Bitfiltrator: A general approach for reverse-engineering Xilinx bitstream formats

Sahand Kashani, Mahyar Emami, James R. Larus
School of Computer and Communication Sciences
EPFL
Lausanne, Switzerland
{sahand.kashani, mahyar.emami, james.larus}@epfl.ch

***Abstract*—As the usage of FPGAs spreads, engineers will inevitably employ them in ways unforeseen—or unwanted—by their manufacturers. Xilinx's toolchains offer multiple points for customizing the FPGA compilation flow, but all flows must end with Vivado as it is the only tool capable of generating the bitstream to program an FPGA. Xilinx does not document its bitstream format, so users who wish to bypass Vivado and modify a bitstream directly must reverse-engineer it to discover the location and format of cells.**

Prior work has reverse-engineered parts of the bitstream format for security or debugging/instrumentation activities, but no paper has explained *how* to do this reverse engineering systematically! Code from prior efforts (when available) is hard-coded to reverse engineer a specific device and is difficult or impossible to use for another one. These efforts—focused on applications instead of reverse-engineering—compel engineers who need to modify a bitstream to rediscover unwritten practice.

Our work bridges this gap by explaining: (1) the various parameters needed to navigate a bitstream correctly, (2) the experiments to obtain them, and (3) the many pitfalls and erroneous assumptions to avoid while undertaking this endeavor. We demonstrate our technique by using it to extract the bitstream format of initial LUT equations, LUTRAM contents, BRAM contents, and register values in Xilinx UltraScale and UltraScale+ FPGAs. Our methods are implemented in an open-source tool, *Bitfiltrator* [1], that can extract device layouts and architecture-specific bitstream formats for these cells automatically and without physical access to an FPGA.

***Index Terms*—Xilinx, Bitstream, Reverse Engineering, UltraScale, UltraScale+**

I. Introduction

Xilinx's Vivado toolchain is currently the only way to generate a bitstream for the company's high-performance UltraScale and UltraScale+ FPGAs. Unfortunately, bitstream generation is time-consuming as Vivado must load a design checkpoint and run design rule checks (DRC) to verify a design's legality before producing its bitstream. Designs that fail DRCs will not be generated to avoid damaging a device. Though Vivado allows selective disabling of DRCs before bitstream generation, the process can still take tens of minutes to load a design checkpoint for a large project before generating the bitstream. In short, bitstream generation will be slow if using Vivado is necessary. This raises the question of whether Vivado can be bypassed entirely?

Modifying an existing bitstream is one way to this end, and numerous projects have taken this path. Prior work has reverse-engineered parts of various bitstream formats. For example, StateMover [2] demonstrated that a design running on an FPGA could be stopped, its bitstream read from the device, its user-visible state *extracted from the bitstream*, and this state then passed to a software simulator. This simulator continues the execution and exposes complete visibility into a design's state. After the software simulation, StateMover retrieves the design's state from the simulator and *embeds it back into the bitstream*. The FPGA is programmed with the new bitstream and continues executing the design at native speed. In addition, StateMover [2] manipulated a partial bitstream to disable LUTRAM masking during readback. Similarly, XBERT [3] employed knowledge of BRAM bits' locations in a bitstream to support APIs for zero-cost access to BRAMs on Xilinx FPGAs using partial reconfiguration. BitMan [4] reverse-engineered the bitstream format of parts of the interconnect and clocking network to re-reroute signal connections without using Vivado. Other work conducts fault injection experiments using FPGAs to study designs used in harsh environments [5]. Faults are emulated by directly modifying an FPGA's bitstream rather than re-generating it, to save time when studying numerous faults.

Applications clearly could benefit from the ability to modify an FPGA's bitstream directly, but how does one do this?

An FPGA's high-level configuration sequence may be documented [6], but the binary format of the configuration data is not. It is unlikely a manufacturer will make this format public since a bitstream's binary format is tightly coupled to the proprietary physical layout of the underlying device [7]. Users who wish to manipulate a bitstream directly must then reverse-engineer its format.

The papers on prior applications of bitstream modification did not, however, explain *how* they reverse-engineered the bitstreams. This is understandable as space constraints limit a paper's ability to explain details. Some projects have made their reverse-engineering source code available, which offers an executable description of the techniques missing from the papers. Unfortunately, this code contains device-specific constants of unclear origin. Attempting to reuse these codebases for other devices is further complicated by embedded assumptions that do not hold for other devices. To the best of our knowledge, BitMan [4] is the most advanced tool for bitstream manipulation as it can even convert FPGA bitstreams into a netlist [8]. However, BitMan is only distributed in binary form, and the publically-available version does not support new, large multi-SLR devices. In summary, much of the knowledge gained from past reverse-engineering projects is unavailable for others to learn from and apply to other devices.

Our work bridges this gap by focussing on the methods necessary to derive the fundamental parameters needed to implement a bitstream manipulation tool. We explain why these parameters are needed, where they come from, and how to infer them. We also describe various pitfalls and erroneous device modeling assumptions. We implement an automated tool, Bitfiltrator, that uses a systematic algorithm to locate the precise position of a resource's initial configuration bits in a bitstream. We used Bitfiltrator on 34 UltraScale and UltraScale+ FPGAs to locate the configuration bits of initial LUT equations, LUTRAM and BRAM contents, and register values. Bitfiltrator does not require access to a physical FPGA to extract its device-specific parameters and its source code is available [1]. Bitfiltrator is written entirely in Python and was developed and tested on x86-based processors. It can also run on ARM-based processors typically found in heterogeneous SoC-FPGA devices, and can therefore be used in autonomous or self-healing systems that rely on bitstream manipulation to correct errors.

