routing algorithm assumes that all wires add the same delay time; the algorithms only counts the amount of wires, but does not consider the type of wires. It is expected that this had a negative impact on the accuracy of the generated DEs. In Chapter 9 a solution for this shortcoming is given.

The second concept, which exploits the fan-out in order to increase delay, has not been proven to work yet. However, as with the previous one, this method's problems are known and can be solved easily. At this moment, there is not enough information to make judgements on it's capabilities. Nevertheless, a working version of this method has many options for optimisation, due to the large number of parameters available: the number of pins, the length of the wires to those pins and the types of BELs used. Additionally, the performed measurements showed a relatively low spread, in comparison with the first method, even though there was a low degree of control. Further experiments can determine if the variations can be controlled, or that they are the result of low accuracy. In the end, there is no evidence that this concept cannot succeed. Therefore, in combination with previously mentioned arguments, this method is judged to have the highest chance of becoming a method for creating high precision delay elements.
Chapter 9

Recommendations

Increasing the path length is a working method to create delay in an FPGA. However, the developed method can be improved. Obviously, the bugs in the program should be prevented. Furthermore, the routing algorithm used in the program can be improved upon; timing models could be extracted from Vivado to increase the accuracy of the custom router. The resolution can then be based on the predicted delay of single wires, instead of on the amount of used tiles.

The added delay by increase fan-out has a chance of success if the buffered interconnect points are evaded. Then, assuming it works, more accuracy in the delay can be achieved by introducing control over the placement of BELs and the routing (and therefore the length and capacitance of the wires).

A better model of the FPGA fabric could help with predicting the DE time values. The model used in this thesis can be improved upon by increasing the complexity. For example, if the resistance of the wires and PIPs is taken into account, the model would consist of multiple cascaded RC filters. Additionally, efforts can be made to perform simulations on the fully implemented designs (manipulated using a RapidWright program) in Vivado. As explained in Chapter[5] this was not done in this thesis.

Because the next steps in the development of a DE based on increase fan-out are very small, it is recommended to first continue the research in this direction. After implementing this technique, while accounting for buffered PIPs, the effectiveness of the method can truly be evaluated.

Lastly, an improved testing setup should be developed. This setup should, for one, account for environmental effects by controlling and monitoring the temperature of the device and its core voltage. This would increase the measurement accuracy. Additionally, prototyping and testing can be sped up and the setup can be reduced in complexity by implementing frequency measurement hardware in the FPGA itself (using an asynchronous counter and the processing system on a SoC device). An architecture for this purpose is proposed in Appendix[A].
References


Appendix A

Proof of Concept Design

An simple architecture for development of the DEs was designed and partially implemented during this thesis. It was designed for the ZedBoard. The design describes a system that creates glitches and feeds them into a signal monitor, based on a tapped delay line (see Figure A.1). The design features a controller in the programmable logic which controls the signal generator, the delay line, and the programmed interconnect features (a demultiplexer is used to switch the system between RO mode and signal monitor mode).

![Figure A.1: Block Diagram of proof of concept system that was not fully implemented. Bold lines indicate combined signals and wires (busses).](image)

The tapped delay line can also be put into a loop with an inverter, turning it into a ring oscillator. This way, the system contains all the functions needed to measure delay line, without changing the programming of the logic. In order to exclude the interconnect delay from that of the delay line during measurements, it should be measured too. This is done using a second demultiplexer.
Figure A.2: The two modes in which the ring oscillator can be configured.

The controller in the FPGA part of the device is connected to the processing system through the AXI interface of the device. On the processing system, a conventional program can run that serves as an interface between an external device (which can be connected over USB) and the system on the FPGA. This way, an external computer can command the system and log the measured data.

Figure A.3 shows the architecture in a block diagram in Vivado. Not all modules in the diagram are implemented, they are nonfunctional black boxes.

