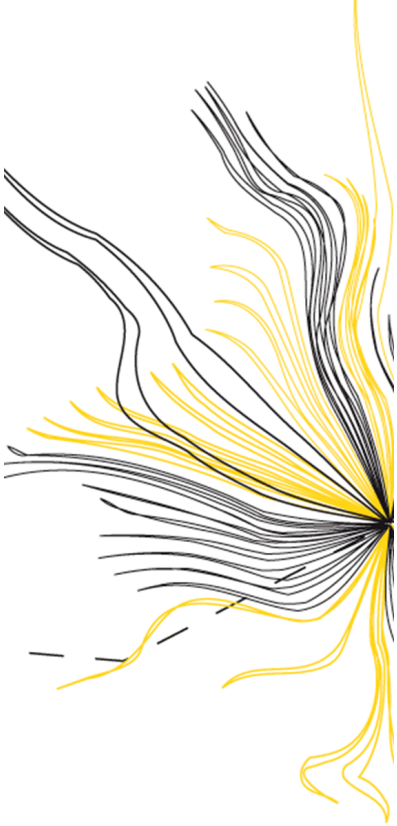
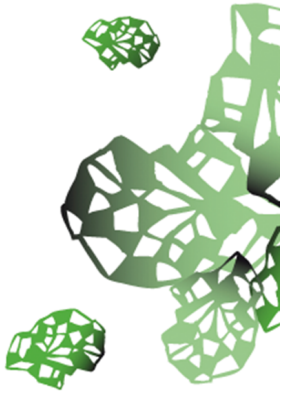


UNIVERSITY OF TWENTE.

**Faculty of Electrical Engineering,
Mathematics & Computer Science**

Bitstream Manipulation to Support Relocation and LUT Adaptation

Hessel den Hertog
BSc Thesis
s1698354



Supervisors:
dr. ing. D. M. Ziener
prof. dr. ir. R. N. J. Veldhuis
Madiha Sheikh

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

Contents

1 Introduction

Field Programmable Gate Arrays or for short "FPGAs" are semiconductor devices that are based around a matrix of *Configurable Logic Blocks* (CLBs) connected via programmable interconnects. Nowadays with a method called partial reconfiguration, certain areas on the FPGAs can be reconfigured or modules can be interchanged on the board. Partial reconfiguration dynamically modifies prebuild design modules during runtime without interrupting the other logic on the board. In this way a lot of power can be saved and area utilization can be optimized (1). For instance with partial reconfiguration one can interchange modules within a certain reconfigurable area without implementing both of them and therefore reducing the used up area by the modules.

Another way to reconfigure the design would be through bitstream manipulation. The composition of the bitstream will be more elaborated upon in Section 2.2. If the designer has a deep understanding of the bitstream and what it represents it is possible to create for instance dummy *Look Up Tables* (LUTs) which might confuse any possible attackers. By placing extra LUTs, with contents but without functionality, in the bitstream, the attacker might lose track of the actual design. The dummy LUTs work as a distraction. Nowadays, adversaries can have many motivations for obtaining or manipulating a configuration bitstream. FPGAs are already widely used in a variety of branches. For instance medical, aviation and industrial all use devices in which FPGAs are often part of cyber-physical systems (2).

With bitstream manipulation one can also change the contents of the LUTs to achieve a different functionality of the design. A big advantage of bitstream manipulation is being able to change the design without having to rerun all the original steps of the FPGA design flow. This would save the user time when designing a project. This design flow is further discussed in Section 2.1. This thesis will present a program that can manipulate any LUT on the fabric of the development board used in this work.

So the main advantage of LUT adaptation during runtime is that it would save a lot of time since all the necessary design steps have to be done once, and then the bitstream can be manipulated. For small changes only the bits would have to be modified and the stream has to be uploaded again to the FPGA board. Also, with bitstream manipulation, relocation should be possible to realize. Ideally this has to be done during runtime.
This leads to the following goal for this research:

The goal of this thesis is to design a program that allows bitstream manipulation in order to achieve LUT adaptation and to support relocation of reconfigurable modules in a created design.

The reason to explore the field of bitstream manipulation is to get a better view of the bitstream contents. The LUT adaptation/manipulation can be seen as an attack on the bitstream. The contents will be changed in order to modify/damage the initial design. Meanwhile, the relocation of the module can be seen as a countermeasure to prevent this attack. The contents of the LUTs will be shifted to another location on the fabric, so any

(sensitive) LUTs targeted by an attacker will not remain static on the fabric.

The remainder of this paper is structured as follows: Section 2 will talk about theoretical background, Section 3 describes the concepts that will be implemented in this research. Section 4 talks about the toolflow used for these concepts. Section 5 will discuss the implementation. The report will end with Section 6 which is about testing and evaluation. The conclusion is in Section 7.

2 Theoretical Background

This section will talk about the flow that is used when creating a FPGA-based design. Furthermore, the section will go more in depth in the theory of certain aspects that will be used throughout this research. First, the architecture of FPGA is briefly mentioned. Then a few essential bitstream words are highlighted, followed by a short section about module relocation. The section ends with a small overview of the configuration interfaces.

2.1 FPGA Design Flow

As stated in the introduction this report will talk about bitstream manipulation. In order to create a bitstream, the program Vivado IDE version 2018.3 (see Section 4.1) is used. In Figure 1, one can see a general overview of the FPGA design flow. The designer starts off with creating a circuit in *Hardware Description Language* (HDL). This can be done by using the *Block Diagram* window in the design suite to add and connect different *IPs* (Intellectual Property) and use the Vivado wrapper to generate a HDL code that represents the circuit in the Block Diagram window. Another option would be to start with an empty file and write their own program in HDL code. Furthermore, a constraints file has to be created to ensure correct pin connections between the ports in the HDL code and the actual I/O options on the hardware that will be used in this work.

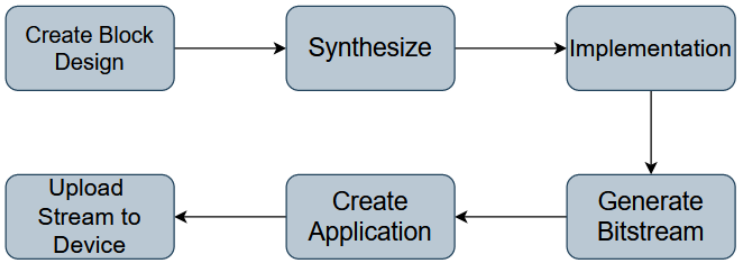


Figure 1: General overview of the FPGA design flow

After the design is validated, the next step is to synthesize the design. The code that is created in the first step is, during synthesis, converted to a netlist of LUTs and FlipFlops. Following the flow from Figure 1, the next step would be implementation. This will take care of placement and routing of the design. After the implementation step is finished, one is able to have a look at the implemented design. Here, final constraints changes can be made before going to the next step, which is generating the bitstream. The bitstream composition will be discussed later on in Section 2.2. In order to create an application project the hardware is exported and the Xilinx SDK is launched (see Section 4.2). The design flow ends with uploading the bitstream and application to the FPGA.

2.2 Bitstream Configuration

Nowadays, most devices have a combination of a *processing system* (PS) and *programmable logic* (PL). In order to let the PS communicate with the PL, an *Advanced eXtensible Interface* (AXI) is used. A schematic overview can be seen in Figure 2. The user can write software which can be run on the processing system. Via the AXI interface, the configuration frames in the PL can be changed.

The architecture of the PL is arranged in columns. These columns are called resource columns and contain only 1 type of resource. The resources are for instance *Block Random Access Memory* (BRAM) blocks, *Digital Signal Processors* (DSPs) and CLBs. The CLBs on the programmable logic consists of 2 different slices, named Slice-M and Slice-L. These slices contain *Basic Elements of Logic* (BELs), like LUTs and Flip Flops. The CLBs are subdivided in a number of frames. These frames contain the configuration words, which configure the elements in the CLB. These frames are the smallest addressable units on the PL configuration memory and are responsible for the configuration of both slices in the CLB. The configuration words in a bitstream contain the information to configure these frames. A comprehensive look on how the frames are configured can be found in Section 4.6.

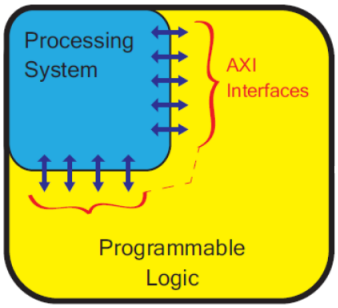


Figure 2: Basic overview of a development board (3)

Bitstream manipulation is a rather simplified approach for changing implemented hardware. All the information for configuring the FPGA is within the bitstream. So it is important to know the details of all the information that the bitstream holds. There is a user guide which has an extensive explanation of an example bitstream (4). There are different types of bitstreams. In this thesis, the *.bit file* and *.bin file* are used. The difference between these files is that the .bit files contain a header in the bitstreams, whereas the .bin files do not have this header.