The rest of this paper is organized as follows: We start with some background on Xilinx FPGAs and their structure (§ II). We then continue with a bitstream's structure (§ III) and describe how device-specific parameters are used to locate a configuration frame in the bitstream and how to infer them (§ IV). Upon finding a configuration frame, we then explain how to extract the architecture-specific configuration bits of a given resource's initial value within the identified frame (§ V). Finally, we experimentally verify our parameters and discuss future directions (§ VI) before concluding (§ VII).

II. Device structure

Figure 1 shows an annotated floorplan of a Xilinx Alveo U50 datacenter FPGA, which we will refer to in the following sections. Xilinx FPGAs consist of multiple Super Logic Regions (SLR), each containing a grid of clock regions (e.g., `X4Y4` and `X5Y2` in Figure 1). A clock region has a columnar structure where columns contain a homogeneous resource (CLBs, DSPs, BRAMs, etc.) and associated clocking. The height of a clock region in UltraScale and UltraScale+ devices is 60 CLBs, 24 DSPs, and 24 18Kb BRAMs [9].

Vivado identifies cells in a design by a basic element (BEL) name. BELs are named using a *resource-specific* prefix and XY coordinate system (e.g., `SLICE_X136Y247/A6LUT` in Figure 1). BELs have *properties* that Vivado uses to set specific configuration bits when generating a bitstream. For example, LUTs have a 64-bit `INIT` property that configures their equation.

Our goal is to map the `INIT`-related properties of a specific BEL (LUT, LUTRAM, BRAM, and register) to bit offsets within the bitstream.

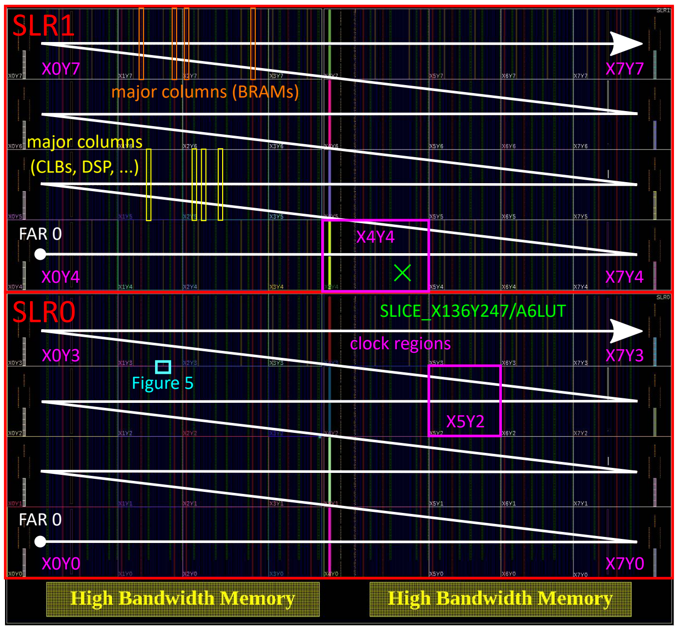

Fig. 1: Floorplan of an Alveo U50 datacenter FPGA composed of two SLRs. We highlight two clock regions in magenta, 4 major columns containing BRAMs in orange, and 4 major columns containing any other homogeneous resource (CLBs, DSPs, etc.) in yellow. Frame addresses start from 0 at the bottom-left clock region of each SLR and increase in the direction of the white arrow. A zoomed in image of the cyan region is shown in Figure 5 and will be explained in Pitfall 4.

III. Bitstream structure

Xilinx bitstreams are composed of two parts: a text header and a binary section. The header is a key-value store of metadata about the bitstream [10], and the binary section contains the bitstream data. The binary section is parsed by first locating a unique byte pattern called the `SYNC_WORD` [6]. Data after the `SYNC_WORD` consists of a sequence of packets that write to the internal registers of a configuration processor. Packets can be parsed until a write to the `CMD` register occurs with a payload of `DESYNC` [6]. The process repeats until all packets are consumed. The configuration processor has 32 registers, but the configuration-related packets only write to three registers: `FAR`, `FDRI`, and `IDCODE`.

1) `FAR` register: The smallest configurable unit in Xilinx FPGAs is called a *frame*. Every frame has a unique address composed of a 4-tuple: a block type[1] (`CLB_IO_CLK` or `BRAM_CONTENT`), a major row, a major column, and a minor column (cf. Figure 2). In the bitstream, the Frame Address Register (`FAR`) stores the current frame's address. Frames with a block type of `BRAM_CONTENT` contain the initial contents of BRAMs in the device. All other device resources (CLBs, DSPs, IOs, interconnect, etc.) are configured by frames with a block type of `CLB_IO_CLK`. All frames in an architecture

[1]The block type field is 3 bits wide, but only two values are legal: (1) `0b000` for `CLB_IO_CLK`, and (2) `0b001` for `BRAM_CONTENT`.

