Open-Source Bitstream Generation

Ritesh Kumar Soni,[1,2] Neil Steiner,[2] and Matthew French[2]

[1]Department of Electrical and Computer Engineering
Virginia Tech
Blacksburg, Virginia
rsoni@vt.edu

[2]Information Sciences Institute
University of Southern California
Arlington, Virginia
neil.steiner@isi.edu, mfrench@isi.edu

***Abstract*—This work presents an open-source bitstream generation tool for Torc. Bitstream generation has traditionally been the single part of the FPGA design flow that could not be openly reproduced, but our novel approach enables this without reverse-engineering or violating End-User License Agreement terms.**

We begin by creating a library of "micro-bitstreams" which constitute a collection of primitives at a granularity of our choosing. These primitives can then be combined to create larger designs, or portions thereof, with simple merging operations.

Our effort is motivated by a desire to resume earlier work on embedded bitstream generation and autonomous hardware. This is not feasible with Xilinx *bitgen* because there is no reasonable way to run an x86 binary with complex library and data dependencies on most embedded systems.

Initial support is limited to the Virtex5, but we intend to extend this to other Xilinx architectures. We are able to support nearly all routing resources in the device, as well as the most common logic resources.

I. Introduction

Field-Programmable Gate Arrays (FPGAs) are programmable hardware devices that perform no function until configured by a configuration bitstream. Modern SRAM-based FPGAs support *active*, *partial*, and *repeated* configuration, meaning that devices can be reconfigured as often as necessary, in whole or in part, while the device continues to operate. These properties provide a wide range of benefits, but some of the benefits are unintentionally limited by the available bitstream generation tools.

FPGA manufacturers are careful to protect bitstream details because their customers are concerned about the possibility of designs being extracted and reverse-engineered, and because the manufacturers would suffer severe financial loss if someone were to produce bitstream-equivalent devices at lower cost. Manufacturers provide bitstream generation tools that work well for the majority of their customers, but sometimes fall short of what researchers need. Nonetheless, attempts to understand or reverse-engineer bitstreams are consistently discouraged by the manufacturers [1].

But many uses of bitstream information are legitimate and benign. Our intent is not to extract anything from bitstreams or vendor software, but simply to refactor the bitstream generation capability for special purposes, and we are open-sourcing this capability in that vein. Two cases of particular interest from prior work [2] are rapid bitstream modification and embedded bitstream generation:

Rapid bitstream modifications for customization, dynamic tuning, interactive debugging, or autonomous control require numerous small changes to designs. We wish to incrementally describe these changes in XDL physical netlists, and rapidly turn them into partial bitstreams. In traditional flows, each design change requires rebuilding the design, placing and routing it, and re-generating the bitstream. And even for very small partial bitstreams, Xilinx *bitgen* generates a full bitstream and compares it to some reference bitstream before it can generate the partial bitstream. Our *micro-bitstream assembly* approach instead allows an application to convert small XDL changes directly into partial bitstreams that can be written to files or to a device ICAP port for active self-reconfiguration.

Special capability is similarly required for embedded bitstream generation. Many reconfigurable applications use pre-generated bitstreams, but some applications need to dynamically generate bitstreams at runtime. A tool like *bitgen*, as an x86 executable with significant data and OS dependencies, is not suitable for use in most embedded systems. The special constraints of embedded systems require a bitstream generator with a modest memory footprint that can be compiled for available hard- or soft-processors. Pairing micro-bitstream assembly with Torc will allow us to embed bitstream generation within autonomous and other special purpose systems.

This work takes a novel approach to basic bitstream generation that does not require reverse-engineering. The approach facilitates rapid generation of bitstreams for small designs, and is suitable for use in embedded systems.

Section II provides background and related work. We present our hypothesis and approach in sections III and IV. We discuss the details of library creation in Section V and of bitstream generation in Section VI. And finally, we present results in Section VII, before concluding in Section VIII.

II. Background

We begin by discussing prior work and related work, with particular emphasis on the tools that we build upon. We then include an overview of bitstream structure to provide context for the reader.

A. Prior Work

Independent bitstream generation for Xilinx devices is not without precedent. JBits [3] was the first known tool able to modify existing bitstreams for XC4000 and later for Virtex and Virtex2, but it was unable to generate complete bitstreams from scratch. A completely rewritten update extended that support to include VirtexE, Virtex2P, Spartan2E, and Spartan3, and was able to generate complete bitstreams with the same fidelity as *bitgen*. That extended capability was never officially acknowledged or released, and was simply described as a "Device API" [2], but it was successfully embedded into a real hardware system and used for the purpose of autonomously modifying itself while running.

A number of research groups have developed bitstream generation capabilities for internal purposes, but have not drawn attention to those capabilities. This is true of Wires-on-Demand [4], qFlow [5], and ERDB [6]. Others have simply manipulated bitstreams at the frame granularity without providing any generation capability of their own [7].