Figure A.3: Block Diagram of proof of concept system that was not fully implemented.
Appendix B

Program Source Code

In this appendix includes the developed Java code (using the RapidWright framework). Both programs are the content of a .java file which is placed in an Eclipse project, together with the RapidWright source code.

B.1 Detour Routing Program

The following Java program uses the RapidWright framework to route the nets with a detour (as explained in Chapter 4).

```java
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Comparator;
import java.util.HashSet;
import java.util.List;
import java.util.PriorityQueue;
import java.util.Iterator;
import com.xilinx.rapidwright.design.Cell;
import com.xilinx.rapidwright.design.Design;
import com.xilinx.rapidwright.design.Net;
import com.xilinx.rapidwright.device.BEL;
import com.xilinx.rapidwright.device.Device;
import com.xilinx.rapidwright.device.PIPE;
import com.xilinx.rapidwright.device Site;
import com.xilinx.rapidwright.device.Tile;
import com.xilinx.rapidwright.device.Wire;
import com.xilinx.rapidwright.edif.EDIFCell;
import com.xilinx.rapidwright.router.RouteNode;

import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.text.SimpleDateFormat;
import java.util.Date;

public class DetourRouter_V3 {

    // Constants
    private final static String[] prefixChars = {"A", "B", "C", "D"};

    private final static String dcpInFile = "C: \Users\User\Documents\UT\M12\EXEC_I\Project_partially_placed\checkpoint.dcp";

    private static int detourTiles;
    private static Design design;
    private static Device device;

    // Change with each iteration of DEs
    private static int delIndex;
    private static Cell deSrcCell;
    private static EDIFCell deParentCell;
    private static Net deNet;
    private static Tile destinationTile;

    public static void main(String[] args) throws IOException {
```
String timeStamp = new SimpleDateFormat("HH:mm").format(new Date());
double percentage = 0.0;
double dper = 0.13888888888888888;

for (int uu = 0; uu < 30; uu += 1) {
    detourTiles = uu;
    System.out.println("Run # " + uu);
    // Import Design and Device form DCP
    design = Design.readCheckpoint(dcpInFile);
    device = design.getDevice();
    List<Cell> deSrcCells = new ArrayList<Cell>(design.getCell());
    Iterator<Cell> deCellIterator = deSrcCells.iterator();

    // Iterate over each Cell
    while (deCellIterator.hasNext()) {
        deSrcCell = deCellIterator.next();
        // Continue if it is part of a delay element, or if it is the last delay element
        if (!deSrcCell.getName().matches(".*monitor_0/U0/GEN\[\[.*\] \[.*\] \[.*\]"));
            continue;
        // Get deIndex from deSrcCell's name
        deIndex = Integer.parseInt(deSrcCell.getName().replaceFirst(".*monitor_0/U0/GEN\[\[.*\] \[.*\] \[.*\]"), "").replaceFirst("\[\[.*\] \[.*\]"));
        // Continue if it is the last DE element
        if (deIndex >= 48) continue;
        // Get ready for manipulating the design
        deParentCell = deSrcCell.getParentCell();
        deNet = design.getNet(deSrcCell.getSiteInst().getSiteWireMap().get("A").getName());
        deNet.connect(deSrcCell, "O");
        deNet.connect(getDestCell(), "I0");
        List<Tile> tileList = getAllowedTiles();
        RouteNode startRN = deNet.getSource().getRouteNode();
        RouteNode endRN = deNet.getSinkPins().get(0).getRouteNode();
        destinationTile = getTileByCount(detourTiles);
        // get start and end RouteNode
        List<PIP> tbPIPs = findShortestRoute(startRN, endRN, tileList);
        deNet.getPIPs().addAll(tbPIPs);
        deNet.lockRouting();
        System.out.println(String.format("%.2f", percentage) + " DE Net: " + deNet.getName());