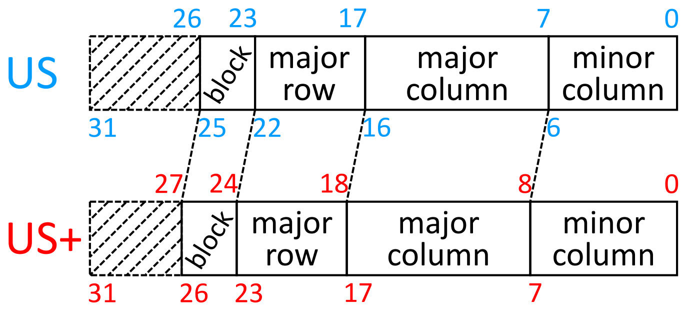

Fig. 2: The frame address format in UltraScale and UltraScale+ devices [6]. The address is composed of a 4-tuple: block type, major row, major column, and minor column. The block type is either `CLB_IO_CLK` or `BRAM_CONTENT`. The number above/below each field marks its start/end offset in the 32-bit `FAR` word. UltraScale+ devices have an expanded minor column field compared to UltraScale devices.

have the same size[2] and span one element (CLB, BRAM, DSP, etc.) horizontally with a height of one clock region [11].

2) `FDRI` register: A frame at address `FAR` is configured by writing a frame-sized payload to the Frame Data Input Register (`FDRI`). Multiple consecutive frames can be configured with a single, large write to `FDRI`. The configuration processor *auto-increments* the contents of `FAR` at every frame-sized interval. Frame addresses in every SLR start from 0 and increase following the order defined by the white arrow in Figure 1: minor column, major column, major row, and block type [4].

3) `IDCODE` register: Writes to the `IDCODE` register identify the target device to be configured. Writes to the `FDRI` register must be preceded by a write to `IDCODE`. The bitstream of a small, single-SLR device writes to the `IDCODE` register once, whereas the bitstream of a large multi-SLR device writes a different value to `IDCODE` to access a specific SLR.

```
# SLR 1
REG = IDCODE ,COUNT =       1, PAYLOAD = { 0x04b37093 }
REG = FAR    ,COUNT =       1, PAYLOAD = { 0x00000000 }
REG = CMD    ,COUNT =       1, PAYLOAD = { WCFG }
REG = FDRI   ,COUNT =       0, PAYLOAD = {  }
REG = FDRI   ,COUNT = 6679260, PAYLOAD = { ... }
# SLR 0
REG = IDCODE ,COUNT =       1, PAYLOAD = { 0x04b22093 }
REG = FAR    ,COUNT =       1, PAYLOAD = { 0x00000000 }
REG = CMD    ,COUNT =       1, PAYLOAD = { WCFG }
REG = FDRI   ,COUNT =       0, PAYLOAD = {  }
REG = FDRI   ,COUNT = 6679260, PAYLOAD = { ... }
# SLR 2
REG = IDCODE ,COUNT =       1, PAYLOAD = { 0x04b24093 }
REG = FAR    ,COUNT =       1, PAYLOAD = { 0x00000000 }
REG = CMD    ,COUNT =       1, PAYLOAD = { WCFG }
REG = FDRI   ,COUNT =       0, PAYLOAD = {  }
REG = FDRI   ,COUNT = 6679260, PAYLOAD = { ... }
```

Listing 1: Alveo U200 bitstream dump. We omit non-configuration-related packets for brevity. The U200 is a large FPGA with three SLRs. Note that the SLRs are not configured in order.

A typical full bitstream contains a *single* write to `IDCODE` and `FAR` followed by a device-sized write to `FDRI` with all

[2]Frames are composed of 123 32-bit words in UltraScale devices and 93 32-bit words in UltraScale+ devices [6].

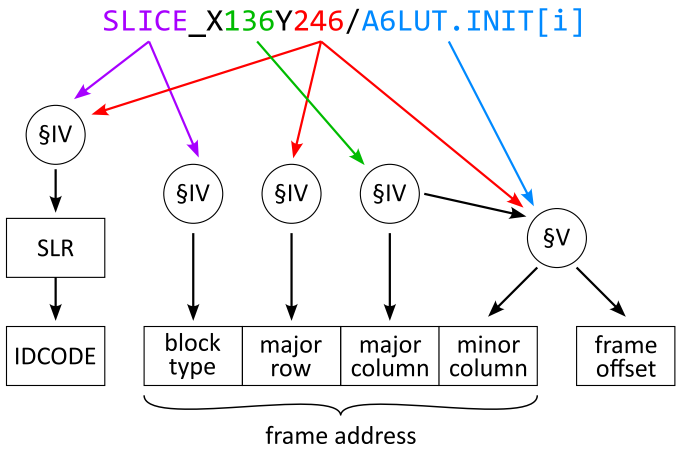

Fig. 3: The process for mapping a bit in a LUT's `INIT` property to a bit in the bitstream. The numbers in each vertex correspond to the paper section that explains each step in detail.

configuration frames. In contrast, a typical partial bitstream contains *multiple* small, scattered writes to these registers and covers a subset of the device's frames. Listing 1 details the configuration-related packets in the full bitstream of a U200 datacenter accelerator card. The U200 is a large device with three identically-sized SLRs configured independently by changing `IDCODE`.

Pitfall 1: SLR ordering in the bitstream

The order of writes to the `IDCODE` register does *not* match the bottom-to-top order of SLRs shown in the Vivado GUI! The U200 above, for example, configures SLR1 before it configures SLR0 and SLR2.

We can use an SLR object's `CONFIG_ORDER_INDEX` property (not to be confused with `SLR_INDEX`) from Vivado to recover the order in which SLRs must be configured in a multi-SLR device. Matching the sequence of idcodes in a complete bitstream against the SLR configuration indices associates the SLRs with their idcodes.

