Bitfiltrator Extended

CSEE, Columbia University
535 West 116th Street New York, New York 10027, USA

Isabelle Friedfeld-Gebaide
if2266@barnard.edu

Noam Hirschorn
nyh2111@columbia.edu

Dan Ivanovich
dmi2115@columbia.edu

***Abstract*— Xilinx bitstreams are encoded in Vivado to instruct FPGAs how to run code. However, the translation from code to bitstream is not distributed by Xilinix. The Bitfiltrator paper hypothesized how to reverse engineer this process on the Xilinx Ultrascale and Ultrascale Plus architecture. This paper attempted to recreate what that team did, and to use the same methods to analyze the Xilinx 7-Series FPGAs, the architecture preceding Ultrascale.**

Analysis of the original Bitfiltator paper and code was able to be completed in addition to most of the necessary changes to convert the methods used for Xilinx Ultrascale and Ultrascale Plus reverse engineering of the bitstream. The paper was able to complete an analysis of the 7-Series FPGA's bitstream and a partial recreation of the reverse engineering process found in the original Bitfiltrator paper, with full implementation for a select number of 7-Series FPGAs .

***Keywords*— Bitfiltrator, FPGA, 7 Series, Reverse Engineer, Xilinx**

I. Introduction

An FPGA (Field Programmable Gate Array) is a programmable device which can perform hardware simulation. During compilation of a piece of code for hardware simulation of an FPGA, instructions to an FPGA for how to set each of its component parts is generated, this set of instructions is known as the bitstream. Xinlinx's Vivado is currently the only way to generate a bitstream for an Ultrascale/Ultrascale+ FPGA [1]. Unfortunately, Xilinx does not publically release the formula by which Vivado translates user code to FPGA bitstreams.

The Bitfiltrator paper [1] analyzed various fields in order to reverse engineer the addressing format and formula of the Ultrascale/Ultrascale+ FPGA bitstream to match up with real components of the Ultrascale/Ultrascale+ FPGAs.The results indicate a potential of on-the-fly reprogramming of FPGAs without going through the overhead of using Vivado, the tool which normally generates these bitstreams.

One benefit, on-the-fly reprogramming of FPGAs is faster. Due to the Bitfiltrator method one can bypass fault-testing and other time consuming confirmation checks from Vivado.

Another benefit, detecting if the bitstream has been modified is easier using the Bitfiltrator method. The bitstream can be read from the FPGA as it runs, and it could then be reverse engineered to ensure that the program's functionality is still as intended, and help guarantee that no changes were made just before the bitstream was generated.

A final benefit, reverse engineering of the bitstream can allow the hiding of sensitive information. If a custom bitstream can be created, sensitive information could be placed in LUTs harder to access (whether to protect from side channels or other attacks), which could be chosen via a specially crafted bitstream.

This paper attempted to to extend the Bitfiltrator methods from Ultrascale/Ultrascale+ FPGAs architecture to the 7-Series FPGAs architecture. The architecture of the FPGAs differs in subtle but important ways from Ultrascale paper, which is what Bitfiltrator[1]worked on.

II. Xilinx Architecture Overview

The Xilinx architecture is defined in terms of a hierarchy with a series of levels. These levels each define various related sections of the very regularly organized grid structure of FPGAs. The highest level is the device, which is the overall FPGA device, the next level after is the SLR (Super Logic Region), followed by the FSR (Fabric Sub Region/Clock Region) as seen in Figure 1.

These chips are often small enough that many devices only have SLR, particularly with the 7-Series FPGAs (with a select few Virtex devices having multiple)[2]. Some of the Virtex-7 chips are sufficiently large enough that Stacked Silicon Interconnect (SSI) technology is necessary in order to prevent communication between SLRs from being too slow [2]. SSI effectively increases the overall capacity of the device by combining multiple SLRs on a passive interposer layer [2].

One region relevant to the Bitfiltrator method is the Clock Region. This is a region of the SLR which uses a direct connection from the central clock of the device to serve as a clock source for all the elements in the area. It is effectively a neighborhood of an FPGA, with a defined size across architectures. One of the primary focuses of the Bitfiltrator paper and method [1] is to translate data from the clock region to its addressing location as encoded in the bitstream.

