Open-Source Bitstream Generation for FPGAs

Ritesh K Soni

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

Peter M. Athanas, Chair
Patrick R. Schaumont
Mark T. Jones

Date: August 9, 2013
Blacksburg, Virginia

Keywords: FPGA, bitstream, low-level assembly, open-source

Open-Source Bitstream Generation for FPGAs

Ritesh K Soni

(ABSTRACT)

Bitstream generation has traditionally been the single part of the FPGA design flow that has not been openly reproduced. This work enables bitstream generation for "limited" resources without reverse-engineering or violating End-User License Agreement terms. Two use cases in particular have motivated this work—embedded bitstream generation and fast bitstream generation for small changes in design—both of which are not feasible with the Xilinx's bitstream generation tool.

The approach is to first define a set of primitives which can implement an arbitrary digital design and create a library of *micro-bitstreams* of the primitives. An input design is then mapped to the set of primitives and a bitstream for the design is generated by merging the corresponding *micro-bitstreams*. This work uses architectural primitives. Initial support is limited to the Virtex-5 and Virtex-7 family of FPGAs from Xilinx, but it can be extended to other Xilinx architectures. Nearly all routing resources in the device, as well as the most common logic resources are supported by this work.

Acknowledgments

First and foremost, I am sincerely thankful to my advisor, Dr. Peter Athanas, for giving me an opportunity to be a part of the Configurable Computing Lab. His approach to research is very motivating and I have learnt a lot working under him. I would also like to thank Dr. Mark Jones and Dr. Patrick Schaumont for serving as members of my committee. The courses that I took under Dr. Jones and Dr. Schaumont in the first semester of my Masters laid a good foundation for my graduate studies.

My special thanks to Neil Steiner of USC-ISI for his guidance during my internship there. This thesis is a continuation of the work I did during the internship and the central idea of this work is Neil's brainchild. He has also been very helpful with Torc support without which this work would not be successful.

I am thankful to my parents and my brother, who have been there for me through thick and thin and encouraged me to do my best in this research.

I would like to thank my lab mates; Krzysztof, for sharing his knowledge of bitstreams; Kaiwen, for his advice on the writing work; Kavya, for her advice with all the official work, and the rest of the CCM family, for the bike rides, the lunch sessions, and all the interesting discussions.

I am thankful to all my friends in Blacksburg for the fun times outside work. I am also thankful to all my other friends around the world whose confidence in me have kept me going.

I am thankful the the VT Ping Pong club; the regular practice sessions helped me keep fit and maintain a competitive spirit.

I am grateful to God for everything.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Field-Programmable Gate Arrays (FPGAs) are configurable digital integrated circuits that can be programmed to implement arbitrary digital functions. FPGAs consist of a matrix of configurable logic blocks, I/O blocks, embedded memory, routing switches, and other dedicated functional blocks. The configurable resources can be programmed to one or more predefined options, which is controlled by configuration bits called a *bitstream*. The *bitstream* contains information about how each resource on an FPGA is to be configured and it is used to program the FPGA. Modern Xilinx 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 [1].

FPGA designers usually don't have to know about the bitstream layout or how it is created. They enter the design in a Hardware Description Language (HDL), which is converted to a bitstream after a series of steps as shown in Figure 1.1. A design in HDL is first synthesized and then mapped to a given FPGA device. The mapped design only informs what type of

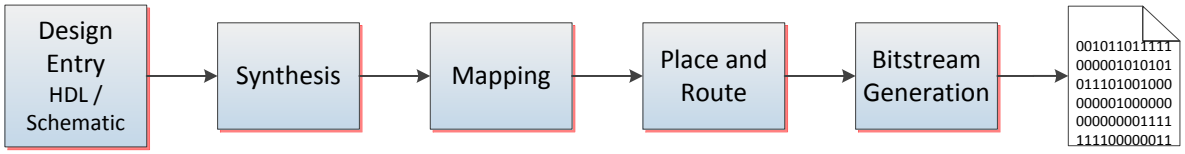


Figure 1.1: FPGA design flow

logic resources of the FPGA will be used and how they will be configured. Next, the design is placed to select the logic resources on the FPGA chip and then routed to connect the placed logic resources. The placed and routed design contains all information about how the FPGA resources should be configured to implement the design. Finally, the placed and routed design is converted to *bitstream*, which is used to program the FPGA.