IV. Extracting Device Parameters

We now discuss our systematic approach to determine the device-specific parameters needed to map a BEL's name to a subset of the configuration frames in the bitstream. Section V further develops this approach and will explain how to identify, within these frames, the specific frame and frame offset that map to a specific bit of the BEL's `INIT` property. Our method is hierarchical and is outlined in Figure 3.

It is essential to disable bitstream encryption and compression in Vivado [12] to permit binary analysis before undertaking any of the following steps.

A. Frame indexing

Frames are identified by address, so we must be able to locate a frame in the bitstream from its address. Frames in a full bitstream are written as a single SLR-sized array to the FDRI register with a base frame address of 0. In contrast, frames in a partial bitstream are written as multiple smaller arrays to the FDRI register with different base frame addresses. In both cases, similar to the FPGA's configuration processor, a frame can be associated with its address by keeping track of the base frame address seen when parsing a write to FAR in the bitstream, and incrementing this base address at frame-sized intervals until the array is consumed. Since a frame address consists of multiple fields (cf. Figure 2), we need to know the limits of each field to increment the address correctly. We first describe the constraints that make it possible to find these limits.

Every frame spans one clock region vertically [11], so the row-major field in the frame address must correspond to the Y-offset of the clock region in which the BEL is located. The Y-offset is *relative* to the lowest row of the SLR as frame addresses start from 0 in each SLR.

The frame address structure reveals that every major column contains multiple minor columns. Any two major columns that contain the same resource (CLBs, DSPs, BRAMs, etc.) must have the same number of minor columns; otherwise, the resources would not be homogeneous.

The block type field is located in the upper bits of the frame address and its legal values are 0b000 for CLB_IO_CLK and 0b001 for BRAM_CONTENT. As a result, though BRAM columns in a device may be physically adjacent to those of CLBs, DSPs, etc., the frame address structure shows that their contents are stored after CLB_IO_CLK frames.

Finally, the number of major columns with block type BRAM_CONTENT must match the number of BRAM columns in the device.

Given these properties, Figure 4 illustrates how frames of an SLR are ordered in the bitstream. The device-specific parameters needed to locate a configuration frame, therefore, are: (1) the number of major rows, (2) the number of major columns per row, and (3) the number of minors per major column. These three values must be derived for both the CLB_IO_CLK and BRAM_CONTENT block types. We tackle these points next.

B. Enumerating frame addresses

Enumerating all valid frame addresses in a device is the simplest way of inferring the device-specific parameters introduced in § IV-A. All valid frame addresses exist only in a full bitstream. Therefore, our discussion below assumes a full bitstream, not a partial one, is being analyzed.

Unfortunately, the bitstream does not list the valid addresses as the operations in the bitstream write the FAR register only once with a base frame address of 0 and then perform a single, large write to the FDRI register. The configuration processor auto-increments the frame address at frame boundaries without external feedback.

Devices, fortunately, often have additional features that can help with reverse engineering. In particular, Vivado is capable of generating bitstreams with per-frame CRCs [12]. Such bitstreams have two desirable properties:

1) Each frame is loaded individually.
2) Every write to FDRI is followed by a write to the CRC register. The address of the next frame is then *explicitly* written to FAR.

Generating a bitstream with per-frame CRCs for an FPGA, extracting the writes to the FAR register from its binary section, and filtering out frame addresses with a reserved block type[3], will yield all valid frame addresses in every SLR. Reconstructing the number of major rows, major columns per row, and minor columns per major column for both CLB_IO_CLK and BRAM_CONTENT block types is then straightforward.

Enumerating all valid frame addresses uncovers two bitstream navigation pitfalls which we discuss next.

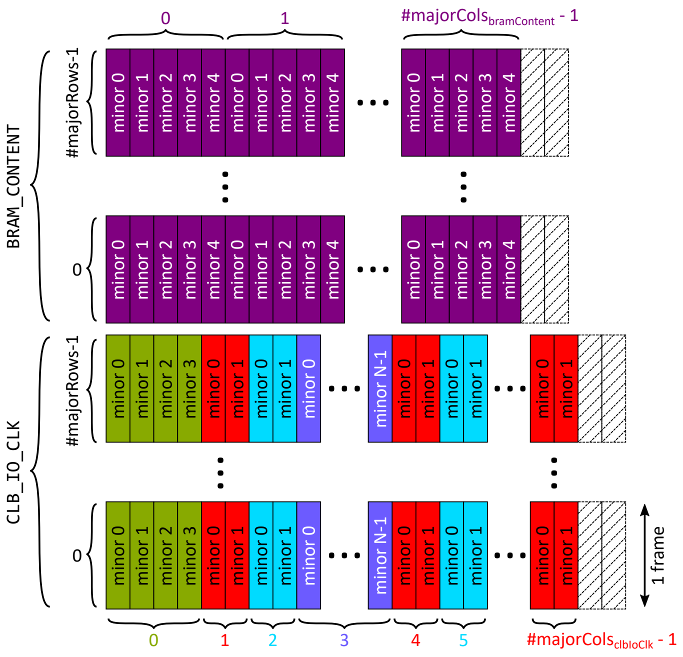

Fig. 4: The bitstream frame addressing scheme. Major row indices are shown on the left, major column indices at the top and bottom, and minor column indices in each rectangle (frame). Every color represents a homogeneous resource such as a CLB column, DSP column, etc. Frames of each block type (CLB_IO_CLK and BRAM_CONTENT) with the same major column index have the same number of minor columns, otherwise the column would not contain a homogeneous resource. The number of minors in each major column shown here are for illustration purposes only and do not reflect those in a device. The two empty frames at the end of each row are discussed in Pitfall 3.