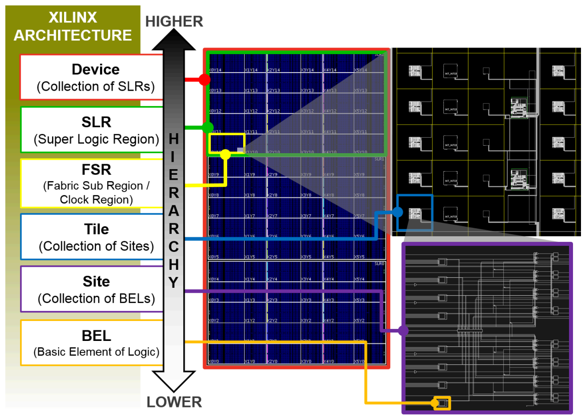

Fig. 1. Levels of architectural hierarchy in Xilinx FPGAs. Starting with Device as the highest level in the architectural hierarchy in Xilinx FPGAs, followed by SLR, FSR (Clock Region), Tile, Site and ending with BEL. [3]

Each clock region contains rows with two types of resources – that are important to the Bitfiltrator method – CLBs (configurable logic blocks) and BRAMs (Block RAM/ memory)[1]. CLBs allow for the FPGA to actually perform logic and calculations while BRAM serves as memory. These rows also have major columns and minor columns, which is the lowest level defined in the bitstream [1]. This can be compared to a building number in a mailing address, while SLRs can be compared to the state.

This lowest level of granularity defines the addresses of slices, which actually make up individual CLBs and BRAM. These resources make up the rows and columns inside of a clock region. Additionally, there is a specialized type of CLB which is a DSP (Digital Signal Processor). Each slice will help make up one of these resources.

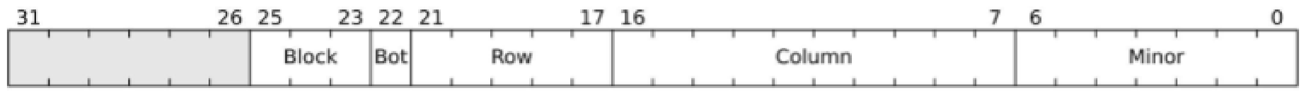

Fig. 2. Xilinx 7-Series FAR (Frame Address Register) [4]

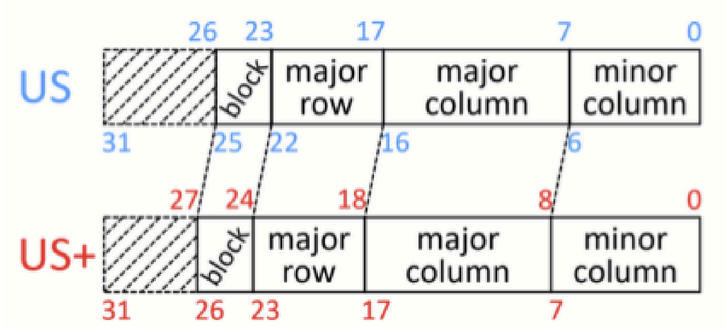

Fig. 3. The frame address format in UltraScale and Ultra- Scale+ devices. The address is composed of a 4-tuple: block type, major row, major column, and minor column. The block type is either CLB_IO_CLK or BRAM_CONTENT. The number above/below each field marks its start/end offset in the 32-bit FAR word. UltraScale+ devices have an expanded minor column field compared to UltraScale devices. [1]

The bitstream assigns addresses values to each slice. The will call a slice's address with the FAR (Frame Address Register) and then use the FDRI (Frame Data Register Input) to pass on configuration details. The FAR is defined by 4 primary elements in the Ultrascale architecture and 5 in the 7 series. The SLR element defines the SLR of the slice, while the major row, major column, and minor column provide more detailed information. The extra element in the 7 Series is the top/bottom bit, also known as the half bit. This bit determines which half of the device the slice is in, in a manner which will be explained below [1].

The final relevant register is IDCODE, which contains the name of the SLR (which is important if there are multiple), since information is often with respect to the SLR rather than the device. For example, the major row in Ultrascale is defined as what row the slice is in in the SLR[1].

A. Key Differences between 7-Series FPGAs and Ultrascale

The primary difference between the 7-Series FPGAs and Ultrascale/Ultrascale+ architectures is the introduction of a top/bottom bit (Bot) in the FAR, as can be seen in Figure 2.

In Ultrascale, the rows of a clock region are simply incremented from the bottom of SLR, to create a relative row address, as can be seen in Figure 3. In the 7-Series FPGAs, the relative row address is instead calculated as a distance from the center of the device, using the top/bottom bit. The 7-Series top/bottom configuration contains an additional layer of hierarchy, and it completely changes the calculation of the relative row address when compared to its successor, the Ultrascale/Ultrascale+ FPGAs.