Bitstream generation is the only part of the FPGA design flow that has not been openly reproduced; open-source tools are available to map, place, and route an HDL design. The Verilog-to-Routing (VTR) Project [2] is a popular open-source tool which can synthesize, pack, place, and route a Verilog design. The reason for lack of open-source bitstream generation tools is that the FPGA manufacturers do not disclose details of the bitstream structure [3]. 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 [4].

This work presents a way to generate bitstreams that does not require reverse-engineering. The idea is to map a design to a set of predefined primitives, and generate bitstream for the design by merging bitstreams of the constituent primitives.

1.1 Motivation

Though the bitstream generation tool provided by vendors work great, there are some use-cases for which the tool is not suitable. Two possible use cases that motivated this work are given below.

1.1.1 Embedded Bitstream Generation

Many reconfigurable applications, specially FPGA based autonomous systems, might need to dynamically generate bitstreams at runtime [5]. Any fault tolerance application can make use of embedded bitstream generation capability. Such an application can detect faulty resources in FPGA, re-place, re-route the design accordingly, *generate a bitstream*, and configure the device with the new bitstream, all without help of any external resource. The Xilinx tool for bitstream generation, called *bitgen*, is not suitable for use in most embedded systems as it is an x86 executable and it has significant data and OS dependencies. 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.

The bitstream generation tool created in this work is entirely based on the Torc library [6], which is known to run on an embedded system. Thus, by extension, this tool can also be compiled to run on any embedded system.

1.1.2 Fast Bitstream Generation

Design scenarios like customization, dynamic tuning, interactive debugging, fault injection or autonomous control require numerous small changes to designs. In traditional flows, even though the changes are small, each design change requires rebuilding the design, placing

and routing, and re-generating the bitstream. And even for very small partial bitstreams, Xilinx *bitgen* generates a full bitstream and compares it to a reference bitstream before it can generate the partial bitstream. For small changes in a design, bitstream generation for the changes should happen rapidly.

The approach used in this work allows an application to convert small design changes directly into partial bitstreams.

1.2 Contributions

A bitstream generation API in C++ has been created as a part of this work. This API will be available as a part of the open-source library Torc [6]. Initial support is provided for Virtex-5 and Virtex-7 family of Xilinx FPGAs, but can be extended to other families of Xilinx FPGAs. Bitstream generation is not supported for *all* resources present in the two family of FPGAs; the supported resources are listed in Table 1.1 [1]. These tiles can implement most of the logic part of a design and can cover more than 90% routing resources of an FPGA.

	Supported tile types
Logic	CLB, DSP, and BRAM
Routing	INT and CLBLM

Table 1.1: Resources supported

The custom bitstream generator is also demonstrated to run on an embedded system based on `XUPV5-LX110T` development platform[7].

[1] Xilinx FPGA resources are grouped into *tiles* of different types; he tile types supported in this work are listed in the table.

1.3 Thesis Organization

This thesis is organized as follows.

Chapter 2: Background provides information on Xilinx FPGA layout, bitstream structure, and XDL file format, knowledge of which will help understand this work better. A short introduction to Torc library, on which this work depends heavily, is also given.

Chapter 3: Prior Work discusses the previous efforts made on independent bitstream generation.

Chapter 4: Hypothesis and Approach begins with the hypothesis on which this work is based and then goes on the explain the general approach to create a bitstream generator based on the hypothesis.

Chapter 5: Implementation Details explains all the steps of bitstream generation as implemented in this work.

Chapter 6: Results and Analysis evaluates the bitstream generator API on aspects of resource coverage, fidelity, extensibility, portability, run-time performance, etc.

Chapter 7: Conclusion concludes this work.