[3]Once all frame configurations are written to FDRI, the bitstream launches the FPGA startup sequence by writing to the CMD register. This write to CMD is followed by a write to FAR with a reserved block type of 0b111. This write to FAR is irrelevant to the configuration of the FPGA as it comes after all writes to FDRI, hence why we filter it out.

Pitfall 2: Hidden major rows in devices

The number of clock regions visible vertically in the Vivado GUI is *not* always equal to the number of major rows in the device! For example, the smallest member of the Kintex UltraScale family (`xcku025`) has three rows of clock regions, but extracting its frame addresses using the method in § IV-B reveals it is, in fact, composed of five rows.

The next larger member of the Kintex UltraScale family (`xcku035`) also has five rows of clock regions, and the frame addresses of both devices are identical (at least at the binary level). It is likely that the smaller device is simply a restricted version of the larger one. This same observation can hold for other devices.

Pitfall 3: Empty frames at the end of a major row

In addition to bitstreams with per-frame CRCs, Vivado also supports generating a *debugging bitstream*. A debugging bitstream also loads each frame individually, and every write to the `FDRI` register is followed by a write of the current frame address to the `LOUT` register (instead of a write of the frame's CRC value to the `CRC` register). Writing to the `LOUT` register drives data to the `DOUT` pin of the FPGA in a serial daisy-chain configuration and is useful for debugging how far the device has advanced into its configuration.

Interestingly, the number of frames in a debugging bitstream is less than the number of frames in a standard bitstream. Creating a design where a majority of BELs on an FPGA is populated, emitting both a full and debugging bitstream from the same design checkpoint, and comparing the frames reveals that there are always two empty frames at the end of a major row in a full bitstream.

One therefore must take these two empty frames into account when incrementing frame addresses at row boundaries. We believe the frame addresses emitted by the debugging bitstream are the "real" ones used in a design and the two empty frames at the end of each major row are likely an implementation detail.

Note that debugging bitstreams exist only for standard FPGAs [12], not SoC-FPGA devices. However, we experimentally confirmed that SoC-based devices also exhibit these two empty frames at the end of each major row.

C. Mapping resource columns (CLB, BRAM, etc.) to major columns

We can locate any frame in a bitstream with a valid frame address. We now need to identify which of the major columns correspond to CLBs, BRAMs, or DSPs.

One way of extracting this information is to compare a carefully-crafted bitstream against an empty[4] one and identifying the major columns that differ between the bitstreams. The method proceeds as follows: (1) iterate over all clock regions' columns, (2) place *one* instance of a resource in every column that has a corresponding resource, (3) generate a bitstream, and (4) compare the bitstream against the empty bitstream. Many frames will differ between the generated and empty bitstreams. The major columns of these differing frames are those containing the resource of interest. It is important to avoid using the device's interconnect when constructing a design to reduce the differences when comparing bitstreams. Instantiating a resource, marking it as `DONT_TOUCH`, connecting its inputs to 0, and disconnecting its outputs, generally does the trick.

This method is appropriate for DSPs, but there is a shortcut for CLBs and BRAMs. Xilinx devices support *readback capture* [13], a mechanism to extract a bitstream from a programmed FPGA and identify the *user*-state bits of the underlying design. A user-state bit is a user-configured memory (register, LUTRAM, BRAM, BRAM register). As the bitstream format is undocumented, Vivado generates a *logic location* file that indicates the bit position of a specific user-state bit in the bitstream. In addition to the bit positions, the file conveniently contains the frame address of instantiated state bits. Therefore, one can create the same crafted bitstream as above, but instead of comparing bitstreams, parse the generated logic location file to extract the major column numbers for all user bit frame addresses. Extracting CLB major column numbers is done with a design containing a register in every CLB column, as LUTs and registers are both contained in CLBs and therefore share the same major column [11]. BRAM major columns can be determined by instantiating any block-based memory implemented using BRAMs (FIFO, etc.) in every BRAM column.

Pitfall 4: Virtual device-high CLB columns are not all fully populated

Single-SLR devices have homogeneous resource columns in each major row. Determining the resource columns in *one* row is therefore sufficient to know the resource columns in *all* other rows. However, this property does not hold for multi-SLR devices.

In multi-SLR devices, major rows at the boundary of two SLRs contain special "LAGUNA" tiles that enable signals to cross between SLRs. The LAGUNA tiles take up space previously occupied by a virtual device-high CLB column as shown in Figure 5. In other words, SLR-boundary rows contain fewer CLB resources than SLR-internal rows.

Care must be taken when looking up a slice's major column number in SLR-boundary rows to avoid erroneous frame indexing. The method described in § IV-C yields

[4]Note that Vivado will not generate a bitstream for an entirely empty design. A near-empty proxy is a single register with its inputs connected to 0.