Another difference is regarding the basic 18kB BRAM block. In the Ultrascale, the block is the 18E2 while the 7-Series FPGAs uses 18E1, which are instantiated differently in Verilog. The 18E2 is a more updated version which has inputs to cascade between BRAM blocks, but that element was not actually used in the paper. Nevertheless, it was important to switch the block and code in ring_bram.v to use the type of BRAM that is actually on the device.
A similar effect applies to the DSP, where the paper uses DSP48E2 when it needs to use DSP48E1 for the 7 series.

III. Methodology

For the most part, reversing the bitstream for a 7-Series device can be done using the same method as the original paper. The original paper's codebase revolved around the CLI-driven `create_device_summary.py`, which uses a combination of Python, Verilog, and TCL scripts to generate and parse the necessary bitstreams. create_device_summary.py exactly follows the reversing methodology described in the paper, so the team set the measurable goal of adjusting the codebase such that the steps in `create_device_summary.py` will successfully run on a 7-Series device. This process involved adjusting almost every step of the original process to account for major architectural differences — particularly the lack of SLRs, the addition of half bits, and the change in logic location formats. Ultimately, this goal was reached.

The largest changes made to adapt to the 7-Series devices was the introduction of half bits and the removal of SLRs. The data structures in the code by Kashani et al. was highly dependent on there being multiple SLRs, and major changes had to be made to virtually all functions used in the create_device_summary process to parse and store bitstream information in the authors' JSON-like defaultdict trees. Further changes had to be made to account for differences in metadata encoding in the bitstream, such as the FPGA part numbers. 7-Series devices store their part numbers differently, which made the code by Kashani et al. completely unable to determine the architecture (Ultrascale/Ultrascale+/7-Series) of a generated 7-Series bitstream. For a basic summary of how changes regarding the top/bottom bit were implemented, see src/major_architecture_level_changes.txt.

In general, the team tried to implement all changes in a way that would still allow the codebase to function on Ultrascale/Ultrascale+ devices as it originally had. Sometimes, this was easy to do, but major data structure adjustments for the inclusion of half bits often made this much, much harder. For the sake of completing the adaptation in time, many of these more-difficult adjustments, which would've required writing completely separate classes and functions for Ultrascale+ vs 7-Series, were left unimplemented. Thus, the codebase now only works for the 7-Series devices — although it will still be able to parse Ultrascale+ devices correctly, the major changes made to core data structures means the modified codebase is unable to store its results in memory. Some further simple but incredibly-tedious development work would be required to make this codebase a complete extension capable of handling Ultrascale, Ultrascale+, and 7-Series devices alike.

In development work, Artix-7 devices were used. As the "lower power and cost" option of the 7-Series families, the Artix-7 boards had fewer components, which made them ideal for development work. The submitted codebase has been tested and confirmed to successfully step through the `create_device_summary.py` process on multiple Artix-7 FPGAs.

A. Issues faced

One difficulty of the paper was in access. There were issues in running the code on some computers, which made it much more difficult to debug issues as they primarily had to be solved through static analysis, and fixes could not be quickly and easily tested. The team therefore had times when some team members were unable to contribute even when they had time, to lack of being able to determine what the next immediate steps should be.

An additional issue was lack of clarity on the goal. Initially, there were some issues in figuring out what the project was supposed to be, whether it be to focus on verifying the findings of the original paper or attempting to recreate its methods without using the provided code. This led to a delay in actually being able to start the project proper.

The code produced by this paper does not seem to fully work on the other 7-Series families (Kintex and Vertex), as all of our development time was spent working with Artix-7. There are other some small but critical issues in how some attributes of the Kintex and Vertex devices are encoded, such as DSP and LL naming, but our code is able to get devices from these families most of the way through the reversing process.

IV. Breaking Bitfiltrator

We also have spent a bit of time analyzing the Bitfiltrator [1]code to see if there would be any ways to prevent this reverse engineering. The primary assumptions of this code seem to be on the consistency of the layout, which would be a great cost to change. Even making the size of clock regions not quite as standardized would disrupt addressing used in Bitfiltrator, but would again involve disrupting the regularity of the device.

However, an easier method Xilinx could use would be to simply change the order/way that the code translates between a slice or clock region (as given in the absolute format of `X_Y_`) and its corresponding address in the FAR (as determined by its major row/column and minor column. Right now, Bitfiltrator [1] was able to use the information that the FAR will simply increment from the bottom row of each SLR in the ultrascale architecture.