Chapter 2

Background

This chapter first discusses the Xilinx FPGA architecture and then the Xilinx bitstream structure, both based on the Virtex-5 family. Format of an XDL file, which is an input to the bitstream API, is discussed next. Finally, an introduction is given to the open-source library Torc [6] which is used by this work.

2.1 FPGA Architecture

The resources of Xilinx FPGAs are grouped together into *tiles*. Depending on functionality of the resources, tiles can be of different types—routing, logic, memory, clock, input/output, etc. The Virtex-5 architecture includes 111 tile types, each of which is instantiated in one or more devices in the family, and most of which contain routing resources [8]. Table 2.1 lists some of the tile types present in Virtex-5 FPGAs along with their functionality.

Tile Type	Functionality
INT (Interconnect)	Routing resources
CLB (Configurable Logic Block)	Logic resources
BRAM (Block RAM)	Block RAM
DSP (Digital Signal Processing)	Dedicated multipliers, barrel shifters, etc.

Table 2.1: Virtex-5 tile types

2.1.1 Tile Layout

A Xilinx FPGA is arranged as a two-dimensional array of heterogeneous tiles described by a tile map. The tile layout is very regular and in most of the chip one column contains tiles of one type only. As the tiles of different types vary in size, each column contains a different number of tiles. Figure 2.1 shows a conceptual tile layout of an FPGA based on the Virtex-5 architecture. The sequence of tile types along the row differs for different devices. For example, the device `xcv5lx50` of the Virtex-5 family starts with an IOB column, then a BRAM column, followed by six CLB columns, and so on.