        String wiresUsed = "";
        for (PIP ppp : tbPIPs) {
            wiresUsed += (ppp.getStartWireName() + ", ");
        }
        // Export DCP
        design.writeCheckpoint("C:\\Users\\User\\Documents\\UT\\M12\\EXEC_1\\detourv3\" + timeStamp + "\".dcp");
        if (uu > 3) {
            uu++;
        }
    }
}

public static Site getCurrentSrcSite() {
    return deSrcCell.getSite();
public static Tile getCurrentSrcTile() {
    return getCurrentSrcSite().getTile();
}

// Returns the 6LUT BEL from a Site by its number (0 to 3)
public static BEL getLUTBELFromSite(Site site, int num) {
    return site.getBEL(prefixChars[num%4] + "6LUT");
}

public static Cell getDestCell() {
    return design.getCell(deSrcCell.getName().replaceFirst("GEN\DE\[\" + deIndex + "\]", "GEN\DE\[" + (deIndex + 1) + "]")
}

// return tiles used in a route
public static List<Tile> getRouteNodePathTiles(RouteNode nn) {
    List<PIP> pips = nn.getPIPsBackToSource();
    List<Tile> tl = new ArrayList<Tile>();
    Iterator<PIP> pipIterator = pips.iterator();

    while (pipIterator.hasNext()) {
        PIP pip = pipIterator.next();
        Tile pipTile = pip.getTile();
        if (!tl.contains(pipTile)) {
            tl.add(pipTile);
        }
    }

    return tl;
}

// return tiles that are allowed to route the net through
public static List<Tile> getAllowedTiles() {
    int tileRow = deSrcCell.getSite().getTile().getRow();
    int tileColumn = deSrcCell.getSite().getTile().getColumn();

    List<Tile> allowedTiles = new ArrayList<Tile>();
    allowedTiles.addAll(Arrays.asList(device.getTiles()[tileRow]));

    // Only allow Tiles to the right:
    Iterator<Tile> allowedTilesIt = allowedTiles.iterator();
    while (allowedTilesIt.hasNext()) {
        Tile tileIt = allowedTilesIt.next();
        if (tileIt.getColumn() < tileColumn) {
            allowedTilesIt.remove();
        }
    }

    // Also allow the two sites needed to route the next DE
    Tile destinationTile = getDestCell().getSite().getTile();
    allowedTiles.add(destinationTile);
    allowedTiles.add(device.getTile(destinationTile.getRow(), destinationTile.getColumn() + 1));

    return allowedTiles;
}

// function to find a wire in a wire and route it
public static List<PIP> findShortestRoute(RouteNode start, RouteNode end, List<Tile> tiles) {
    List<PIP> output = new ArrayList<PIP>();
    int bestSolution = 20000000;

    for (int i = 0; i < destinationTile.getWireCount(); i++) {
        RouteNode interNode = new RouteNode(destinationTile, i);

        List<PIP> o = findRoutingPathConstrained(start, interNode, tiles);
        if (o == null) continue;
        List<PIP> a = findRoutingPathConstrained(interNode, end, tiles);
        if (a == null) continue;

        if (o.size() + a.size() < bestSolution) {
            bestSolution = o.size();
            output.clear();
            output.addAll(o);
            output.addAll(a);
        }
    }