There are a few bit sequences mentioned in this user guide that are important for this project, which will be briefly discussed. First, there is the configuration data word 0x30002001, which indicates a write to the *Frame Address Register* (FAR). The FAR register holds the address at which the first *Frame Data Register Input* (FDRI) word is written. A typical bitstream starts writing the first FDRI word at address 0 and auto-increments to

the final count. If one is able to change the FAR register one can change the location to where the design is loaded to the device. The FAR register is described according to the table in Figure 3 (4).

Address Type	Bit Index	Description
Block Type	[25:23]	Valid block types are CLB, I/O, CLK (000), block RAM content (001), and CFG_CLB (010). A normal bitstream does not include type 011.
Top/Bottom Bit	22	Select between top-half rows (0) and bottom-half rows (1).
Row Address	[21:17]	Selects the current row. The row addresses increment from center to top and then reset and increment from center to bottom.
Column Address	[16:7]	Selects a major column, such as a column of CLBs. Column addresses start at 0 on the left and increase to the right.
Minor Address	[6:0]	Selects a frame within a major column.

Figure 3: Description of the FAR (4)

Secondly, there is the configuration word 0x30004000 which indicates the start of writing the FDRI words. After word 0x30004000 follows a word that indicates the total amount of configuration words that are loaded into the registers. This value is especially convenient to check whether the program processes the configuration words correctly.

2.3 Module Relocation

There are a couple of aspects that have to be considered when one wants to relocate a module to another part of the FPGA in terms of available resources. These aspects are summarized by the term 'footprint'(5). A footprint of a module has to match with the available resources at the target area on the board. There are different type of footprints which will shortly be elaborated. The *Resource footprint* spans the resource blocks that are used by the module. These are the CLB, DSP and BRAM columns used by the module. Next is the *Wire Footprint* which deals with the routing resources. The *Timing Footprint* also has to be considered. The clock signals that are used on the Xilinx FPGA boards run in horizontal lines and then split up in vertical splines that go up and down. If a module is shifted from a downwards spline to an upwards spline it could affect the timing as well. Furthermore, if a module is relocated on the FPGA it can be possible that this relocating results in longer routing delays at some points in the fabric (6).

It is important that the static design does not route any wires through the partial areas. In order to make sure that this does not happen, a blocker macro can be used, which occupies all the wires that are within the partial areas. By doing so, the routing of the static design is forced to go around the partial area. This way the signals belonging to the static design are prevented from going through the partial areas. In order to have an interface with the

areas, tunnel wires are used. Tunnel wires are wires that are unblocked and form a path for each signal to its corresponding connection primitive. These connection primitives are used to anchor signals that aren't connected yet. These signals, in this case, are the input and output signals of the module.(7). The connection primitives, tunnels and blocker are more extensively discussed in Section 5.2.2.

2.4 Configuration Interfaces

There are different ways to configure the frames on the PL. In this section a few techniques will briefly be discussed.

JTAG cable
JTAG, named after the *Joint Test Action Group* who initialized the IEEE 1149.1 standard, is a method that configures the PL via the use of a cable. This JTAG cable is directly connected between the hardware and a host computer. In this way, the configuration data can directly be downloaded. This interface is a very straightforward way of configuring the FPGA.

FSBL
Another method to configure the programmable logic is with the usage of a *First Stage Boot Loader* (FSBL). With this technique, the configuration data is directly uploaded to the memory when the hardware device is powered on. In this way, no extra cables from the host computer have to be attached to the hardware. Another advantage of this method is that user code can simultaneously be uploaded to the PS.

PCAP
PS-processor Configuration Access Port, or for short PCAP, is yet another way to configure the FPGA. With PCAP, the PL can be configured with data using the PCAP bridge (8). This bridge provides access to the PL configuration module in software. This is what makes this configuration technique very interesting. However the PCAP can only configure the frames with binary files (9).

One has to be aware that the interfaces to the programmable logic are mutually exclusive. One has to be careful when switching between these interfaces. All outstanding transactions have to be completed before the interfaces are switched. (8).

3 Concept

As mentioned in the Introduction, there are two main research goals of this thesis. Namely **the adaptation of a look-up table** and **the relocation of a module**. In this section, these two goals will be elaborated upon.

3.1 LUT Adaptation

Since the LUT is one of the basic elements of logic within a slice in the programmable logic, it is important to understand how these elements are represented in the configuration data (more on this later in Section 4.6). With LUTs, all kinds of combinational logic circuits can be realised. Simple logic gates like AND and OR gates can be implemented but also more elaborate truth tables, which can be combined to make even bigger designs. If it is possible to change the contents of one or more of this small basic elements, the results can be catastrophic. When, for illustration, adversaries manage to disrupt the outcome of a truthtable, the design can already malfunction by merely changing a few logic gates to an AND or an OR.

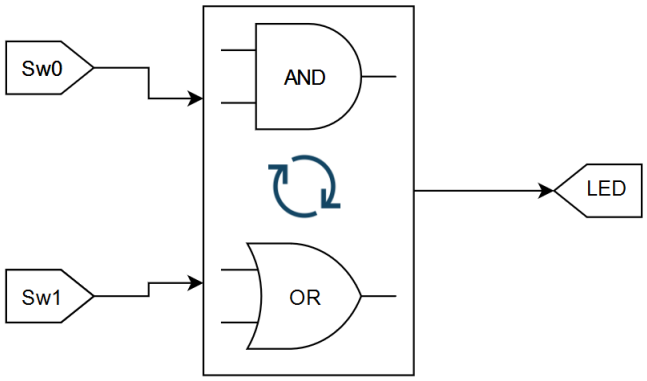


Figure 4: Concept for look-up table adaptation

The concept for this LUT adaptation is to make a meaningful change in the bitstream, in order to change the functionality of this LUT. With meaningful is meant that the design should have a working outcome and that we don't just simply break it by modifing the bits. For simplicity reasons, this research shows a change from an AND to an OR gate (see Figure 4). In order to achieve this goal, one must have a deep understand of the FPGA architecture. Questions like, how to find the LUT contents in the bitstream, or, how do we interpret the contents of the bistream, have to be answered. Also, in order to investigate these certain aspects, a bitstream has to be generated. A simple AND gate design will be build in a design environment, in which small changes in one specific LUT can be examined. In order to show that the LUT functionality has changed, onboard switches and LEDs are used to have visual proof. The steps taken to reach this goal, will be eleborated upon in Section 4

3.2 Module Relocation

In order to relocate a design or a reconfigurable module, one should again take a closer look at the FAR register. The FAR address indicates where the first FDRI word is written in the configuration memory. So in order to relocate the design, the FAR address should be changed to shift the design. For this project the choice is made to floorplan two reconfigurable areas above eachother. Furthermore, both the recongifurable areas span a height of 1 clockregion. This is beneficial since this makes it possible to apply certain properties for these areas. Another choice that is beneficial during implementation is that the two areas span the same horizontal X coordinates on the development board. This is very convenient since the number of available resource columns will be the same.

Reconfigurable areas are vertically divided into two slots. We call the left slot West and the right slot East. For this goal it is desired to move a module from the West slot of reconfigurable area 1 to the West slot of reconfigurable area 2, which resides below area 1. This is illustrated in Figure 5

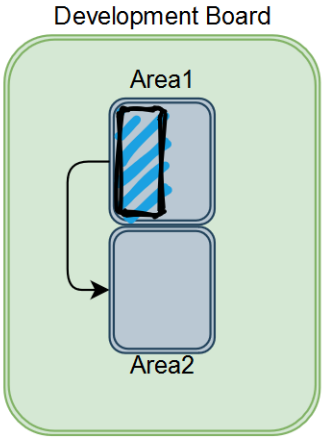


Figure 5: Relocation from Area 1 to Area 2

In order to show that the module actually has moved from one area to the other, the output of the areas will be attached to different LEDs. Area 1 will occupy LEDs 0 to 3 whilst area 2 will use LEDs 4 to 7. This is in order to show that the relocation actually happened. The module will contain logic resembling an adder function and will use the onboard switches as input.

4 Toolflow

Now that the two goals have been set and discussed, an implementation should be found in order to reach these goals. This section will cover this part. First it will elaborate which design environment and tools are used for this research. It will continue with the chosen development board and will end with a comprehensive view of the FPGA fabric.

4.1 Vivado Design Suite

The design environment that is used for this project, is Vivado IDE version 2018.3. Vivado is able to perform all the design flow steps in an organised way. Vivado has different options for creating a design. It has both a graphical and a TCL interface. One can create a block diagram, validate it and let Vivado manage wrapper and auto update. This creates, depending on the user settings, a Verilog or VHDL code. Furthermore, it allows for easy pin mapping between the logical and physical pins and supports additional bitstream settings. This means that for instance a binary file or a logic location file can be generated as well, when generating a bitstream. Furthermore, it is easy to relocate LUTs to another part of the fabric in the graphical interface of the implemented design in Vivado. After this, it is possible to regenerate a bitstream to see what has changed within the bitstream.