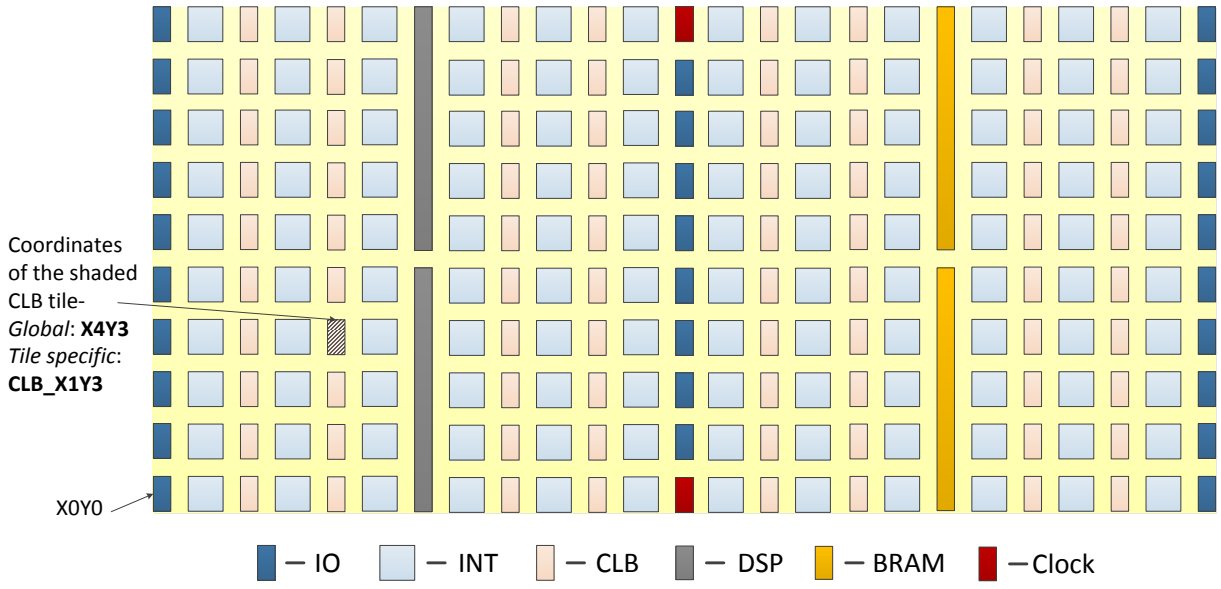


Figure 2.1: Simplified FPGA tile layout

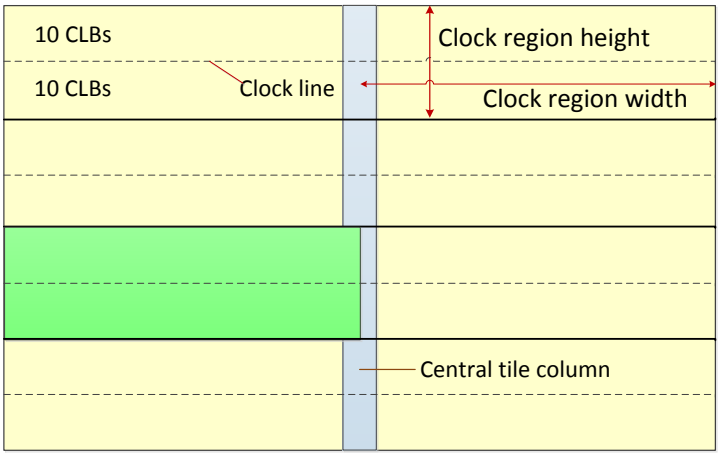


Figure 2.2: Partitioning of an FPGA into clock regions. In the FPGA shown above, there are eight clock regions; the green region is one of the eight clock regions. The central tile column is included in either the right or the left clock region.

Ever since the Virtex-4 series, Xilinx FPGAs have been partitioned into clock regions for efficient and zero skew distribution of the global clock signal [9]. The width of a clock region spans half the chip area [8] and the height of a clock region, in terms of tile count of a particular tile type, remains fixed for a given family of FPGA. In the Virtex-5 family the height of a clock region is 20 CLB tiles. Figure 2.2 shows clock regions in an FPGA. The number of clock regions varies across devices in a family. Clock regions are relevant to this work because of the way they map to bitstreams.

There is a global coordinate system to identify each tile on a FPGA. There is also a coordinate system for each tile type. Therefore, each tile gets two coordinates—one from the global coordinate system and one from the coordinate system of its type. Information about the tile layout of any Xilinx FPGA can be obtained from the Xilinx tool *xdl* [1] [10]. This tool produces out both the global and the tile specific coordinates for every tile in the device and also details of the resources present in the tiles. Torc has a database of the tile layout and the resources for the devices it supports.

[1] xdl -report -pips -all_conns <device_name>

In short, an FPGA can be depicted as tiles of different types and sizes arranged regularly on a plane with one column containing tiles of one type only. Each tile can be uniquely identified by a tile coordinate system. The tiles are further grouped into rectangular regions, called clock regions.

2.1.2 Tile Resources

A routing tile, also called a routing multiplexer or routing switch, consists of programmable resources to make connection between wires linked to the tile. These programmable resources are called Programmable Interconnect Points or in short PIPs. Figure 2.3 shows a conceptual routing tile, where the wire A can be connected to wires B, C, or D and each of these connections is controlled by a separate PIP. Some of the tile types containing routing resources are—INT, CLBLM, and CLK.

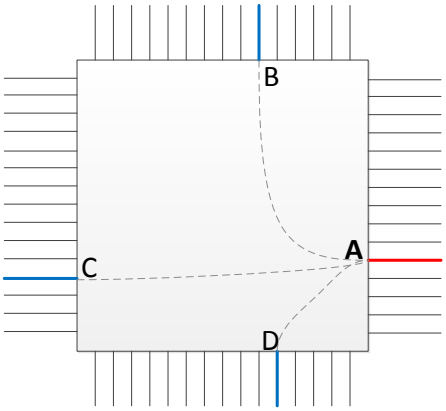


Figure 2.3: A routing tile

Resources in logic tiles are further grouped into *sites*. For example, in Virtex-5 FPGAs the logic tile CLBLL contains two sites of type SLICEL and a DSP tile contain two sites of type DSP48E and routing resources [11]. The logic sites contain different types of configurable resources. Figure 2.4 shows a SLICEL site as present in the Virtex-5 FPGAs.