Work by Silva and Ferreira [8] sounds conceptually similar to ours, in that it assembles bitstreams out of discrete components, but it generally does so at a very different granularity and is not suitable for arbitrary bitstream generation. Silva and Ferreira are specifically interested in fast embedded bitstream generation for directed acyclic graphs, and their work consequently carries a number of performance-related restrictions. The bitstreams are built from components like "adders, comparators, and multipliers" that may not overlap and are placed in a reserved dynamic region. These components must be placed in vertical stripes, and connectivity is only permitted between adjacent stripes, based upon a defined subset of routing resources. In practice the approach is closer to late-binding of components [9] than it is to the generation of arbitrary bitstreams that we need.

The most concerted and successful published bitstream reverse-engineering effort came from a tool named *debit* by Note and Rannaud [10]. *Debit* provided substantial capabilities for Virtex2, Virtex4, Virtex5, and Spartan3, with anticipated extension to Altera architectures, but seems to have attracted too much unfavorable attention. The host site http://www.ulogic.org/trac was permanently removed from service in summer of 2010.

B. Related Work

Our work relies heavily upon Torc [11] for XDL support, bitstream frame and packet processing, and device exploration. Torc is an open-source C++ infrastructure and toolset for reconfigurable computing, intended for custom research applications, CAD tool development, architecture exploration, or applications that need to work with real device data. Torc can be used for research in synthesis, mapping, placing, and routing, and for productivity enhancements, power optimization, radiation tolerance, security, and other domains that are sometimes overlooked by the commercial industry.

Torc includes four main APIs. The Generic Netlist API provides an object model and read/write capabilities for unmapped EDIF netlists. The Physical Netlist API provides an object model and read/write capabilities for mapped XDL netlists. The Device Architecture API provides exhaustive

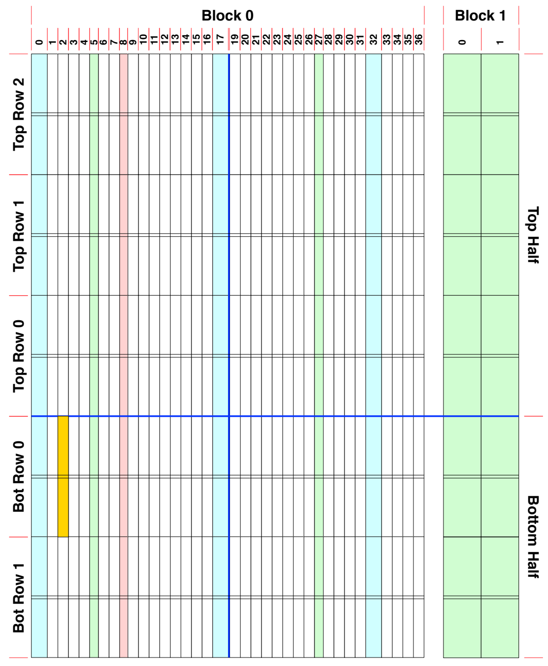

Fig. 1. XC5VLX30 configuration space drawn to scale. The bitstream consists of block types 0 and 1, where 1 is only used for BRAM content. The device is divided into top and bottom halves, each with multiple rows. Clock regions are bounded by rows and by the center of the device. Column numbers are displayed along the top. IOB columns are blue, DSP columns are red, BRAM columns are green, and CLB columns are white. A clock word runs through the center of every row. The highlighted area at coordinates <block 0, bottom half, row 0, column 2> consists of 36 frames.

logic and wiring descriptions for numerous Xilinx architectures. And the Bitstream Frames API provides read/write capabilities for configuration bitstreams down to the frame granularity. No information is provided about bits inside frames, except as documented in the various Xilinx configuration guides. Torc also includes tools for routing, placement, and other CAD functions.

C. Bitstream Structure

Modern FPGAs are arranged as heterogeneous two-dimensional arrays described by a tile map. There is a correspondence between the tile map and the configuration space of the device as it exists in the bitstream, but that correspondence is generally complex. In Xilinx architectures beginning with Virtex4, a device is divided into top and bottom "*halves*" that are not always of the same height—see Figure 1. Those halves are further divided into *rows* of equal height that contain two horizontally adjacent *clock regions*. The rows are further subdivided into *columns* and then into *frames*—the smallest addressable part of the FPGA configuration. Together with one additional coordinate called a *block type*, the half, row, column, and frame define a unique *frame address*.

The size of the frames depends upon the device or architecture that they belong to, but remains constant across a particular device.[1] Every configuration frame in the device also has an associated frame address that determines its location.

[1]Spartan-6 I/O frames are a known exception to this rule.