4.2 Xilinx Software Development Kit

As stated in the design flow section, the hardware of the project including the bitstream is exported. After this step the Xilinx *Software Development Kit* (SDK) tool is launched. This can be launched from within the Vivado IDE and creates SDK files local to the Vivado project directory, being for instance the files used for initialization of the hardware platform. In Xilinx SDK, one is able to create a number of different design applications. Furthermore a FSBL application can be made. The FSBL is needed in order to create a BOOT.bin file. If the BOOT.bin file is copied to a SD card and inserted into the board, it will configure the FPGA and will execute the program. In order to see how the program behaves, or for the purpose of debugging, there is a SDK terminal that will print statements (if the designed program contains any) if the UART cable is connected between the host computer and the development board.

4.2.1 FatFs Library

When an application is created in Xilinx SDK, a *Board Support Package* (BSP) is generated. It is possible to modify the settings of this package and a designer can chose to add certain libraries to this package. The library which is added for this project is the *Generic Fat File System Library* (10). Herinafter we call this library FatFs. The main functionality of this library is to operate as an application interface between the application and a storage element. This storage element can for example be a SD-card. This functionality is needed in order to load a bitstream from the SD-card onto the device. The library also contains

functions for read and write operations of the files that reside on the SD card. This is especially useful when it is desired to change certain elements in the bitstream and save it under a different name. The original bitstream will still be intact.

4.2.2 Device Configuration Interface

Whereas the FatFs library takes care of the files on the SD card, the actual frames on the device still have to be configured. For that the Device Configuration Interface will be utilized. This configuration interface is already part of the BSP when an application is created. This interface has three main functionalities, being [1] AXI-PCAP, [2] Security policies and [3] XADC. The functionality that will be used for this research is the AXI-PCAP. This AXI-PCAP is used to configure the PL. The configuration can be initiated from within a software application created in the Xilinx SDK environment.

4.3 BitMan Tool

Changing the bitstream at the correct places and in the right manners can result in a meaningful change in hardware. Earlier work has been done in this field of manipulating the bitstream. One tool that is extremely helpful for getting a better insight in the bitstream and how it behaves after a hardware change is made in the graphical interface of Vivado, is the BitMan tool created by Khoa Dang Pham (11). This tool works via the command prompt and allows for certain operation options like merging, relocating, copying and replacing parts of the bitstream. Furthermore it contains an option to look at the contents of all the CLBs that are occupied on the device. In Section 4.6, the relation between the bitstream and the CLBs is dicussed more intensively. One more function of BitMan is the cut option. This feature is especially handy when doing module relocation. The function can cut out the part of the bitstream which is merely responsible for configuring a module that is occupying a certain area on the FPGA.

4.4 GoAhead Tool

In order to relocate modules on the FPGA there have to be certain reconfigurable areas (islands) present on the FPGA board. A tool that comes in handy at this point is the GoAhead Tool (5). This tool allows for creation of both reconfigurable areas as well as individual modules by selecting proper X-Y coordinates on the FPGA. This can be done via either single commands, or a .goa script, which contains a list of commands that eventually build the design. With GoAhead, the hardware description language files and tcl scripts necessary for Vivado IDE, can be generated. Furthermore, it can generate blocker wires which ensure proper routing between the module and the static design. The general flow used for the relocation part of the thesis can be seen in Figure 6.

First, the areas and modules are floorplanned in GoAhead, which are accompanied by the tcl scripts needed for Vivado to set the right properties and connections. Next step is to generate the VHD files for implementing the floorplanned regions on the FPGA. The

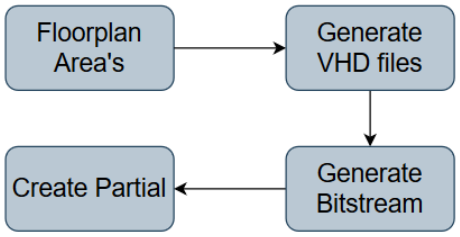


Figure 6: Design flow with the GoAhead tool

following step will be to generate a bitstream for the implemented areas in the Vivado. For the modules, again BitMan(11) can be used to manipulate the bitstream. The bitstream selection, responsible for configuring the resources within a area, has to be cut out, after which the partial bitstream can be used for reconfiguration of that area. This is the final step of the GoAhead Design Flow.

4.5 Zedboard

In order to upload a bitstream containing the implemented hardware, or to test a program written in Xilinx SDK, a development board is needed. The development board that is used for this project is the Zedboard (12). This is a physical board that has multiple I/O options, like LEDs and switches, that can be used by the designer for showing a functionality of an application or for debugging purposes. Furthermore it has a SD card slot. This slot will be

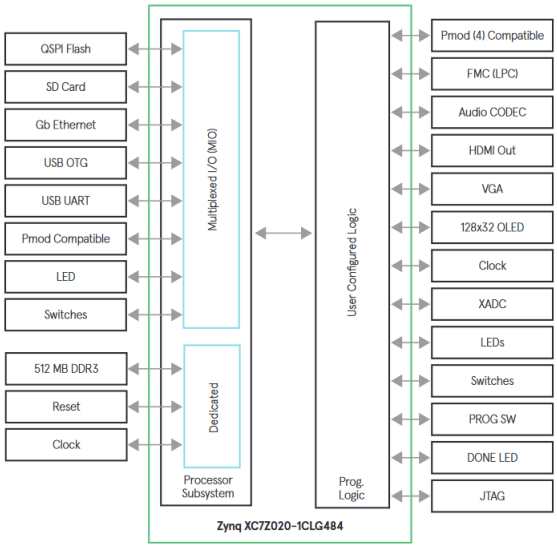


Figure 7: A schematic overview of the interfaces of the Zedboard (12)

used to get the bitstreams, pregenerated by Vivado and/or BitMan, on the board in order to manipulate them via an application created in the Xilinx SDK. In Figure 7 one can see an overview of the Zedboard and the interfaces present on the board.

4.6 Coarse Granularity

When taking another glance at Figure 3 that shows the information in the FAR register, one can see that the bits [25:23] indicate a certain block type. All these blocktypes are placed as resource columns on the FGPA as can be seen in Figure 8. Each resource column has a number of tiles. A clockregion spans 50 tiles containing CLBs or 5 tiles with BRAM/DSP blocks.

For this project we are mainly interested in the blocktype CLB since it is desired to change the LUT contents within those CLBs to achieve different functionality. Bit 22 indicates the top/bottom half of the fabric. In the top half (Top = 0) reside the clock regions X1Y2 and X2Y2. The other clockregions are part of the bottom half (Top = 1).

Bit [21:17] select the clock row. The top half of the Zedboard only has 1 clock row (Y2), so there is no selection needed. However at the bottom part are two possiblities (Y0 or Y1) (see Figure 8). The bits [16:7] select a major resource column within a clock row. For the continuation of this report, this column coordinate will be called X_T. As indicated before there are multiple resources columns.

Then last but not least there are the bits [6:0] which will select a minor address within a major resource column. These minor addresses are called frames.

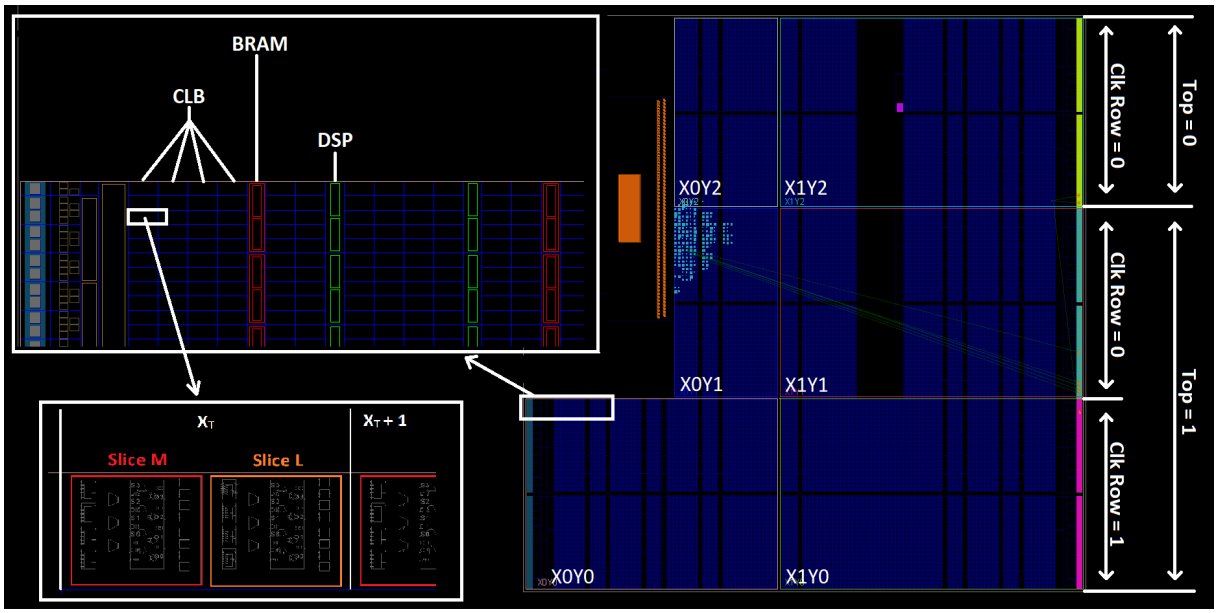


Figure 8: Resource columns,Top and Clk rows on the Zedboard

All frames in Xilinx 7 series FPGAs devices have a fixed, identical length of 3232 bits (101 words of 32 bits) in the bitstream. To get a better understanding of how the frames

relate to the CLB we will take a closer look at a few CLBs. CLBs have 35 minor addresses (13). A CLB consists of two slices (M or L). Each slice contains 4 LUTs, 3 Muxes, 8 Flip Flops and a Carry Chain. Each frame can only contain information for 1 type of element in the slice, but more frames can have information for the same type of element. For instance the contents in the LUTs in slice M are represented in the Frames 31-34.