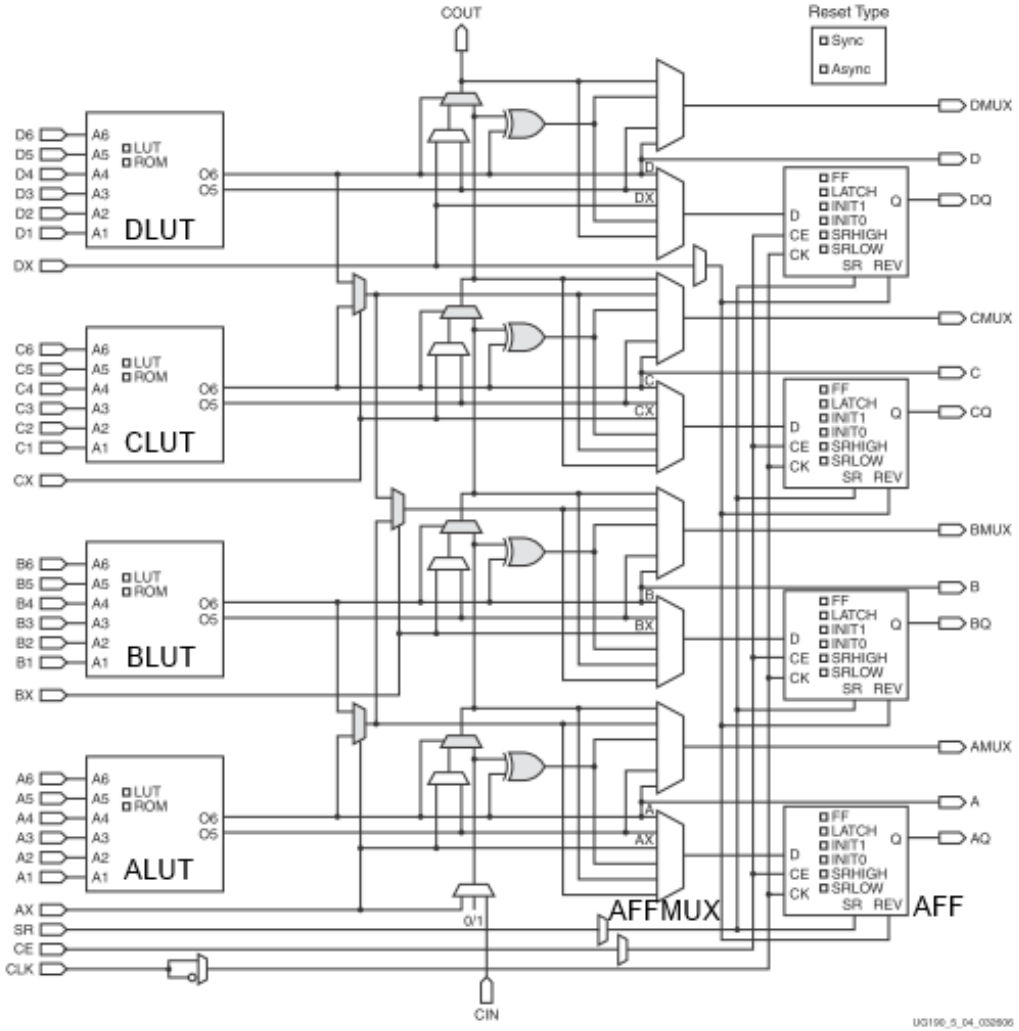


Figure 2.4: A SLICEL site. This figure has been taken from Xilinx User Guide 190 [8]

The slice has four look-up tables (LUTs)—A, B, C, and D—each of which can be configured to implement an arbitrary Boolean function of six inputs or two arbitrary Boolean functions of five inputs each. The LUTs in another site type, SLICEM, can be configured as memory elements also. SLICEL has four storage elements which can be configured either as edge triggered D flip-flop or level-sensitive latches. Each storage element has two more configurable attributes that control its initial state and set/reset behavior. The site also has a few configurable multiplexers to control data flow within the slice. For example, the input

of the storage element AFF is driven by the configurable multiplexer AFFMUX, which can be configured to pass one of the six input signals.