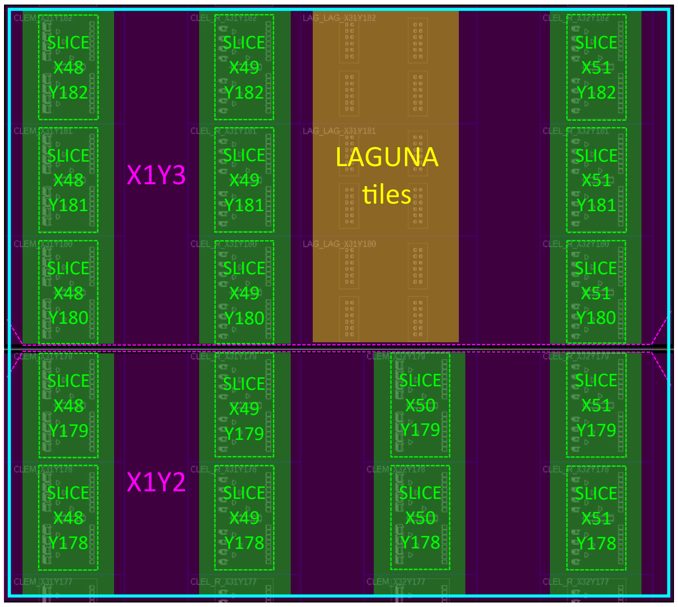

Fig. 5: Zoomed in region in cyan from Figure 1 showing a small section of the last major row in SLR0 before the SLR0–SLR1 boundary. The LAGUNA tiles in SLR-boundary rows enable inter-SLR connections and take up space previously occupied by CLB column `SLICE_X50Y*` in SLR-internal rows. As a result, the slice range `SLICE_X50Y180`–`SLICE_X50Y239` does *not* exist in the device and major row 3 in SLR0 has physically less CLB columns than the other major rows.

the exact major column number for all slices in a device, regardless of the row in which they are located.

D. Putting it together: Forming a partial frame address

We now illustrate how to construct the idcode and partial frame address for a specific BEL. We use the 6-LUT from Figure 1 (`SLICE_X136Y247/A6LUT`) as an example.

We start by determining the idcode whose frames we must search. `SLICE_X136Y247` is the 248th slice in a virtual device-high column. Knowing a clock region's height is 60 CLBs (cf. § II) and that the SLRs in the U50 are 4 clock regions high (cf. § IV-B), we infer that the slice is located in SLR1. We now know its idcode (cf. Pitfall 1).

Next, we identify the block type of the slice. We know slices are not initial BRAM contents, so the block type must be `CLB_IO_CLK`.

Then we compute the major row of the slice. We again use our knowledge of a clock region's height to infer that the slice is located in absolute row major 4. This corresponds to major row 0 when indexing is relative to SLR1.

Finally, we obtain the major column. We know the major column indices for a given resource in every row (cf. § IV-C). Looking up the 137th entry of the CLB major column indices in major row 0 yields the major column number of the slice, which is major column 262 in the U50.

V. Extracting Architecture Parameters

The work to this point derives the device-specific parameters needed to locate a frame. These parameters permit us to form three of the four elements in a frame address for a BEL: (1) block type, (2) major row, and (3) major column.

The remaining parameter is the minor column and frame offset of each bit in a BEL's `INIT` property within a major resource column. Note that these are *architectural* parameters since the structure of a specific resource column is identical among all devices that share a common architecture.

A. BRAM format

We start by selecting a BRAM column and creating a design that populates all its BELs. There are 24 18Kb BRAMs in a column with a total of 422 368 bits. Given the long time to generate a bitstream, no fuzzing-based approach can practically enumerate every bit's minor and frame offset in any reasonable amount of time.

We again exploit that BRAM contents are user-state bits to make Vivado do the work and generate a logic location file that describes the frame addresses and offsets of all 422 368 bits. Parsing the logic location file reveals the minor addresses and frame offsets.

B. CLB register and LUT format

Resource columns are formed of *tiles*. CLBs are found in four types of tiles[5]: `CLEL_L`, `CLEL_R`, `CLEM`, `CLEM_R`. In theory, the BELs in each type of tile could have a different format in the bitstream, so the experiments below need to be run for all four types of tile columns[6]. Like BRAMs, the experimental setup here consists of an entire column of a single tile type populated with LUTRAMs (if applicable for the subset of CLBs containing LUTRAMs), LUTs, and registers.

Discovering the minor and frame offsets of registers and LUTRAMs is simple as the tools consider these BELs to be user-state bits. We again use the logic location file to determine their format in a column quickly.

However, determining the minors and frame offsets of standard LUTs is more complicated as these BELs are not user-state bits, and no logic location file is available. We want to discover the format of the 64-bit LUT equation for each LUT in a column. There are 480 6-LUTs in a column and therefore 30 720 configuration bits. Though an order of magnitude smaller than the number of BRAM bits in a column, this number is large enough to make a column-high fuzzing-based approach impractical. However, hardware—in general—is highly regular. It is reasonable to expect all tiles of the same type to have the same format in the bitstream, with different offsets. We, therefore, only need to fuzz an individual tile in the CLB column. Each tile contains 8 6-LUTs and thus 512 configuration bits, a tractable number to fuzz. Fuzzing

[5]Their names differ slightly between UltraScale and UltraScale+ architectures, but they serve the purpose.