The height of one clockregion spans 101 configuration words of 32 bits each. This results in a total height of 3232 words, which is equal to 1 frame. In the middle of every clockregion resides a clock lane. This clock lane is configured by configuration word 50 in the frame, leaving words 0-49 and 51-101 for configuring the CLB's.

Figure 9 shows how the configuration of the CLBs is done. 2 words are needed to configure 1 CLB. Since the height of one clockregion is 50 CLBs we have (2 x 50 configuration words + (word clock lane)) = 101 words, which matches the amount of words within 1 frame.

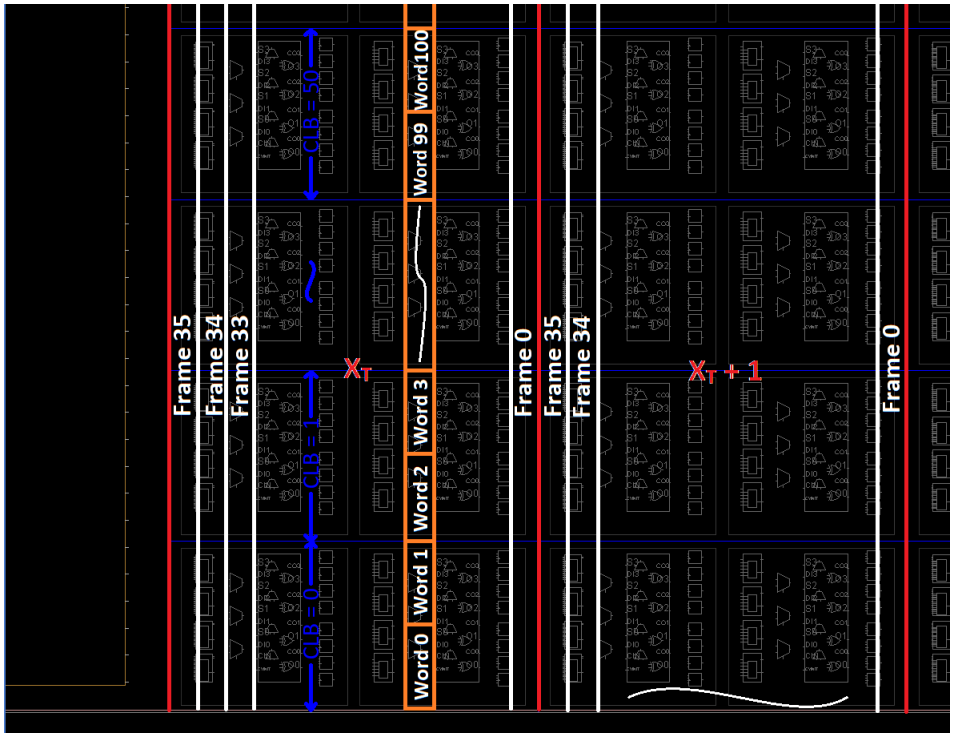


Figure 9: Configuration of CLBs with frame words

To summarize the above: The CLBs are configured using 2 words from 35 frames each. Frames 31-34 are responsible for the LUTs in slice M whilst frames 25-28 are responsible for the LUTs in slice L.

The contents of a single LUT comprises 64 bits which are configured as shown in Figure 10. The first 16 bits [15:0] of LUT A are configured by the first 16 bits of word K(also called halfword K). Then the first 16 bits of LUT B are configured by the last 16 bits of 32 bit word K in frame 31. We move on to the next word K + 1 which in turn will configure the

first 16 bits of LUTs C and D. Now the first 16 bits of all the LUTs in slice M of one CLB have been configured. Words K + 2 and K + 3 will configure the LUTs in the next CLB and so on untill words K + 99 and K + 100 configure the first 16 bits of the LUTs in the CLB at the top of the clock region. Then we move on to frame 32 which configures the LUT bits [31:16] and so on till frame 34 when all the 64 bits, of all the LUTs in slice M, of all the CLBs, in column X_T are configured. Knowledge on how these LUTs are configured is very important when the right elements have to be extracted from the bitstream.

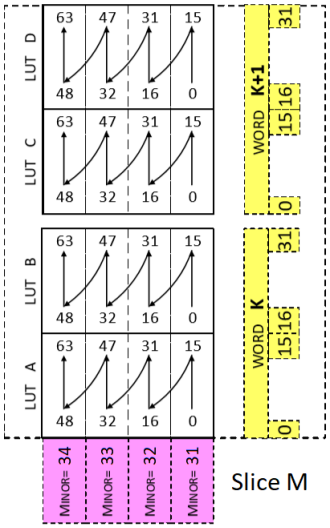


Figure 10: Configuration of LUTs in Slice M of a CLB

5 Implementation

5.1 Changing LUT

As is known, the bitstream consists of bits that can be represented in various ways. Common ways to display a bitstream are in binary form or in hexadecimal form. When handling large bitstreams the hexadecimal representation would give more insight in what is happening in the bitstream. For the purpose of debugging the bitstream, a simple block diagram is build in the Vivado IDE containing an AND gate structure (see Figure 11).

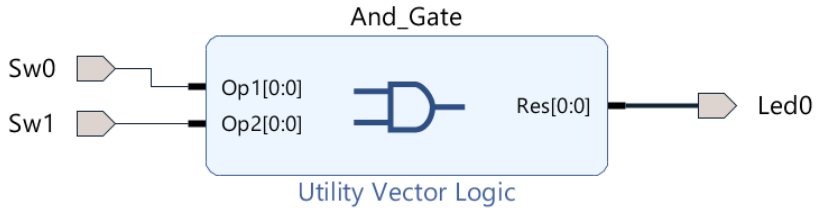


Figure 11: Vivado block diagram of an AND gate configuration

For the logic of the AND gate, the IP "Utility Vector Logic" is used which allows the user to combine 2 inputs, do an operation on the inputs and pass the outcome of the operation to the output. For the inputs and the output, separate ports are created and are connected accordingly to the Utility vector logic IP. The inputs are the switches and the output is a LED.

Following the FPGA design flow, the design is saved, a HDL wrapper is created and the design is synthesized and implemented. After implementation is finished, the design is opened and the netlist appears. The input and output ports are initially connected to peripheral modules but it is desired to use the on-board switches and leds. Using the master XDC of the Zedboard (14) the pins are correctly configured. Furthermore, the connection to the pins of the LUT should be fixed (locked). Locking these pins can be done in the Vivado GUI. This is needed in order to be able to have a consistent bit sequence when changing the functionality of the LUT. Vivado might change the mapping between the logical pins and the physical pins during the implementation phase. This would make things harder when comparing the bitstream of the AND configuration with other bitstreams, since a difference in pin mapping already results in a different bitstream.

When the ports are set and the pins are locked, the bitstream can be generated. Now we have a bitstream to work with. In order to discover what bits represent the AND gate, the BitMan option which prints the CLB info is run. This shows the user the contents of the frames in the CLBs and in which columns this contents is located. The CLB location can be found by looking at the implemented design in the Vivado GUI (see Figure 12) and since it is known which frames are responsible for configuring the LUT (Section 4.6), the bits that configure the LUT can be found. These bits appear to be 0000 5555 5555 0000 for the AND gate as can be seen in Figure 13a.