This relationship was also explained with regards to 7-Series FPGAs, albeit following a different pattern than Ultrascale. If Xilinx were to have this relationship follow some sort of encoded pattern however (so it could not be determined from the user guide what major row `Y_` would map to), this would disrupt the premise on which Bitfiltrator[1] was founded, with the added benefit that it would not actually require a change in hardware. Instead, all that would be necessary would be to edit Vivado so it would know how to perform this mapping. This could be viewed as a safe proposition since the point of Bitfiltrator[1] is that it is difficult to actually track the operations Vivado performs to transform code into a bitstream, so this encoding would be hard to break just by attempting to find it in Vivado.

Another simple option would be to simply restrict information about the size of different parts of the hierarchy. In particular, it is important to know the size of a clock region to then be able to measure where it is. The Bitfiltrator [1] paper gives an example of finding the SLR from knowing the number of CLBs per clock region and the number of clock regions per SLR [1]. While the number of clock regions per SLR was something the method had to figure out, it was explicitly told in a user guide about how tall each clock region was. This information was important for being able to make further inferences of the rest of the paper . Of course, there is a cost to hiding this information, in that it can make it generally harder to use/study the FPGA if the size of the clock regions is relevant. Nevertheless, this could also be a viable option as it avoids the extremely distasteful option of disrupting the regularity of the hardware.

V. Conclusion

While the extension of the Bitfitrator method to 7-Series FPGA may have run out of time, there were a few key takeaways. First is the way different versions of an architecture can change things which do not even appear relevant at first blush. While differences such as the introduction of a top/bottom bit were expected, other issues happened simply because different versions of registers were used for the BRAM and DSP.

Another takeaway was just how similar each type of resource is in an FPGA. Everything was strictly hierarchical, and even the different resources were still treated with very similar code since they all fit into the same addressing structure with major and minor columns, major rows, half bits, and an SLR.

A final lesson is about the importance of communication over the course of a team project. It isn’t enough to just set some time in one’s schedule each week to work on a project if there is no coordination about who is doing what, what the next task is, and when is the crunch time.

VI. Future work

We currently hypothesize that our changes should be enough for the code to work on the 7-Series FPGAs overall. However, we were unable to finish the debugging process in time. We examined each time the code referenced a major row and edited it to reflect the way it should be relatively addressed as outlined in the 7-Series FPGAs User guide. To test it, all that would be needed would be to go to the project u rays comparator files and replace the u rays database file with one from project X rays, which is the 7-Series counterpart to U rays.

There is the possibility of an issue with our code when dealing with a device with more than 1 SLR. In Ultrascale, the major rows are relative to the SLR, while in the 7-Series FPGAs, it is in reference to the distance from the center row of the device [5]. This has the potential to cause an issue if our code only finds the distance from the center of the SLR, which some parts may use instead of the center of the device due to the way the code is set up. We considered trying to fix this error, but decided that we would begin by trying to get the code to work with 1 SLR, which includes most 7-Series FPGAs Products.

We were concerned that more errors could happen in the more complicated task of getting the center of the device given that the entire code is built around SLRs being effectively the highest level of hierarchy (with all the information relevant to the SLR being entirely in the SLR array)

VII. Bibliography

[1] Sahand Kashani, Mahyar Emami, James R. Larus (2022): Bitfiltrator, a general approach for reverse engineering Xilinx

bitstream formats. Retrieved from https://github.com/epfl-vlsc/Bitfiltrator

[2] 2016. 7 Series FPGAs Configurable Logic Block. Retrieved from https://fpga.eetrend.com/files-eetrend-xilinx/download/201408/7595-13762-ug4747seriesclb.pdf

[3] 2023. Xilinx Architecture Terminology. *RapidWrightDocs*. Retrieved May 5, 2023 from https://www.rapidwright.io/docs/Xilinx_Architecture.html

[4] 2023. Frame Address Register (FAR). *RapidWrightDocs*. Retrieved May 5, 2023 from https://www.kc8apf.net/2018/08/unpacking-xilinx-7-Series-bitstreams-part-3/

[5] Rick Altherr. 2018. Unpacking Xilinx 7-Series Bitstreams: Part 3 | kc8apf.net. *KC8APF*. Retrieved May 5, 2023 from https://www.kc8apf.net/2018/08/unpacking-xilinx-7-Series-bitstreams-part-3/