[6]All tiles that contain BRAMs are of the same type; hence we need not try multiple tile types to extract a BRAM's format.

the `INIT` property of a CLB tile requires creating a baseline bitstream against which we can compare fuzzed variants. The baseline bitstream is created as follows: (1) Select a CLB with the target tile type of interest, (2) populate all 8 LUTs in the CLB, set their `INIT` property to 0, mark them as `DONT_TOUCH`, set their inputs to 0, disconnect their outputs, (3) place and route the design, and (4) generate a design checkpoint and bitstream. The fuzzed bitstreams are then obtained by loading the baseline design checkpoint and generating 512 bitstreams in which only one bit in the 8 LUT equations is set to 1 by manipulating the LUTs' `INIT` property in Vivado[7]. Comparing these bitstreams against the baseline bitstream will yield the minor and frame offset of all 512 LUT configuration bits. Translating the minor and frame offsets of the LUTs from the fuzzed tile to the other tiles in the column requires only locating an anchor bit in each tile.

C. Putting it together: Forming a complete frame address

Having discovered the minor and frame offset of each bit in the `INIT` property of LUTs, LUTRAMs, BRAMs, and registers, we are now able to form a complete frame address for any such BEL and locate it in the bitstream. We continue the 6-LUT example we partially formed in § IV-D and seek to form a frame address for bit 0 in its LUT equation.

`SLICE_X136Y247/A6LUT` is the 7th LUT in the CLB column when counting from the bottom of SLR1. Bit 0 in this LUT's `INIT` equation is in minor column 11, at frame offset 351.

Table I extends our systematic approach to subsequent bits of the BEL's `INIT` property. Well-defined patterns can be seen for the minor column and frame offset of the configuration bits. One can then construct formulas to directly index a bit in the bitstream given its name. The same technique can be used for other resources of interest (BRAMs, LUTRAMs, registers, etc.).

TABLE I: Frame address of `SLICE_X136Y247/A6LUT` and various bits of its `INIT` property on a U50 FPGA. The minor column and frame offset of the configuration bits follow well-defined patterns.

`INIT` Index	SLR	Block Type	Major Row	Major Column	Minor Column	Frame Offset
0	SLR1	`CLB_IO_CLK`	0	262	11	351
1	SLR1	`CLB_IO_CLK`	0	262	10	351
2	SLR1	`CLB_IO_CLK`	0	262	9	351
3	SLR1	`CLB_IO_CLK`	0	262	8	351
4	SLR1	`CLB_IO_CLK`	0	262	11	350
5	SLR1	`CLB_IO_CLK`	0	262	10	350
6	SLR1	`CLB_IO_CLK`	0	262	9	350
7	SLR1	`CLB_IO_CLK`	0	262	8	350
...	...	...	...	...	...	...
60	SLR1	`CLB_IO_CLK`	0	262	11	336
61	SLR1	`CLB_IO_CLK`	0	262	10	336
62	SLR1	`CLB_IO_CLK`	0	262	9	336
63	SLR1	`CLB_IO_CLK`	0	262	8	336

[7]It is essential the baseline checkpoint be taken post-P&R to ensure all modifications of a LUT's `INIT` property when fuzzing do not modify routing and create noise when comparing fuzzed bitstreams against the baseline.

Knowledge of the full frame address and frame offset for a given bit permits a bitstream manipulation tool to read or write it.

VI. Evaluation

We implement the techniques described in § IV and § V in Python (for bitstream parsing and analysis) and Tcl (for replicating and configuring BELs in Vivado). Bitfiltrator only takes as input a target FPGA part number. No XDC constraint file is needed as none of the experimental setups use I/O pins. All experiments are conducted on a machine with an Intel Xeon E5-2680 v3 processor running Vivado 2021.1.

We use Vivado to enumerate *all* UltraScale/UltraScale+ part numbers and apply our automated flow to devices which do not require a full Vivado implementation license (devices included with the free "WebPack" license). Table II categorizes the devices by their architecture name[8]. Note that the Virtex UltraScale+ devices are all members of the Alveo datacenter-grade device family.

TABLE II: Summary of devices on which Bitfiltrator was tested. Devices are enumerated in Vivado and categorized by their architecture and family name. Bitfiltrator was only tested on devices for which bitstreams can be generated using the free Vivado "WebPack" license. No Virtex UltraScale or Zynq UltraScale+ RFSOC devices are available in the free version of Vivado.

Architecture	Vivado Name	Count
UltraScale	Kintex UltraScale	2 / 12
	Virtex UltraScale	0 / 7
UltraScale+	Kintex UltraScale+	3 / 10
	Virtex UltraScale+	8 / 31
	Zynq UltraScale+	21 / 38
	Zynq UltraScale+ RFSOC	0 / 16

The device-specific parameters extracted are: (1) the number of visible and "hidden" major rows (cf. Pitfall 2), (2) the number of major columns in every row, (3) the number of minor columns in every major column, and (4) the major columns of CLBs, BRAMs, and DSPs.

We use the smallest member of each device family as a proxy to extract the minor and frame offsets (i.e., the architectural parameters) of LUT, LUTRAM, BRAM, and register `INIT` properties to reduce bitstream sizes while fuzzing. Device-specific parameters and LUTs' `INIT` property are fuzzed in parallel over 24 cores. End-to-end runtime is under 48h. Over 90% of the total execution time is spent fuzzing LUT `INIT` properties.

We compared the architectural parameters discovered against those from project U-Ray [14], a project that reverse-engineers the bitstream format of UltraScale and UltraScale+ FPGAs to support the development of open-source FPGA

[8]The architecture names are obtained from Vivado, not marketing materials, as devices' categorization differs between both sources. The following Tcl command is used to extract the full architecture name:
`get_property ARCHITECTURE_FULL_NAME ${part}`.

toolchains. Project U-Ray provides databases that contain the minor column and frame offset of the configuration bits of *tiles* in UltraScale+ FPGAs. Our minors and frame offsets for LUT equations, LUTRAM and BRAM contents, and register values match those from U-Ray.