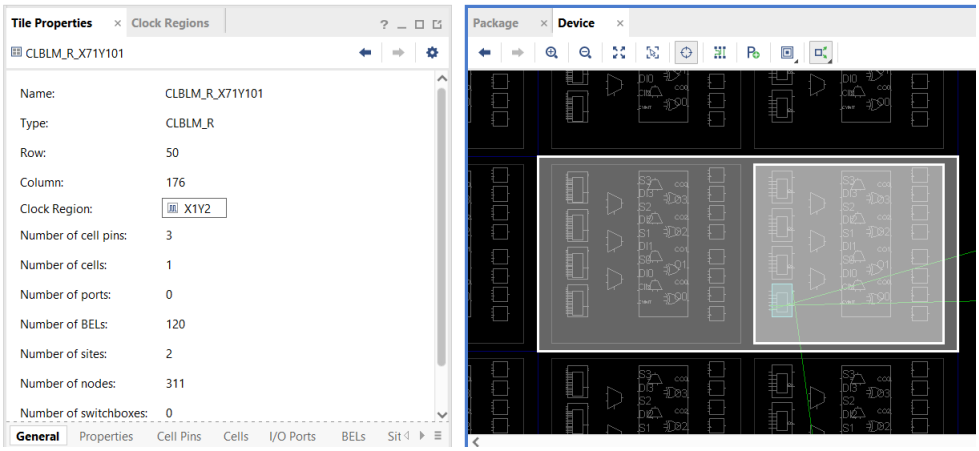


Figure 12: LUT in Slice-L of the configured CLB located at X71_Y101. BitMan however displays this at X71_Y51

```
C:\Users\Hesse\VivadoProjects\AndGate_blck
found Sync
XC7Z020
found a Zynq 7000 device

write 1010808 words of configuration data



First bitstream
Row Address 0
CLB col 71 row 51
Frame 01 :  00 00 80 00 00 00 00 00
Frame 05 :  00 00 40 00 00 00 00 00
Frame 16 :  00 00 00 00 00 02 00 00
Frame 17 :  01 00 00 00 00 00 00 00
Frame 22 :  02 00 00 00 00 02 00 00
Frame 23 :  02 00 00 00 00 02 00 00
Frame 24 :  02 00 00 00 00 02 00 00
Frame 26 :  00 00 55 55 00 00 00 00
Frame 27 :  00 00 55 55 00 00 00 00
CLB col 71 row 52
Frame 07 :  40 00 00 00 00 00 00 00
Frame 12 :  80 00 00 00 00 00 00 00
CLB col 71 row 53
Frame 09 :  00 00 80 00 00 00 00 00
Frame 14 :  00 00 80 00 00 00 00 00
```

(a) And LUT CLB configuration

```
C:\Users\Hesse\VivadoProjects\AndGate_blck
found Sync
XC7Z020
found a Zynq 7000 device

write 1010808 words of configuration data



First bitstream
Row Address 0
CLB col 71 row 51
Frame 01 :  00 00 80 00 00 00 00 00
Frame 05 :  00 00 40 00 00 00 00 00
Frame 16 :  00 00 00 00 00 02 00 00
Frame 17 :  01 00 00 00 00 00 00 00
Frame 22 :  02 00 00 00 00 02 00 00
Frame 23 :  02 00 00 00 00 02 00 00
Frame 24 :  02 00 00 00 00 02 00 00
Frame 25 :  00 00 55 55 00 00 00 00
Frame 26 :  00 00 FF FF 00 00 00 00
Frame 27 :  00 00 FF FF 00 00 00 00
Frame 28 :  00 00 55 55 00 00 00 00
CLB col 71 row 52
Frame 07 :  40 00 00 00 00 00 00 00
Frame 12 :  80 00 00 00 00 00 00 00
CLB col 71 row 53
Frame 09 :  00 00 80 00 00 00 00 00
Frame 14 :  00 00 80 00 00 00 00 00
```

(b) Or LUT CLB configuration

Figure 13: CLB configurations obtained via BitMan. Bitman omits LUTs that only contain zeros

In order to know how to change the bits to get the LUT to behave as another logic function, the same steps are taken but instead the Utility Vector Logic IP is configured as an OR gate. Again the bitstream is generated and BitMan is used to look at the CLB contents. It is found that the LUT contents of an OR is represented in 5555 FFFF FFFF 5555 as can be seen in Figure 13b.

5.2 Implementing the VHD Files from GoAhead

As stated in Section 4.4, GoAhead can be used to floorplan areas and modules on the board using commands. The coordinate span for reconfigurable area 1 is set to (34,50) for bottom left and (43,99) for top right. This creates a rectangle somewhere in the middle of the ZedBoard in clock region Y1. The same size is applied to area 2 but the only difference is that this area resides in clock region Y0. These regions are chosen to be here because the same resource columns ran through both areas.

For the module the parameters for the area dimensions are set differently. Recall from Section 3.2 the areas are vertically divided into 2 slots. Since the concept is to use the West slot, the following coordinates are established. (34,50) for bottem left and (38,99) for top right. This results in half the size of the reconfigurable areas.

After the areas and modules have been floorplanned in GoAhead using the commands, the next step would be to generate the design in Vivado using the tcl scripts that are also generated by GoAhead. For this project the static design and the module are implemented in a separate Vivado project to maintain a clear overview.

5.2.1 Static Design

After the placement and interface constraints have been loaded into Vivado, the design looks like Figure 14. Both the partial areas are visible in the Vivado device overview.

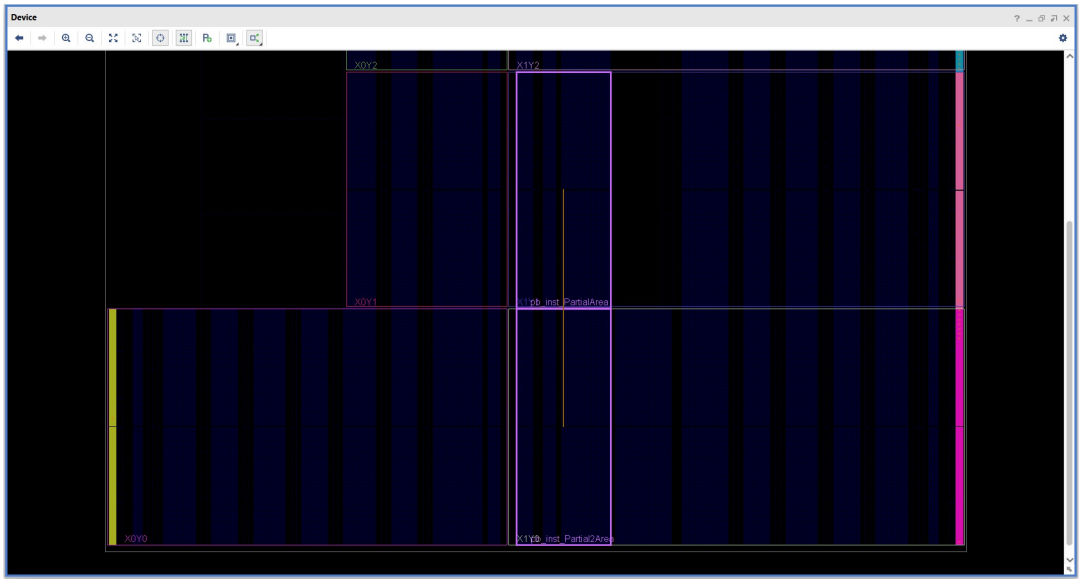


Figure 14: Partial Area 1 and 2 implemented in Vivado

In Figure 14 one can see a brown line that connects the two reconfigurable areas. This line represents bundle wires, meaning that certain nets are both used by area 1 as by area 2. These lines are just for clarification of the design. In case of the static design, these shared signals are the input switches for the adder program of the module. The switches are

shared since it is not possible to reconfigure I/O ports, these must therefore remain in the static region (1). The LEDs are not part of the bundle wire since it is desired to show the relocation of the module with different LED outputs per area.

Next up is the placement of the design, the connections of the clock pins and to insert the blocker. The design is routed and the blocker is removed again. After the blocker is removed, the resulting netlist can be seen in Figure 15. The green line coming from the left is the clock line that first goes into a buffer before it connects to all the clock inputs of the used flip flops in the design. The lines on the right hand side are the connection lines from the I/O ports, which are the switches and LEDs.

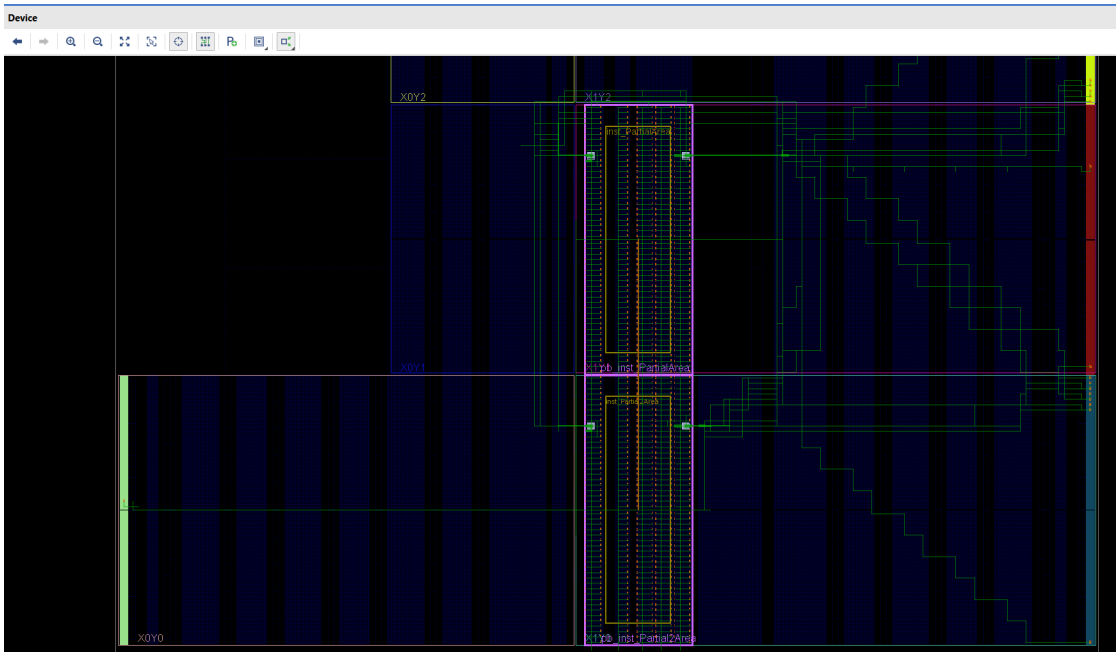


Figure 15: View of static design after removal of the blocker

This is the final step before generating the bitstream, which can be used to configure the FPGA.