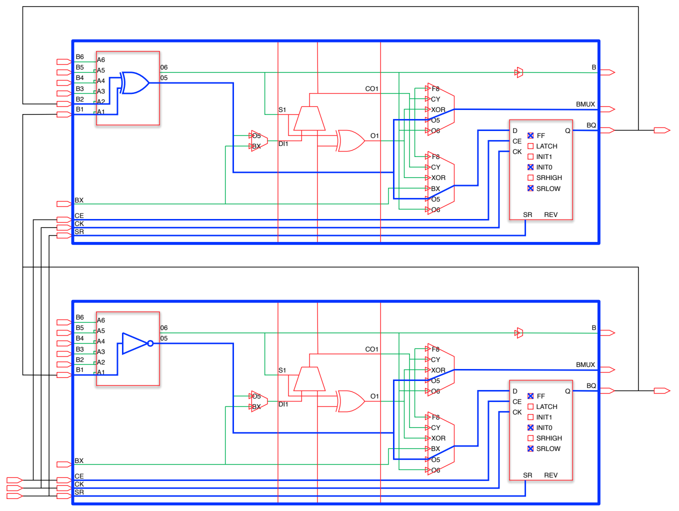
Fig. 2. Example 2-bit counter built out of coarse-grained blocks. The upper block consists of a preconfigured XOR gate with synchronous and asynchronous outputs. The lower block consists of a preconfigured inverter with synchronous and asynchronous outputs. Both of these would be preconfigured primitives that could be instantiated but not modified, and could form part of a Turing-complete set. This approach is simple and portable but scales very poorly and does not efficiently use the underlying hardware resources.

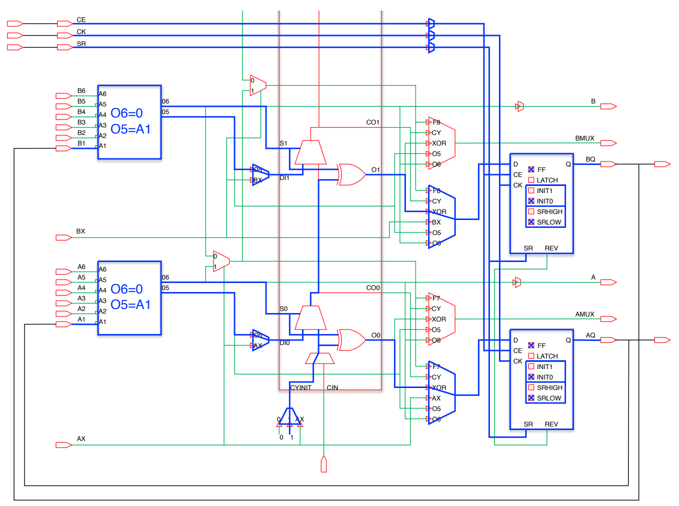
Fig. 3. Example 2-bit counter built out of fine-grained blocks. The blocks are *architectural primitives*. Arbitrarily complex LUT masks are composed from input passthrough functions according to the equations given. The circuit looks as complicated as the coarse-grained version for this very simple example, but is far more flexible, scales linearly, and makes efficient use of the high-speed carry chains. When architectural primitives are used, there is no need for custom mapping, placing, or routing.

But the size of frame address fields vary from one architecture to another, and there is no universal frame address structure that can be used across architectures.

Columns in the bitstream correspond to logic columns in the tile map, and vary in width according to their underlying tile types. Many columns in the tile map are not separately addressable in the bitstream and have zero effective width, as in the case of interconnect tiles. The tile map may show adjacent columns for interconnect and logic, such as INT + CLB, INT + DSP, and INT + BRAM, but only the CLB, DSP, and BRAM columns exist in the bitstream addressing.

Bitstream files consist of a header and a collection of *packets* of various sizes. Most packets read from or write to configuration controller registers, the most important of which is the *frame data register*. A packet can write a single frame, a contiguous set of frames, or the full configuration space of the device. Multiple packets can be used when discontiguous sets of frames need to be written, as is often the case during partial reconfiguration.

All of the bitstream information discussed here is supported by Torc. Our work adds the ability to configure frame contents with XDL logic and routing settings.

III. Hypothesis

Definition: *A* micro-bitstream *is a building block of configuration data—logic or routing or both—that can be used to compose a larger function or design.*

Hypothesis: *A valid bitstream of arbitrary complexity can be composed by offsetting and logically OR-ing a suitable set of micro-bitstreams.*

This hypothesis was derived from the simple fact that an empty bitstream consists mostly of logic zero bits, suggesting that logic one bits generally turn things on. We tested this non-rigorously in hardware with a simple design: We removed a number of settings from one XDL design and inserted them into another XDL design, and generated bitstreams for both. Neither bitstream worked correctly by itself, but when we merged the two bitstreams by logically OR-ing their frame data, we obtained the original functionality.

IV. Approach

Our approach in micro-bitstream assembly was constrained by our requirement to avoid reverse-engineering of bitstream information, while still providing a useful capability to the research community.

This approach encompasses primitive selection, library creation, bitstream generation, and some accompanying implementation details.

A. Primitive Selection

Our hypothesis makes it clear that user bitstreams can be composed of micro-bitstreams which need not correspond to *architectural primitives* like PIPs and logic settings—The set of primitives only needs to be Turing complete. Figure 2 shows an example of a circuit implemented with coarse-grained primitives, while Figure 3 shows the same circuit implemented with fine-grained primitives.