The configurable resources present in DSP and BRAM sites are mainly multiplexers and attributes. Some of the attributes that can be configured in a BRAM site are read/write widths of ports, initial content of the BRAM site—both data and parity, and address extension.

2.2 Bitstream Structure

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 the Virtex-4 family, a device is divided into top and bottom halves in the configuration space—see Figure 2.6. The top and bottom halves may not be of the same height. Those halves are further divided into rows of equal height, such that the rows align with the clock regions of the tile layout. The rows are divided into columns, where each column in bitstream corresponds to a column in tile layout. The columns are further divided into frames—the smallest addressable part of the FPGA configuration space. Together with one additional coordinate called a block type, the half, row, column, and frame define a unique frame address [12]. Figure 2.5 shows composition of a frame address.

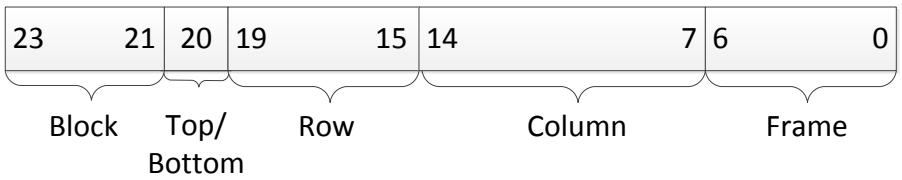


Figure 2.5: Composition of a frame address word for Virtex-5 bitstreams

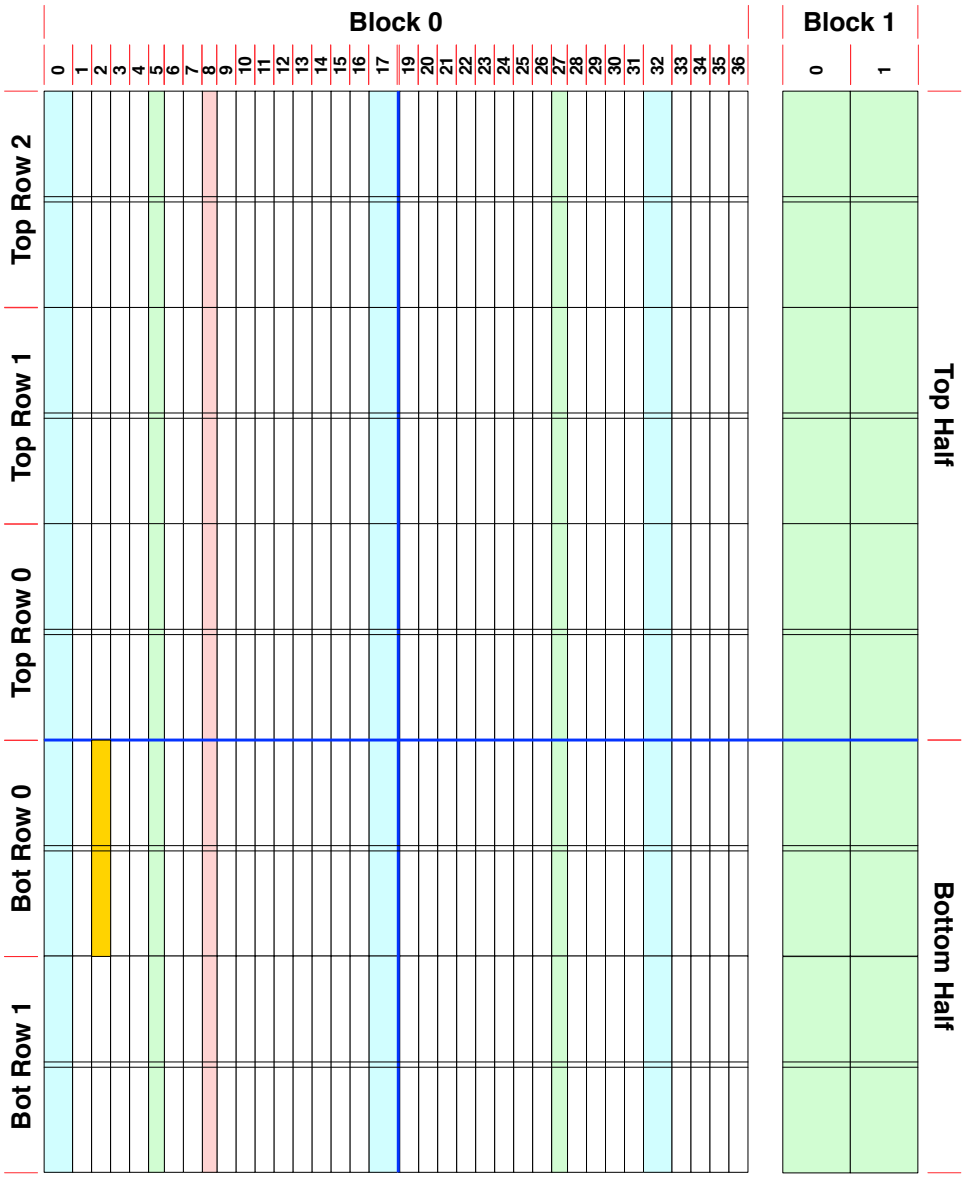


Figure 2.6: Bitstream structure for device XC5VLX30 drawn to scale. The bitstream consists of block types 0 and 1, where 1 is only used for BRAM content. The device is divided into top and bottom halves, each with multiple rows. Clock regions are bounded by rows and by the center of the device. Column 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.