5.2.2 Reconfigurable Module

Next up is the module. It is only required to create one module in order to show the actual relocation from Partial Area 1 to Partial Area 2 (see Figure 14 for areas). An important thing to note on the figure is that there are two extra areas, which are located outside the borders of the reconfigurable module. These 2 areas are both connection primitives. If these connection primitives are not present when building the design, Vivado will optimize these signals away during implementation. (7). One connection primitive covers the incoming signals from the onboard switches and the other one covers the outgoing signals, which will travel to the onboard leds. The floorplanning of the module and the two connection primitives can be seen in Figure 16 below.

Again there are bundle wires present in the device overview. One is highlighted to clearly show that there are 2 bundle wires for the module. The module itself uses both the 8 input signals and 4 output signals. The connection primitive input has a bundle wire with the module for the 8 input signals of the adder. The connection primitive output has a bundle wire for the 4 output signals. That is why there are 2 bundle wires for the module. The wires meet at the centre of the pblocks, just to indicate that a pblock has a bundle wire. The actual connection does not happen in the middle. As already stated, one set of wires goes to the connection primitive responsible for the input signals and the other bundle wire goes to the primitive that connects the output signals. In Figure 17 one can see the remaining wires, after the blocker has been removed from the netlist. It can be seen that the wires first go in a straight line and then disperse over the fabric. This is where the blocker ends and where the wires 'come out' of the tunnel to connect with the connection primitives.

5.3 Writing Adaptation Program

The bitstreams have been created, so now it is time to write a program which can modify them. In order to do so, we break the objective in smaller pieces:

1. Find a way to communicate with the SD on which the bitstream is placed.
2. Divide the configuration words into frames and columns
3. Group the words into LUTs
4. Save the changes that are made in the original bitstream file.

For problem one, the library FatFs is used which is discussed in the Toolflow section. The part of the code that is responsible for opening and reading/writing files from/to the SD card, is shown below in Listing 1. This code opens the file on the SD card and writes it to an address (DestinationAddress) in the memory.

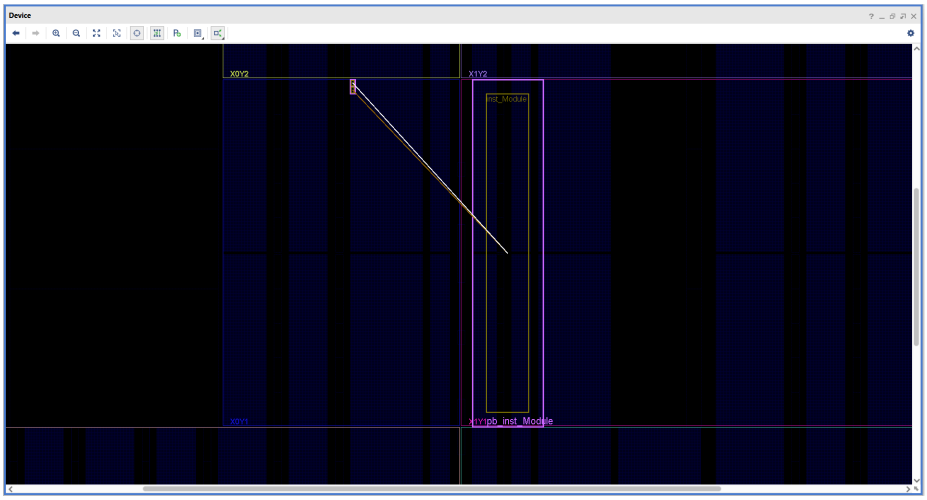


Figure 16: Floorplanning of the reconfigurable module

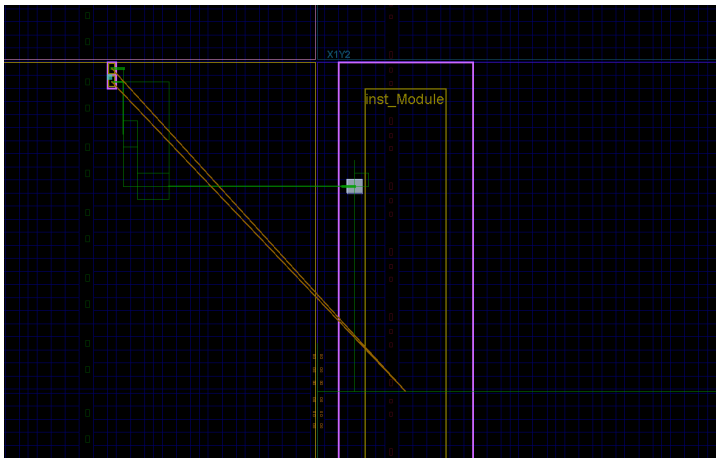


Figure 17: Module wiring with the blocker removed

```
int SD_TransferBitfile(char *FileName, u32 DestinationAddress, u32 ByteLength)
{
  FIL fil;
  FRESULT rc;
  UINT br;

  rc = f_open(&fil, FileName, FA_READ | FA_WRITE);
  if (rc) {
    xil_printf(" ERROR : f_open returned %d\r\n", rc);
    return XST_FAILURE;
  }

  rc = f_lseek(&fil, 0);
  if (rc) {
    xil_printf(" ERROR : f_lseek returned %d\r\n", rc);
    return XST_FAILURE;
  }

  rc = f_read(&fil, (void*) DestinationAddress, ByteLength, &br);
  if (rc) {
    xil_printf(" ERROR : f_read returned %d\r\n", rc);
    return XST_FAILURE;
  }

  rc = f_close(&fil);
  if (rc) {
    xil_printf(" ERROR : f_close returned %d\r\n", rc);
    return XST_FAILURE;
  }

  return XST_SUCCESS;
}
```

Listing 1: Code for transferring files from SD card to memory

For problem two, one needs to take a closer look at the design again. It is known that the address of the first FDRI word is determined by the FAR register. For a typical bitstream, configuration of the FPGA will start at 0x00000000 and after that the address auto increments to the final count. So in order to find the first FDRI it is required to find the FAR location 0x00000000 on the PL. When looking at Figure 8 it can be seen that there are basically 3 different clock rows. Distinguished by the Y coordinate to be either 2, 1 or 0. The matching FAR address for these regions are 0x00000000, 0x00400000 and 0x00420000 respectively. These address values can directly be conducted from the bits that reside in the FAR register (see Figure 3). The 4 in the address sets bit 22 high, meaning that the bottom half rows are selected. The 2 represents the second clock row in the bottom half rows.

From Section 2.2 it is known that word 0x30004000 is followed by a word that indicates the number of configuration words to be written. This in turn is followed by the first FDRI word and this continues till all the configuration words are loaded into the memory. In order to keep track of which coordinates correspond to which words, a set of well set counters is created. First, a *wordcounter* is build in to the program. This counter counts the words starting with the first FDRI word at address 0x00000000. After 101 words a full frame is configured. In order to keep track of the number of frames a *framecounter* is implemented. Remember that word 50 in each frame is supposed to be skipped since this is reserved for the clock lane. It is important to note that not all the resource columns span the same amount of frames.

Algorithm 1: Pseudocode to create a framelist

```
Result: Framelist of fabric
while address ≤ Last FDRI do
    address = address + 1;
    word = word + 1;
    if word == 50 then
        word = word + 1;
        address = address + 1;
    end
    if word == 101 then
        word = 0;
        frame = frame + 1;
    end
    if frame == 35 then
        frame = 0;
        column = column + 1
    end
    if last_column_clkregion then
        clkregion = clkregion - 1;
        column = 0;
    end
end
```

The FAR address starts counting at the bottomleft of clockregion X0Y2 where there are no CLB's present. The area here is reservered for the PS but the FAR already starts counting here. The first configurable frame starts at CLB_X19Y100 in clockregion Y2. For 1 CLB resource column there are 35 frames which means that the column number should be incremented after there have been 101 * 35 FDRI words. This goes on until all the columns of a clock row have passed. Then the counters move 1 clock row down until all clockregions have been processed. A short piece of pseudocode can be seen in Algorithm 1.