If the set of micro-bitstream primitives does not correspond to architectural primitives, the design must first be mapped to the primitive set, and subsequently placed and routed within the custom architecture that they define. Figure 4 shows the general bitstream generation case, where the micro-bitstream primitives are not necessarily architectural primitives. In the special case of a one-to-one correspondence between micro-

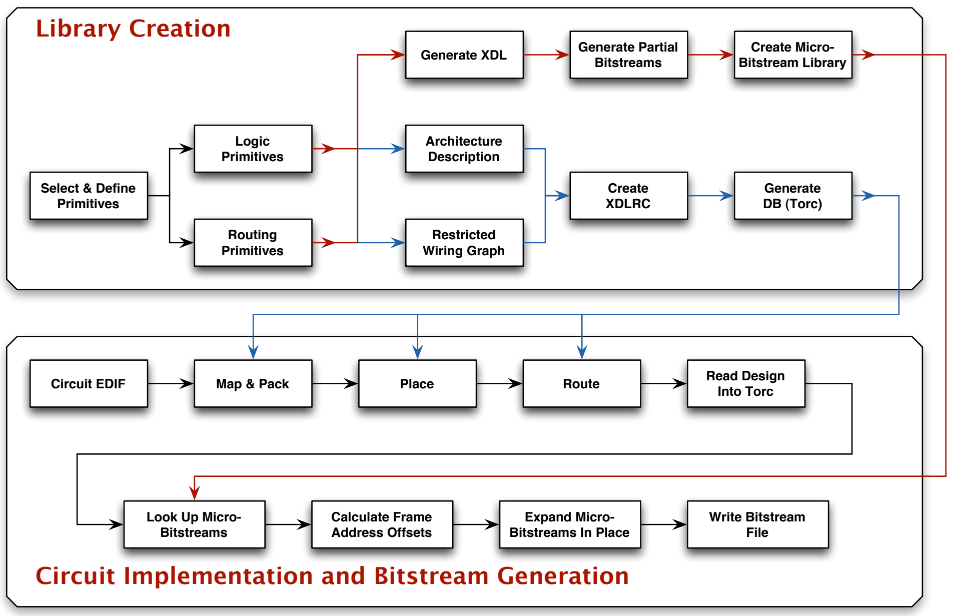

Fig. 4. General bitstream generation flow. The red path shows the creation of the micro-bitstream library and its subsequent use in bitstream generation for user designs. The blue path shows the general case in which primitives do not have a one-to-one correspondence with actual device logic and wiring. In the general case, a custom device database must be created, and a special mapper, placer, and router are required to target the virtual device.

bitstream primitives and architectural primitives, the path shown with blue arrows becomes unnecessary.

B. Library Creation and Bitstream Generation

Our current implementation uses architectural primitives, so there is no need for special device databases or special mapping, placement, or routing tools. The top of Figure 5 shows the library creation flow as implemented, and the bottom of the figure shows the bitstream generation flow. While this process is considerably simpler than that shown in Figure 4, there are still legitimate reasons—*simplicity and speed of implementation*—why some applications might choose not to use architectural primitives.

C. Resource Model and Implementation

Our resource model is that of the device itself. This data includes every routing PIP and every configurable setting described in the XDLRC data, which we obtain indirectly through Torc's device databases.

Much of the design can be blindly mapped from XDL settings to corresponding micro-bitstreams, as long as the resource is supported in our library. We would have used Torc's device database extensively if we needed to map, place, or route the design, but under our resource model the bitstream generation process mainly uses the tile map and its coordinate system.

D. Torc Changes

We added a few utility functions to Torc's Virtex5 Bitstream Frames API in the course of this work. These functions primarily facilitate looking up configuration frame data by frame address and by XDL coordinates. Frame addresses are the natural coordinate system for all bitstream information, but XDL coordinates are more natural for design information.

The XDL functions can also return the range of bits within the requested frames that correspond to the desired tile. There are some assumptions inherent in this process because the configuration guides do not discuss tile boundaries in frames, but it is reasonable to work from what *is* documented: look up the frame height, remove the middle clock word, and divide the remaining bits by the number of tiles in the frame.

One additional function helps to map interconnect tiles to their associated logic tiles. Bitstream frame addressing does not provide separate addresses for interconnect columns: Those columns share an address with the primary logic column that they support. In Virtex4, Virtex5, and Virtex6 architectures, the tile map has interconnect tiles immediately to the left of their corresponding logic tiles. In 7-Series architectures, those interconnect columns alternate between the left and right sides of their logic tiles.

V. Library Creation

Micro-bitstream library creation is a multi-step process. We begin by iterating over our selected primitives and creating an XDL design for each setting. We then convert these XDL designs to NCD and use *bitgen* to generate corresponding bitstreams. Each bitstream is compared to a reference bitstream, and commonalities are discarded, so that what remains is the effect of the XDL primitive. We compress the resulting micro-bitstreams, and stitch them into the library.