A bitstream frame spans the height of a clock region in the sense that a frame affects all tiles in a column within a clock region. Frames can be thought of as a vertical stack of bits placed along the height of a clock region and a few of these stacks put side by side make a bitstream column—see Figure 2.7. The size of the frames depends upon the device or architecture that they belong to, but remains constant across a particular family. Size of a frame in Virtex-5 family bitstreams is 1312 bits, whereas in Virtex-7 family bitstreams, size of a frame is 3232 bits [13].

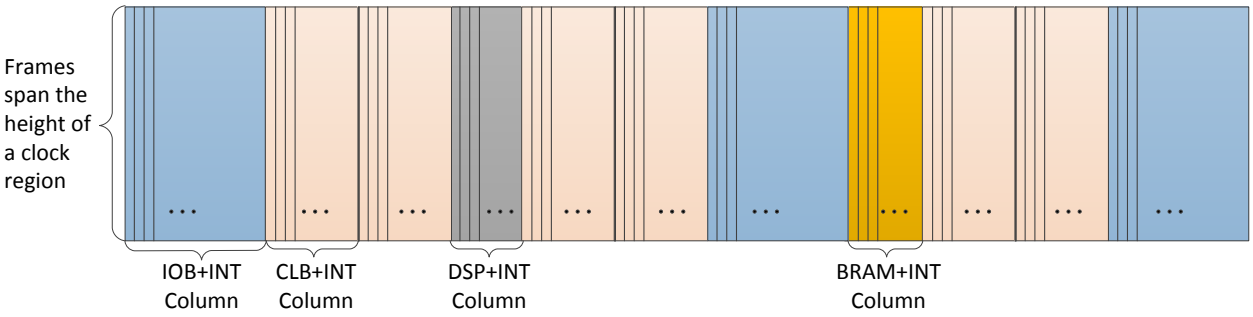


Figure 2.7: A row of bitstream frames. A row is divided into columns which are further divided into frames. A frame spans the height of a clock region. The width of a column depends on the tile type of the associated column in the FPGA tile layout.

Columns in the bitstream correspond to logic columns in the tile map, and vary in width according to their underlying tile types—see Table 2.2. Many columns in the tile map are not separately addressable in the bitstream 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 and these columns contain configuration data for adjacent INT tiles also.