Now a framelist is created, meaning that it is possible to pinpoint any CLB on the device using the same coordinates that Vivado uses to indicate which CLB is selected in the GUI. Now it is desired to make a system that can pick 1 of the 8 LUTs that reside in this CLBs and to change the contents of that LUT. In order to do this, one has to look at the frameindex inside that CLB. If it is desired to change the LUT in a SLICE-M, one has to change the bits that reside in frames 31-34. For SLICE-L these frames are 25-28. In order to find this frame numbers, the words in the CLB with frame numbers 31-34 or 25-28 are selected. The framecounter in combination with an if statement can cover this job. Next up is to split a fullword into 2 halfwords. When looking at Figure 10 it can be seen that one fullword is responsible for configuring 2 LUTs. Thus, with a halfword one has finally reached the 4 bits in the entire bitstream that are responsible for 1 LUT in 1 CLB in 1 Clock row in 1 Clock

region on 1 FPGA.

5.4 Write Bitstream Changes

After the bits have been modified, these changes should somehow be saved in order to test the bitstreams on the development board. For this purpose the FatFs library is used again. During this write operation all contents of the registers, including the registers that are changed, are written to a new file on the SD card. Now the AND-gate functionality of the LUT should have been changed to an OR-gate. It is very important that the correct file length is set within the program. This is necessary for successfully uploading the bitstream to the FPGA.

The same principle is used for the relocation of the module. With the program, the FAR address and the value that this register holds can be discovered. The value can be changed and the a new file with the changed FAR is written to the SD card. It is important to note that this bitstreams for partial reconfiguration only hold the configuration information of the module. This means that the size of the partial bitfile will contain far fewer bytes than the size of the full bitstreams of the logic gates. Therefore the size of the newly written partial file should be adjusted accordingly. Otherwise the configuration of the FPGA will not fully succeed.

6 Testing & Evaluation

6.1 LUT Adaptation

To see whether the program finds the correct LUT, a small test is performed. In this test a bitstream with the AND gate configuration, is placed onto the SD card. Then, the SD card is inserted in the board and the program is launched from within the SDK. All the steps in the program are performed and the LUT contents is changed from 0000 5555 5555 0000 to 5555 FFFF FFFF 5555 (see Section 5.1). The changed bitstream is saved to the SD card with a different name to keep the original bitstream intact. This is done in case that more experiments have to be done with the AND bitstream.

The changed bitstream is loaded to the FPGA via the PCAP. As expected, the functionality is now changed from an AND gate to an OR gate, which is clearly indicated by the output of the LED. Pictures of the switch positions and the LED output can be found in the appendix.

6.2 Module Relocation

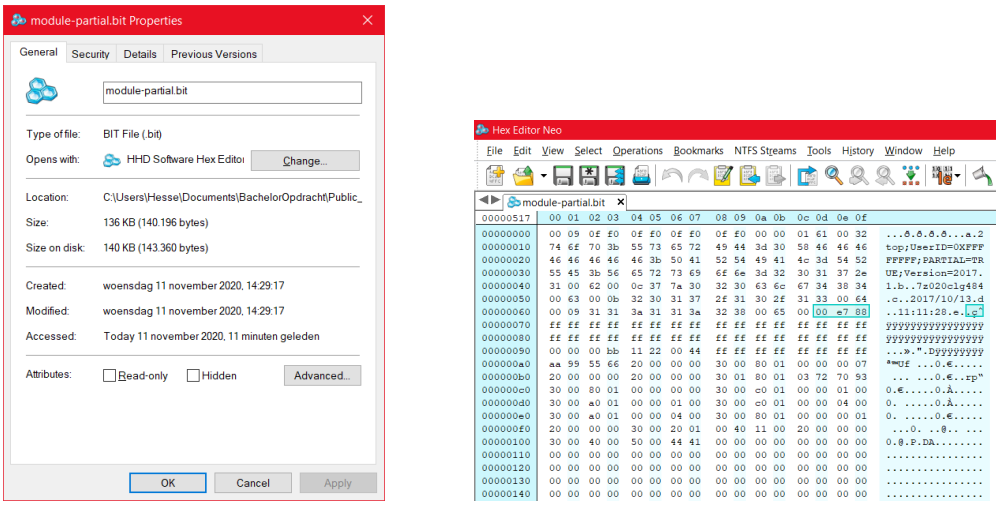
As stated in Section 3.2, it is desired to show the relocation of the module with the usage of LEDs. As a start, The FPGA is configured with the static bitstream. Now all the LEDs on the board are on. This is because the adder functionality of the module is not yet loaded to the board. Since the static design does not contain any hardware blocks, a Xilinx SDK

cannot be launched. This causes problems when generating a FSBL in order to make the program stand-alone. It was found that either the bitstream does not load to the FPGA, or the program was not executed. That's why it is chosen to load the static bitstream to the board via JTAG.

In order to test that the (partial) bitstreams work correctly, they are first loaded to the FPGA using JTAG. First the module is loaded to partial area 1. As a result the leds 0 to 3 respond to the input switches and the design behaves as an adder. In order to see if the relocation works, the value in the FAR of the previous bitstream is changed from 0x00400000 to 0x00420000, meaning that the module would shift from partial area 1 to partial area 2. When the bitstream with the changed FAR is uploaded to the FPGA, indeed the module is now also configured in partial area 2. Both area's have the adder functionality of the module. Next step is to configure the areas with PCAP instead of JTAG.

When working with PCAP, it was found that certain binary files are not compatible with the hardware manager of Vivado. Furthermore it was discovered that BitMan (11) adds a header to the binary files when using the cut option. This causes the PCAP to fail. This makes it impossible to create a partial binary file from a full binary file with BitMan. Furthermore, the header of the partial file contains a mistake. The indicated size in the bitstream does not match the actual size of the file.

After looking at different ways to generate a partial binary file, the following method was used. The partial bitstream is cut out of the full bitstream using BitMan. Now if we open the properties of this partial bitstream it states that the bitstream contains 140196 bytes (see Figure 18a) which would be 02 23 A4 in hexadecimal. However if this bitstream is opened in a hex editor it gives a size of 59272 (e7 88 in hex), which is highlighted in Figure 18b. In order to prevent errors when converting the file from bit to binary, the file size in the bitstream is set to 02 23 A4.



(a) Properties of Partial Bitstream (b) Size of bitstream in hex editor

Figure 18: Differences in size indication by bitstream file properties and the bitstream itself

Now that we have a correct sized partial bitstream, the final step that has to be performed is changing from a bit file to binary file. This is done by a tcl command in Vivado.

Now a binary file is created which can be uploaded using PCAP. This file is placed on the SD card, inserted in the Zedboard and loaded to the memory using the code in Listing 1. The program and static bitstream are initially loaded via JTAG. After the program is running on the board, the JTAG cable can be removed to demonstrate that the partial bitstreams are actually loaded with PCAP and not via the JTAG cable.

When the binary file is not changed using the program, the results are as expected. 4 LEDs behave as an adder, the other 4 LEDs are not effected. However if changes to the partial binary file are made, for instance the FAR address is changed, the result is unexpected. The module is also loaded to area 2, but also to area 1. It is expected that only area 2 has the adder functionality since we merely change the FAR address. The changed bitstream is configured without problems using PCAP. It is assumed that the error occurred by using BitMan but no proof was found for this. Again, the pictures of the switch positions and the LED outputs can be found in the appendix.

7 Conclusion

The goal of this project was to create a program or application that was able to manipulate bits in such a manner to accomplish relocation of reconfigurable modules and to change the contents of LUT tables. A program was build that allows to pinpoint any LUT table on the device and to change its bits. This was achieved by a number of counters in the program that kept track of the addresses, whilst the FDRI words were loaded into the configuration memory. Not only the configuration words can be modified using the program, the FAR register value can also be located and changed in the configuration bitstream. This makes it possible to reconfigure certain areas on the FPGA fabric. By doing so, area utilization can be optimized since multiple modules can be interchanged at the partial area. All this is verified using the onboard switches and LEDs.

7.1 Further Improvement

The program that is build for this project is mostly serving the purpose of changing any LUT on the device. The program in its current state is able to change the contents of the first 16 bits of a LUT at the time. For small changes this would suffice, but if many LUTs have to be adjusted these operations can be quite tedious. One could argue that changing more LUTs at the time would be more convenient. However this also differs on what type of design is chosen. For this thesis only one LUT had to be changed.

In the end of Section 5.1 it is explained how the bits of and AND configuration have to change in order for the functionality to become an OR gate. But this would only be one modification for one specific LUT with restricted pins and a restricted slice. Even moving the configured LUT to another slice, or change the pin mapping of the LUT between the switches and the input pins of the LUT BEL, already gives different bitstreams for the same AND gate functionality. In order to link any bit change to a meaningfull hardware change, one has to take a look at reverse bitstream engineering. As the name of this technique already states, one 'reverses' the bitstream, meaning that from the bits in the bitstream the netlist can be extracted (15). If adversaries were to know which bits to change in a bitstream in order to fulfill their own dark agenda, devices or even humans can be damaged.

Reverse bitstream engineering can also be used for more approachable design changes. For this research, 2 bitstreams were generated and compared to see how the bits have to be changed to get from an AND to an OR gate. If the designer wants yet another functionality, a new bitstream has to be generated and one has to look for the differences again. To investigate what changes are meaningfull one has to experiment with a lot of different implementations and look at the differences in the bits that have occured after a single hardware change. With reverse bitstream engineering, this step can be skipped since one would understand how a bit change would affect the hardware. However, reverse bitstream engineering lies beyond the scope of this project.