A. Primitive Selection

With our decision to use architectural primitives, we now consider those primitives more closely. The Virtex5 architec-

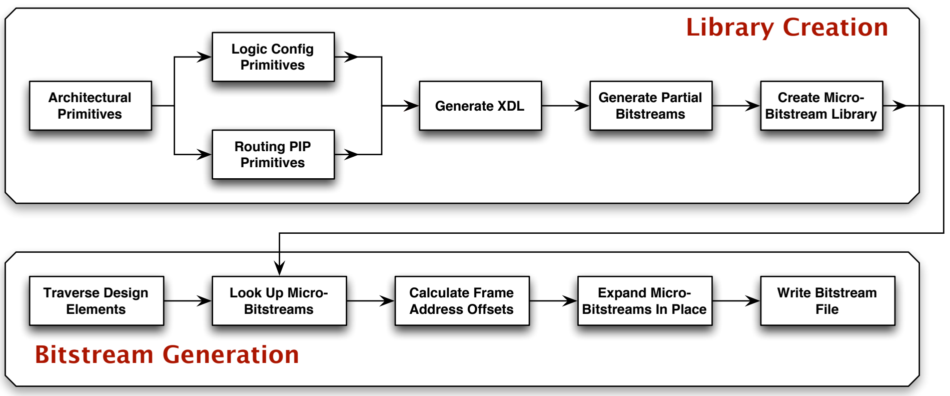

Fig. 5. Simplified bitstream generation flow. Our work currently uses architectural primitives as micro-bitstream primitives, so there is no need for special device databases or special mapping, placement, or routing tools. The bitstream generation process traverses the user design, looks up micro-bitstreams for each supported setting in the design, calculates their proper offsets, expands them in place into the device frames, and writes the resulting bitstream file.

ture includes 111 tile types, each of which is instantiated in one or more devices in the family, and most of which contain routing PIPs. Within a subset of those tiles types are also 53 logic site types, most of which have configurable resources.

Routing support includes all PIPs in CLB and INT tiles and over 50 types of clock tiles. Logic support includes SLICEL, SLICEM, RAMB36_EXP, and most DSP48E sites. Selection of primitives was influenced by multiple criteria, including usage in typical designs: Logic sites SLICEL and SLICEM and routing PIPs in INT tiles are used by every design, so supporting these sites and tiles was a top priority. Clock, BRAM, and DSP primitives were similarly selected because of their prevalence.

B. Examples

When processing logic and routing in XDL designs, we must accommodate different kinds of settings. Examples and characteristics of these settings are provided below.

1) Routing PIPs: Nets are bounded by source and sink pins, and are composed of routing PIPs. Only the routing PIPs affect the bitstream.

```
net "blink",
    outpin "blink" DQ,
    inpin "blink" D6,
    pip CLBLL_X16Y59 L_DQ -> SITE_LOGIC_OUTS3,
    ...;
```

The highlighted PIP in this example specifies a connection between wires `L_DQ` and `SITE_LOGIC_OUTS3` in the tile named `CLBLL_X16Y59`.

2) Logic Settings: Logic settings are always associated with logic sites, each of which may contain zero or more configurable resources. Most resources can take on one of several possible settings or values. In the example below, `CLKINV` is a configurable mux that passes a clock signal either inverted or uninverted.

```
inst "blink" "SLICEL", placed CLBLL_X16Y59
    SLICE_X27Y59, cfg "
    CLKINV::CLK
    D6LUT:blink_i:#LUT:O6=~A6
    DFF:blink:#FF
    DFFMUX::O6
    ...";
```

Resources `DFF`, `DFFMUX`, and `D6LUT` are special cases that we discuss next.

3) LUT Equations: LUT equations are a special case because their settings do not come from a predefined list. They are instead expressed as boolean functions of their inputs and the constants 0 or 1. The inputs are named A1 through An, n being the degree of the LUT.

Virtex5 slices have four fracturable LUTs—named A, B, C, and D—that can generate separate functions of 5 and 6 inputs. The `D6LUT` example above configures the O6 output of the D LUT in LUT mode—as opposed to RAM or ROM mode. The `O6=~A6` equation indicates that its output is the complement of the `A6` input.

A 6-input LUT has 64 memory bits and can therefore take on 2^{64} values. There is no feasible way to generate micro-bitstreams for each of these settings. But it is possible to generate micro-bitstreams for functions `O6=A1`, `O6=A2`, `O6=A3`, `O6=A4`, `O6=A5`, `O6=A6`, `O6=1`, and `O6=0`, and to compose the desired function at runtime by applying the equation in bitwise fashion to these micro-bitstreams.

4) LUT RAM Masks: LUTs can also be configured in RAM or ROM modes, in which case a hexadecimal LUT mask takes the place of a LUT equation. Instead of generating a micro-bitstream for each of the 64 bits in a LUT, we use the Logic Allocation (LL) File that *bitgen* creates when given the '`-l`' (ell) flag. The LL file can be parsed and used to identify the relative frame address and offset of each bit in the LUT. The LUT mask can then be applied in bitwise fashion to each of the configuration bits.

```
inst "lutram" "SLICEM", placed CLBLL_X16Y59
    SLICE_X27Y58, cfg "
    A6LUT::#RAM:O6=0xAC52660033A966F1
    ...";
```

5) BRAM Initialization: BRAM data uses hexadecimal initialization strings for both parity and data, in the same manner as LUT RAM masks. For 16,384 + 8,192 bits of content and parity, the amount of data is much larger, but the LL file provides the same location information as for LUT masks.