    return output;
}
B.2 Fan-Out Routing Program

Below is the written Java program that uses RapidWright in order to add additional BELs to a net. It also attempts to place them manually. However, this was not implemented properly and the program fails to do this correctly.

```java
import java.util.ArrayList;
import java.util.List;
import java.util.Iterator;

import com.xilinx.rapidwright.device.BEL;
import com.xilinx.rapidwright.device.Site;
import com.xilinx.rapidwright.device.Tile;
import com.xilinx.rapidwright.design.Unisim;
import com.xilinx.rapidwright.design.Net;
import com.xilinx.rapidwright.design.Cell;
import com.xilinx.rapidwright.design.Design;
import com.xilinx.rapidwright.device.Device;
import com.xilinx.rapidwright.device.Site;
import com.xilinx.rapidwright.device.Tile;
import com.xilinx.rapidwright.device.Device;
import com.xilinx.rapidwright.device.Tile;
import com.xilinx.rapidwright.device.Device;
import com.xilinx.rapidwright.device.Tile;
import com.xilinx.rapidwright.device.Device;
import com.xilinx.rapidwright.device.Tile;
import com.xilinx.rapidwright.device.Device;
import com.xilinx.rapidwright.device.Tile;
import com.xilinx.rapidwright.device.Device;
import com.xilinx.rapidwright.device.Tile;
import com.xilinx.rapidwright.device.Device;
import com.xilinx.rapidwright.device.Tile;
import com.xilinx.rapidwright.device.Device;
import com.xilinx.rapidwright.device.Tile;
import com.xilinx.rapidwright.device.Device;
import com.xilinx.rapidwright.device.Tile;
import com.xilinx.rapidwright.device.Device;
import com.xilinx.rapidwright.device.Tile;
import com.xilinx.rapidwright.device.Device;
import com.xilinx.rapidwright.device.Tile;
import com.xilinx.rapidwright.device.Device;
import com.xilinx.rapidwright.device.Tile;
import com.xilinx.rapidwright.device.Device;
import com.xilinx.rapidwright.device.Tile;
import com.xilinx.rapidwright.device.Device;
import com.xilinx.rapidwright.device.Tile;
import com.xilinx.rapidwright.device.Device;
import com.xilinx.rapidwright.device.Tile;
import com.xilinx.rapidwright.device.Device;
import com.xilinx.rapidwright.device.Tile;
import com.xilinx.rapidwright.device.Device;
import com.xilinx.rapidwright.device.Tile;
import com.xilinx.rapidwright.device.Device;
import com.xilinx.rapidwright.device.Tile;
import com.xilinx.rapidwright.device.Device;
import com.xilinx.rapidwright.device.Tile;
import com.xilinx.rapidwright.device.Device;
import com.xilinx.rapidwright.device.Tile;
import com.xilinx.rapidwright.device.Device;
import com.xilinx.rapidwright.device.Tile;
import com.xilinx.rapidwright.device.Device;
import com.xilinx.rapidwright.device.Tile;
import com.xilinx.rapidwright.device.Device;
import com.xilinx.rapidwright.device.Tile;
import com.xilinx.rapidwright.device.Device;
import com.xilinx.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilinx.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilih.rapidwright.device.Tile;
import com.xilih.rapidwright.device.Device;
import com.xilinx.rapidwright.edif.EDIFCell;
import com.xilinx.rapidwright.edif.EDIFCellInst;
import javafx.util.Pair;

public class FanoutRouter {
    // Constants
    private final static String[] lutPortNames = {"I0", "I1", "I2", "I3", "I4", "I5");
    private final static String[] prefixChars = {"A", "B", "C", "D");

    private final static String dcplnFile = "C:\\Users\\User\\Documents\\UT\\M12\\EXEC1\\Project_partially_placed\\checkpoint.dcp";

    private final static int fanoutAmount = 400;

    public static void main(String[] args) {
        // Import Design and Device from DCP
        design = Design.readCheckpoint(dcplnFile);
        List<Cell> deSrcCells = new ArrayList<Cell>(design.getCells());
        Iterator<Cell> deCellIterator = deSrcCells.iterator();

        // Iterate over each Cell
        while(deCellIterator.hasNext()) {
            deSrcCell = deCellIterator.next();

            // Continue if it is part of a delay element, or if it is the last delay element
            if (!deSrcCell.getName().matches(".*monitor\_0/\U0/GEN\_DE\[][]\.*\.*") ) continue;

            // Get deIndex from deSrcCell's name
            deIndex = Integer.parseInt(deSrcCell.getName().replaceFirst(".*monitor\_0/\U0/GEN\_DE\[][]\.*\.*", "").replaceFirst("[][]\.*", "));

            // Continue if it is the last DE element
            if (deIndex >= 48) continue;

            // Get ready for manipulating the design
            sinkCounter = 0;
            deParentCell = deSrcCell.getParentCell();
            deNet = design.getNet(deSrcCell.getCellInst().pNetSiteWireMap().get("A").getName());
            System.out.println("\nDE Net: " + deNet.getName());

            while (sinkCounter < fanoutAmount) {
                createSinkCell();
                sinkCounter++;
            }
        }

        // Export DCP
        design.writeCheckpoint("C:\\Users\\User\\Documents\\UT\\M12\\EXEC1\\pp\_fanout\_v1\\cp\_fanout\_new\_v1\_0.\" + fanoutAmount + ".dcp");
    }

    public static Site getCurrentSrcSite() {
        return deSrcCell.getCell();
    }

    public static Tile getCurrentSrcTile() {
        return getCurrentSrcSite().getCell();
    }

    // Returns a Tile that is positioned next to the source Tile
    public static Tile getTileByCount(int count) {
        // Find neighbors until there have been tileOffset amount of CLB tiles
        Tile outputTile = getCurrentSrcTile();
    }
}
```java
int i = 0;
while (i < count) {
    outputTile = outputTile.getTileNeighbor(1, 0);
    while (!outputTile.getTileNamePrefix().matches("CLB(M|L).*")) {
        outputTile = outputTile.getTileNeighbor(1, 0);
    }
    i++;
} return outputTile;