References

[1] Xilinx, "Partial reconfiguration in vivado." https://www.xilinx.com/video/hardware/partial-reconfiguration-in-vivado.html

[2] M. Ender, A. Moradi, and C. Paar, "The unpatchable silicon: A full break of the bitstream encryption of xilinx 7-series fpgas." https://www.usenix.org/system/files/sec20fall_ender_prepub.pdf

[3] M. C. Yang, "Rapid prototyping(1) - integration of arm and fgpa." http://www.cse.cuhk.edu.hk/~mcyang/ceng3430/2020S/Lec09%20Rapid%20Prototyping%20(I)%20-%20Integration%20of%20ARM%20and%20FPGA.pdf

[4] Xilinx, "User guide 470: 7 series fpgas configuration." https://www.xilinx.com/support/documentation/user_guides/ug470_7Series_Config.pdf

[5] C. Beckhoff, D. Koch, and J. Torresen, "Goahead: A partial reconfiguration framework." https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6239789

[6] D. Koch, "Partial reconfiguration on fpgas."

[7] T. Hogekamp, "Framework for fine-grainedpartial reconfiguration on fpgas." http://essay.utwente.nl/80035/1/Hogenkamp_MA_EEMCS.pdf

[8] Xilinx, "User guide 585: Technical reference manual." https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf

[9] ckohn (Xilinx Employee), "How to use pcap to config the pl in zynq." https://forums.xilinx.com/t5/Processor-System-Design-and-AXI/How-to-use-PCAP-to-config-the-PL-in-zynq/td-p/280230

[10] "Fatfs - generic fat filesystem module." http://elm-chan.org/fsw/ff/00index_e.html

[11] K. D. Pham, E. Horta, and D. Koch, "Bitman: A tool and api for fpga bitstream manipulations." https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7927114

[12] "Zedboard ™." http://zedboard.org/product/zedboard

[13] I. Tuzov, "Optimizing and prioritizing LUT-specific essential bits for bit-accurate FPGA-based fault injection," 10/10/2019.

[14] "zedboard master.xdc." https://github.com/Digilent/Zedboard/blob/master/Resources/XDC/zedboard_master.xdc

[15] F. Benz, A. Seffrin, and S. A. Huss, "Bil: A tool-chain for bitstream reverse-engineering." https://ieeexplore.ieee.org/abstract/document/6339165

Appendix

Pictures of Switches Positions and LED Outputs

```
******Enter coordinates of single LUT for extraction******
*
*Regions with Y2 = 2, Y1 = 1 and Y0 = 0
* ClkRegion: = 2
*
* Unvalid Column inputs are:
*<=1, 6, 9, 14, 17, 22, 25, 33, 36, 50, 56, 59, 64, 67, >=72
*
* Column: = 71
Row ranges from 0-49
* Row: = 1
* Slice (M/L): = l
#configwords = 1010808


Lut A = 0000 0000 0000 0000
Lut B = 0000 5555 5555 0000
Lut C = 0000 0000 0000 0000
Lut D = 0000 0000 0000 0000

Address of Lut A&B words are:
 (1) 2F5298(63-48)
 (2) 2F5104(47-32)
 (3) 2F4F70(31-16)
 (4) 2F4DDC(15-0)

Address of Lut C&D words are:
 (5) 2F529C(63-48)
 (6) 2F5108(47-32)
 (7) 2F4F74(31-16)
 (8) 2F4DE0(15-0)

Want to change a LUT? (y/n)
```

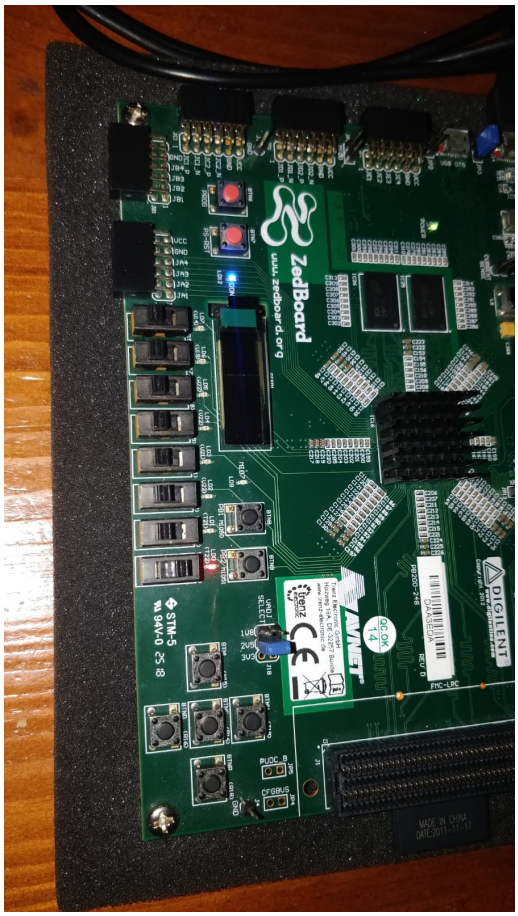
(a) Slice-L LUT configuration AND gate

(b) Switch settings and LED output on the Zedboard

Figure 19: Program and LED output of the AND configuration before bitstream manipulation

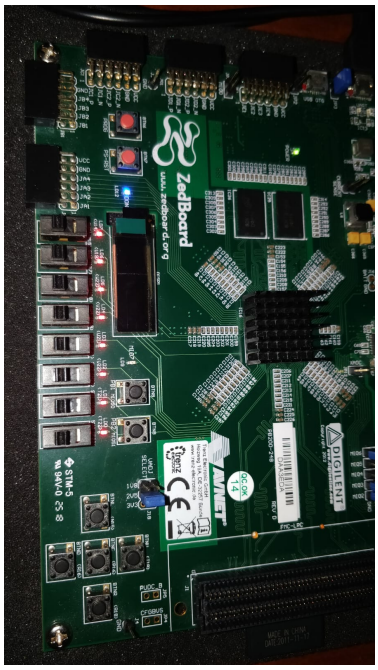
```
******Enter coordinates of single LUT for extraction******
*
*Regions with Y2 = 2, Y1 = 1 and Y0 = 0
* ClkRegion: = 2
*
* Unvalid Column inputs are:
*<=1, 6, 9, 14, 17, 22, 25, 33, 36, 50, 56, 59, 64, 67, >=72
*
* Column: = 71
Row ranges from 0-49
* Row: = 1
* Slice (M/L): = l
#configwords = 1010808


Lut A = 0000 0000 0000 0000
Lut B = 5555 FFFF FFFF 5555
Lut C = 0000 0000 0000 0000
Lut D = 0000 0000 0000 0000

Address of Lut A&B words are:
 (1) 2F5298(63-48)
 (2) 2F5104(47-32)
 (3) 2F4F70(31-16)
 (4) 2F4DDC(15-0)

Address of Lut C&D words are:
 (5) 2F529C(63-48)
 (6) 2F5108(47-32)
 (7) 2F4F74(31-16)
 (8) 2F4DE0(15-0)

Want to change a LUT? (y/n)
```

(a) Slice-L LUT configuration OR gate

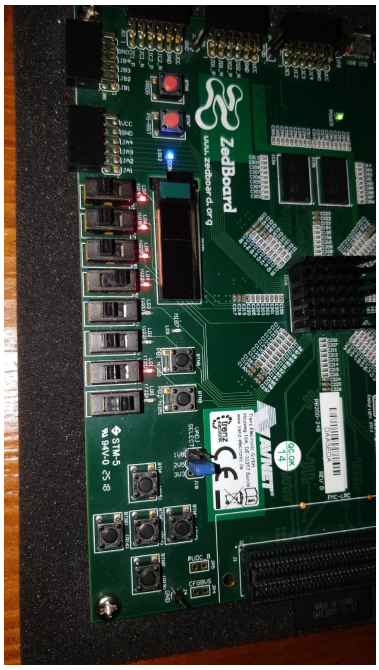


(b) Switch settings and LED output on the Zedboard

Figure 20: Program and LED output of the AND configuration after bitstream manipulation



(a) Static design LED output on Zedboard



(b) Module loaded to partial area 1

(c) Module loaded after FAR change

Figure 21: LED outputs after loading the static design followed by loading the module to area one and the relocation to area 2

List of Acronyms

AXI (Advanced eXtensible Interface)

BEL (Basic Element of Logic)

BRAM (Block Random Access Memory)

BSP (Board Support Package)

CLB (Command Logic Block)

DSP (Digital Signal Processor)

FAR (Frame Address Register)

FDRI (Frame Data Register Input)

FSBL (First Stage Boot Loader)

HDL (Hardware Description Language)

IDE (Intergrated Design Environment)

IP (Intellectual Property)

JTAG (Joint Test Action Group)

LUT (Look Up Table)

PCAP (PS-processor Configuration Access Port)

PL (Programmable Logic)

PS (Processing System)

SD (Secure Digital)

TCL (Tool Command Language)

VHDL (VHSIC Hardware Description Language)

VHD (Virtual Hard Disk)