C. Dependencies

Certain dependent resources have no effect on the bitstream unless their associated primary resource is instantiated in the design. We detect this condition incidentally after iterating over the possible settings for the resource if none of the resulting bitstreams differ from the reference bitstream. `DFFMUX` is a good example: This configurable multiplexer drives the input of flip-flop `DFF`, but if `DFF` is not instantiated, then `DFFMUX` has no effect upon the bitstream regardless of its setting. To properly generate micro-bitstreams for resources with dependencies, the reference bitstream must include the dependency. Some logic resources depend upon other logic resources, while some depend upon the presence of a net driving or driven by the resource. To accommodate these situations, resources with dependencies use a *harness* as described below.

D. Special Settings

Some resources are configured with complex or arbitrary strings, while others are configured by complex code. For example, the Virtex5 device data does not enumerate valid I/O standards for IOBs, so generation of micro-bitstreams for those settings is not automated. As another example, the rules that determine whether IOBs must be configured as VREF pins are dynamically evaluated by *bitgen* based upon the entire design and the I/O standards used in each I/O bank. The first example is easy to resolve because the list of supported I/O standards is published, but the second example is difficult to resolve without reverse-engineering, and is consequently not supported in this work.

E. Logic Site Harnesses

When we find all of the micro-bitstreams for a given logic resource to be empty—indicating that each of them is identical to the reference bitstream—we infer that some dependency must be present. The observed behavior suggests that resources are turned off in the bitstream if they do not drive any other logic or routing resources. There are also a few cases where *bitgen* seems to want input drivers.

We address these cases in two ways: Firstly, we create a harness net that connects to every input and output pin of the logic site of interest. The fact that such a net is nonsensical and unroutable is irrelevant. Secondly, we instantiate all of the primary site resources—all LUTs and flip-flops in the case of slices—regardless of which ones may be implicated by the resource of interest. These two steps seem to generate enough signal path to satisfy the dependencies and prevent the dependent resources from being discarded during bitstream generation. This approach allows us to create and reuse a single harness per site type.

F. Micro-Bitstream Generation

Every logic resource can take one or more settings in addition to the special `#OFF` value. Logic settings are already explicitly grouped by resource in the device architecture data. Groups of routing resources are simply configurable muxes, and are easy to infer from the set of PIPs sharing the same sink wire. The Torc architecture data provides all of this information, and allows us to iterate through devices, tiles, logic resources, logic settings, and routing PIPs.

When generating micro-bitstreams for routing PIPs, we create a sample design, a sample site instance, and a sample net. The instance only serves to provide endpoints for the net, and the net contains only the PIP of interest. We iterate over all the PIPs belonging to the same configurable mux, and generate XDL files for each one in turn. We then convert each of the XDL files to NCD with the Xilinx *xdl* tool, and convert each NCD file into a bitstream with *bitgen*. Finally, we read the reference and generated bitstreams, compare their corresponding frames, and diff the frames that are not identical.

When generating micro-bitstreams for a logic resource, we create a sample design and instantiate a logic site to host the resource. If the resource requires a harness, we insert it. We then iterate over every valid setting for the resource and create a corresponding XDL file. If a harness was used, we create one additional XDL file as a reference design, with the harness inserted and the resource set to `#OFF`. The XDL files are converted to NCD and then to bitstreams, as with routing resources, and frame differences are retained.

The resulting micro-bitstreams consist of sparse binary data that is highly compressible, typically by two or more orders of magnitude.

G. Library Organization

The compressed micro-bitstream data is stitched together into a single library for convenience. The internal structure of the library file is as follows:

```
Tile Type Count
Tile Type 1
    Resource Count
    Resource 1
        Config Count
        Config 1
            Micro-Bitstream Data
            ...
```

H. Micro-Bitstream Validation

The library was validated by creating test designs, generating bitstreams for those designs, and comparing the bitstreams to the corresponding *bitgen* output with the help of Torc. Individual XDL designs were created for every supported logic resource and for a sample of routing PIPs. Complex designs combining large numbers of these micro-bitstreams were also generated and tested to validate our core hypothesis.

We note that Torc does not currently calculate and update frame ECC values, so we ignore the “clock” word at the center of each frame when comparing bitstreams.

VI. Bitstream Generation

Our current bitstream generation process was shown in Figure 5. Since this process essentially merges micro-bitstreams together, we internally call our tool *bitmerge*. The input to *bitmerge* is a placed and routed XDL design, or subset thereof. *Bitmerge* traverses the design and processes the resources one by one. Micro-bitstreams for each supported design element are fetched from the primitive library, positioned according to frame and word offsets, and merged into the base frame set, which may be empty or may have been read from an existing bitstream. These steps are detailed below.

A. Design Traversal

Bitmerge uses Torc to traverse the XDL design, first visiting all instances, and then visiting all nets. For placed instances of supported site types, the frame and word offsets are calculated from the site placement location, and every configuration setting for that instance is processed. For every routed or partially routed net, the PIPs are traversed and the frame and word offsets are calculated from each PIP's tile coordinates.