// Return a List of Pairs that contain a Site and a BEL that will be used as fanout
public static List<Pair<Site, BEL>> getSinkPairs() {
    List<Pair<Site, BEL>> outputList = new ArrayList<Pair<Site, BEL>>();
    int belCount = 0;
    while (outputList.size() < fanoutAmount) {
        Tile tile = getTileByCount(Math.floorDiv(belCount, 8)); // change to next tile every 8 BELs
        Site site = tile.getSites()[Math.floorDiv(belCount, 4)]; // change site every 4 BELs
        BEL newBEL = getLUTBELFromSite(site, belCount % 4);
        outputList.add(new Pair<>(site, newBEL));
    }
    return outputList;
}

// Returns the 6LUT BEL from a Site by its number (0 to 3)
public static BEL getLUTBELFromSite(Site site, int num) {
    return site.getBEL(prefixChars[num % 4] + " 6LUT");
}

public static void createSinkCell() {
    List<Pair<Site, BEL>> belList = getSinkPairs();
    // EDIF attempt:
    // Create Cell and connect nets
    EDIFCellInst sinkLutCellInst = new EDIFCellInst("DE" + deIndex + ".Sink." + sinkCounter, design.getNetlist().getHDIPrimitive(Unisim.LUT6), deParentCell);
    for (int i = 0; i < 6; i++) {
        deNet.getLogicalNet().createPortInst(lutPortNames[i], sinkLutCellInst);
    }
    // Place the cell
    // NOT WORKING
    Cell sinkLutCell = design.createCell("DE" + deIndex + ".Sink", sinkLutCellInst);
    Site nSite = belList.get(sinkCounter).getKey();
    BEL nBEL = belList.get(sinkCounter).getValue();
    design.placeCell(sinkLutCell, nSite, nBEL);
    // EDIF interface attempt:
    // Results in the same error als EDIF route
    // The following commented lines does (mostly) the same as the lines above but results in the same errors in Vivado
    /*
    Site toBePlacedSite = getTileFromSinkCount(2).getSites()[0];
    Cell sinkLutCell = design.createAndPlaceCell(deParentCell, "DE" + deIndex + ".Sink", Unisim.LUT6, toBePlacedSite + "/D6LUT");
    deNet.connect(sinkLutCell, "I0");
    */
}
```