However, project U-Ray only provides a per-tile database of minors and frame offsets, and does not offer translated versions of these numbers for the other tiles of a resource column. Additionally, the databases only cover UltraScale+ devices, not UltraScale ones.

We, therefore, conduct an additional experiment to validate all of our device-specific and architectural parameters. We first populate *all* LUT, register, and BRAM BELs in a device. Then, we use a Tcl script to set the `INIT` property[9] of every BEL to a *random* value and generate a bitstream. Using a random value is essential as fixed or predictable patterns could hide subtle frame address indexing bugs. Finally, we use our device and architectural parameters to read the bitstream and extract the value used as the startup configuration for each BEL. If we can successfully recreate every initial value, we can assume that the derived parameters are correct. We perform this experiment for both UltraScale and UltraScale+ devices and confirm that all reconstructed configurations match those provided to Vivado. This confirmation, in turn, supports our hypothesis that all tiles of the same type have the same format in the bitstream and are translated versions of one another.

In summary, the techniques described in this paper can correctly locate and modify specific configuration bits in Xilinx's UltraScale and UltraScale+ FPGA bitstreams. We believe the same techniques will work for other product lines with minor adaptations. For example, Xilinx's 7-series FPGAs have a similar frame address format to that of UltraScale devices as the minor column, major column, and block type fields are identically-placed and sized to their UltraScale counterparts [15]. The only difference is that the major row field in 7-series FPGAs is one bit narrower than UltraScale devices, and that a "top/bottom" bit is used to select between the top and bottom halves of the device's rows. As a result, it is likely the techniques presented in this paper are directly applicable to 7-series devices. We leave this to future work.

VII. Conclusion

Bitstream manipulation is a necessary step to use an FPGA in novel ways currently unsupported by vendor tools. The undocumented FPGA bitstream file format requires reverse engineering to find the information to support this step. We detailed a systematic, top-down approach to reverse-engineer the format and location of specific cells in Xilinx FPGAs. To the best of our knowledge, this is the first paper to *explain* the foundational techniques for bitstream reverse-engineering. We believe these techniques will be valuable in reverse engineering other FPGAs and improve open FPGA design flows.

[9]For LUTs and registers we set the `INIT` property. For BRAM contents we set the `INIT_xx` and `INITP_xx` properties. For BRAM output registers we set the `INIT_A` and `INIT_B` properties.

References

[1] S. Kashani, M. Emami, and J. R. Larus. (2022, Aug) Bitfiltrator: A general approach for reverse-engineering Xilinx bitstream formats. [Online]. Available: https://github.com/epfl-vlsc/bitfiltrator

[2] S. Attia and V. Betz, "StateMover: Combining Simulation and Hardware Execution for Efficient FPGA Debugging," in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. ACM, feb 2020, p. 175–185.

[3] M. Hofmann, Z. Tang, J. Orgill, J. Nelson, D. Glanzman, B. Nelson, and A. DeHon, "XBERT: Xilinx Logical-Level Bitstream Embedded RAM Transfusion," in *Proceedings of the 2021 IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2021, pp. 1–9.

[4] K. Dang Pham, E. Horta, and D. Koch, "BITMAN: A tool and API for FPGA bitstream manipulations," in *Design Automation Test in Europe Conference (DATE), 2017*, 2017, pp. 894–897.

[5] A. Ullah, E. Sanchez, L. Sterpone, L. Cardona, and C. Ferrer, "An FPGA-based dynamically reconfigurable platform for emulation of permanent faults in ASICs," *Microelectronics Reliability*, vol. 75, pp. 110–120, 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0026271417302202

[6] *UG570: UltraScale Architecture Configuration User Guide*, Xilinx, 2022.

[7] K. Shirriff. Reverse-engineering the first FPGA chip, the XC2064. [Online]. Available: http://www.righto.com/2020/09/reverse-engineering-first-fpga-chip.html?m=1

[8] T. M. La, K. Matas, N. Grunchevski, K. D. Pham, and D. Koch, "FPGADefender: Malicious Self-Oscillator Scanning for Xilinx UltraScale+ FPGAs," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 13, no. 3, sep 2020. [Online]. Available: https://doi.org/10.1145/3402937

[9] *UG574: UltraScale Architecture Configurable Logic Block User Guide*, Xilinx, 2017.

[10] A. Nishioka and P. Freidin. (2001, nov) Tell me about the .BIT file format. [Online]. Available: http://www.fpga-faq.com/FAQ_Pages/0026_Tell_me_about_bit_files.htm

[11] *UG909: Vivado Design Suite User Guide, Dynamic Function eXchange*, Xilinx, 2021.

[12] *UG908: Vivado Design Suite User Guide, Programming and Debugging*, Xilinx, 2021.

[13] S. Tapp, *XAPP1230: Configuration Readback Capture in UltraScale FPGAs*, Xilinx, 2015.

[14] T. Ansell. (2020, jul) Project U-Ray: Xilinx UltraScale Bitstream Documentation (pre-built databases). [Online]. Available: https://github.com/f4pga/prjuray-db/tree/master/zynqusp

[15] *UG470: 7 Series FPGAs Configuration User Guide*, Xilinx, 2018.