B. Resource Processing

Processing is simple for most resources: *Bitmerge* reads the appropriate configuration settings, looks up each corresponding micro-bitstream in the library, and merges it by OR-ing the bits into place.

1) Routing PIPs: For routing PIPs, *bitmerge* looks up the tile type and fetches the appropriate micro-bitstream using the source and destination wires as the key. The micro-bitstream is expanded and merged in place with the base frame set.

2) Logic Settings: For most logic settings, *bitmerge* looks up the logic site and resource and fetches the appropriate micro-bitstream using the setting as the key. The micro-bitstream is expanded and merged in place.

3) Special Case: LUT Equations: LUTs configured in LUT mode use a boolean equation as the setting. The output is assigned a function of the LUT inputs and the constants 0 or 1. The library contains micro-bitstreams for each variable and each literal: `D6LUT::#LUT:O6=A1`, `D6LUT::#LUT:O6=A2`, `D6LUT::#LUT:O6=1`, and so on. The desired function is formed by applying the boolean expression to the appropriate micro-bitstreams.

To generate the configuration bits for a LUT equation, *bitmerge* parses and evaluates the expression. When the code encounters a variable or literal in the equation, it fetches the corresponding micro-bitstream from the library, expands its set of frames, and pushes it onto a stack. When the code encounters a boolean operation, it pops two frame sets from the stack, applies the operation in bitwise fashion to the frames, and pushes the resulting frames back onto the stack. When the parsing completes, the only set of frames remaining on the stack is merged with the base frame set.

4) Special Case: Hex Strings: A few resources use fixed length hex strings as values. The library contains micro-bitstreams for each individual bit position of applicable resources, so *bitmerge* fetches these using the bit position as the key. It expands and merges each of these into place. For example, the hex string `0x38` can be composed by OR-ing the micro-bitstreams for bits `0x20`, `0x10`, and `0x08`.

C. Frame Address Calculation

Torc can create an empty frame set for any supported device. Data from full or partial bitstreams can be loaded into those frames, and *bitmerge* can modify the frame contents. Each frame in the set is implicitly mapped to its corresponding frame address, and every frame includes a modification flag. *Bitmerge* consequently does not need to calculate frame addresses, but instead relies on Torc to return the proper frames for each requested tile.

D. Bitstream Assembly

As each resource is processed, the appropriate micro-bitstreams are merged into the base frame set. When all resources have been processed, the resulting frames are wrapped into bitstream packets and written to a bitstream file.

VII. Results

Bitmerge can be evaluated in terms of device and tile type coverage, extensibility and portability, runtime performance, and size.

A. Resource Coverage

Bitmerge supports logic sites of type SLICEL, SLICEM, DSP, and BRAM. Except for some DSP locations, all resources in these sites are supported, and most of the logic in real designs can be implemented with these sites. For routing, *bitmerge* supports INT, CLBLL/CLBLM, and all clock tiles. These routing tiles cover the majority of the routing resources in any device. Without logic support for IOBs and global clock buffers, *bitmerge* cannot yet create bitstreams for complete designs. Many additional logic resources can be supported, but some others cannot because they would require reverse-engineering. A simple solution is to configure any unsupported resources in a base bitstream that is then imported by *bitmerge*.

B. Extensibility and Portability

Bitmerge currently supports only Virtex5 devices. Extending support to other architectures will require the creation of suitable harnesses for every logic site, but the rest of the process should remain the same.

C. Runtime Performance

We tested runtime performance for a single PIP change, for a single logic resource change, and for a large design, on a workstation with a 3 GHz Intel Xeon 5160 and 4 GB of memory. The large design targets the XC5VFX130T and includes $20,518$ configurable instances with $405,431$ logic settings, and

TABLE I
Bitmerge PERFORMANCE RESULTS.

Design	*bitmerge* (s)
Single Routing PIP Design	2.5
Single Logic Setting Design	2.5
Large Design (100 % full XC5VFX130T, 20,518 logic sites)	216.0

92, 227 nets with 1, 425, 932 PIPs. The design also utilizes 100 % of the slices in the device. 19 of the 20, 518 configurable instances in the design are not currently supported—primarily items like IOBs, DCMs, and clock buffers.

Table I shows the runtime performance of *bitmerge*. License agreement terms preclude benchmarking *bitmerge* against Xilinx software, but we note that *bitmerge* compares quite favorably, particularly when the data originates in XDL form.

D. Library Size

The micro-bitstream library is 738 KB in size. This includes everything necessary for logic resources in SLICEL, SLICEM, DSP48E, and RAMB36_EXP sites, and for routing PIPs in INT, CLBLL/CLBLM, and clock tiles. In the future, we expect to *gzip* this data as Torc already does for its device databases.

VIII. Conclusion

We have described an open-source bitstream generator for Torc that requires no reverse-engineering of tools or configuration bitstreams. The motivation originates from two critical needs of ours: The ability to quickly make a large number of customizations to existing bitstreams, and the ability to embed bitstream generation inside the system that it targets.

Making bitstream changes from inside Torc will allow us to amortize the overhead of program startup, database initialization, and file I/O. Making bitstream changes from inside an embedded system will allows us to perform dynamic tuning, or to change the system autonomously while it is running. With this capability we can resume past work on hardware autonomy, leading to systems that are far more flexible, resilient, and in control of their own operation.

Our current approach creates a library of micro-bitstreams corresponding to architectural primitives, and combines them into arbitrarily complex designs by expanding and OR-ing their frame contents. The input is XDL data, with an optional base bitstream, and the output is a new or modified bitstream.

While this capability does not support the full set of device resources, it is sufficient for changes to the vast majority of the device. Our routing PIP coverage can be extended to nearly 100 %. Our logic resource coverage can be extended up to a point with the approach described in this paper, but there is a subset of resources and settings that would require reverse-engineering. Those resources and settings are excluded from future work.

Our micro-bitstream library currently supports Xilinx Virtex5 devices, but our approach can be extended to other Xilinx architectures. In practice, every architecture has its own peculiarities, especially at the bitstream level—asymmetries in the number of top and bottom clock regions, or CLB mirroring in 7-Series devices, for example. Even for regular parts of other architectures, it will still be necessary to build site harnesses to enable dependent resources.

Future work will include tighter integration with Torc, support for additional architectures, and better compression of our library. We are currently planning tests and an application that will more thoroughly exercise our bitstream generation and its integration with Torc.

IX. Acknowledgments

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR001-11-C-0041. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA. Approved for Public Release, Distribution Unlimited.

References

[1] A. Megacz, “A library and platform for FPGA bitstream manipulation,” in *Proceedings of the 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM 2007, (Napa, California), April 23–25*, 2007, pp. 45–54. [Online]. Available: http://dx.doi.org/10.1109/FCCM.2007.10

[2] N. J. Steiner, “Autonomous computing systems,” Ph.D. dissertation, Virginia Tech, March 2008. [Online]. Available: http://scholar.lib.vt.edu/theses/available/etd-04102008-194601

[3] S. Guccione, D. Levi, and P. Sundararajan, “JBits: Java based interface for reconfigurable computing,” in *Proceedings of the Second Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference, MAPLD 1999, (Laurel, Maryland), September 28–30*, 1999.

[4] P. Athanas, J. Bowen, T. Dunham, C. Patterson, J. Rice, M. Shelburne, J. Suris, M. Bucciero, and J. Graf, “Wires on demand: Run-time communication synthesis for reconfigurable computing,” in *Proceedings of the 17th International Conference on Field-Programmable Logic and Applications, FPL 2007, (Amsterdam), August 27–29*, 2007, pp. 513–516. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4380705

[5] T. Frangieh, R. Stroop, P. Athanas, and T. Cervero, “A modular-based assembly framework for autonomous reconfigurable systems,” in *Reconfigurable Computing: Architectures, Tools and Applications*, ser. Lecture Notes in Computer Science, O. Choy, R. Cheung, P. Athanas, and K. Sano, Eds. Springer Berlin / Heidelberg, 2012, vol. 7199, pp. 314–319. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-28365-9_26

[6] K. Kępa, F. Morgan, and P. Athanas, “ERDB: An embedded routing database for reconfigurable systems,” in *Proceedings of the 21st International Conference on Field-Programmable Logic and Applications, FPL 2011, (Chania, Crete), September 5–7*, 2011, pp. 195–200. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/FPL.2011.43

[7] E. L. Horta and J. W. Lockwood, “Automated method to generate bitstream intellectual property cores for Virtex FPGAs,” in *Field Programmable Logic and Application*, ser. Lecture Notes in Computer Science, J. Becker, M. Platzner, and S. Vernalde, Eds. Springer Berlin Heidelberg, 2004, vol. 3203, pp. 975–979. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-30117-2_110

[8] M. L. Silva and J. C. Ferreira, “Creation of partial FPGA configurations at run-time,” in *Proceedings of the 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools, DSD 2010 (Lille, France), September 1–3*, 2010, pp. 80–87. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5615638

[9] C. Lavin, M. Padilla, J. Lamprecht, P. Lundrigan, B. Nelson, and B. Hutchings, “HMFlow: Accelerating FPGA compilation with hard macros for rapid prototyping,” in *Proceedings of the 19th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM 2011 (Salt Lake City, Utah), May 1–3*, 2011, pp. 117–124. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5771262

[10] J.-B. Note and Éric Rannaud, “From the bitstream to the netlist,” in *Proceedings of the 2008 ACM/SIGDA 16th Annual International Symposium on Field-Programmable Gate Arrays, FPGA 2008 (Monterey, California), February 24–26*, 2008, pp. 264–264. [Online]. Available: http://doi.acm.org/10.1145/1344671.1344729

[11] N. Steiner, A. Wood, H. Shojaei, J. Couch, P. Athanas, and M. French, “Torc: Towards an Open-Source Tool Flow,” in *Proceedings of the 2011 ACM Nineteenth International Symposium on Field-Programmable Gate Arrays, FPGA 2011, (Monterey, California), February 27–March 1*, 2011, http://torc.isi.edu.