Bitstream files consist of a header and a collection of packets of various sizes as shown in Figure 2.8. The header contains meta-data like design name, file modification time, target device, etc. The packets contain configuration commands and configuration data. The configuration commands read from or write to configuration controller registers and drive a

Column type	Width in frames
CLB	36
DSP	28
Block RAM	30
IOB	54

Table 2.2: Virtex-5 bitstream column width

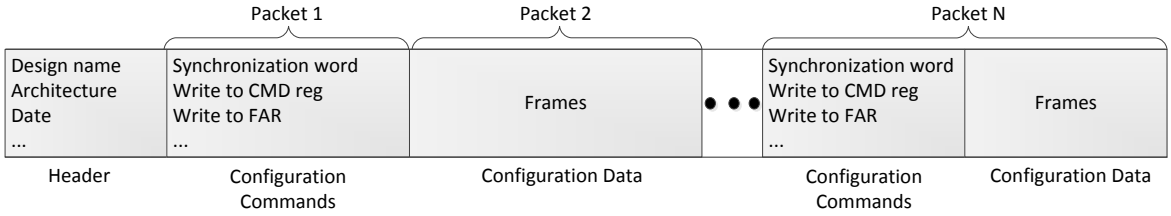


Figure 2.8: Simplified view of a bitstream file. Configuration commands and data can be in same or different packets.

small state machine in the FPGA. The configuration data can be a single frame, a contiguous set of frames, or the full configuration space of the device. Each frame is directed to the right location on the FPGA by the Frame Address Register (FAR) present on the FPGA. When a contiguous set of frames are written, the FAR is set for the first frame and then it increments automatically. Multiple packets can be used when discontiguous sets of frames need to be written, as is often the case during partial reconfiguration.

Correspondence between FPGA tile layout and bitstream configuration space at the column level can be derived from the information released by the Xilinx configuration user guides and the information collected from Xilinx tools on FPGA layout and bitstream frame addresses. As discussed earlier, the tile layout information of an FPGA can be obtained from the Xilinx *xdl* tool. The Xilinx bitstream generation tool, *bitgen*, gives the option to generate debug bitstreams that has every frame tagged with its FAR address. By combining these pieces of information, mapping from a tile coordinate to a bitstream column can be created.

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

2.3 XDL File Format

The input to bitstream generator created in this work is a design in XDL format. Xilinx Design Language (XDL) is a human readable file format provided by Xilinx to represent a design netlist after mapping to the FPGA primitives. An XDL design can be optionally placed and routed. Xilinx provides the *xdl* tool to convert an XDL file to a NCD file, the binary equivalent of XDL format, which can be converted to a bitstream using *bitgen*. An NCD file can be converted back to XDL file using the same *xdl* tool. XDL provides a very powerful interface to Xilinx devices, and is very popular in research community [10].

Listing 2.1 shows a placed and partially routed XDL design. The XDL header in Line 1 declares the design name, `system`, and the device, `xc5vfx130t`, on which the design will be implemented. The part number of the device is `ff1738` with a speed grade of -2. After the header comes the body of the XDL, which contains *instances*, *nets*, and optionally *modules*.

```
1  design "system" xc5vfx130tff1738-2 v3.2 ;
2
3  inst "blink_counter" "SLICEL", placed CLBLM_X36Y139 SLICE_X56Y139 ,
4    cfg " A5LUT::#LUT:O5=0 A6LUT::#LUT:O6=(A6+~A6)*((A1@~A4))
5         ACY0::O5 AFF:id:#FF AFFINIT::INIT0 AFFMUX::XOR AFFSR::SRLOW" ;
6
7  inst "LED" "IOB", placed RIOB_X66Y23 AL7 ,
8    cfg " DIFFI_INUSED::#OFF DIFF_TERM::#OFF IMUX::#OFF OUSED::0 PADOUTUSED::#OFF
9         PULLTYPE::#OFF TUSED::#OFF OUTBUF:LED_OBUF: PAD:LED:
10          DRIVE::12 OSTANDARD::LVCMOS25 SLEW::SLOW " ;
11
12 net "clk_BUFGP" ,
13   outpin "blink_counter" AQ ,
```

```
14    inpin "LED" O ,
15    pip CLBLM_X47Y75 SITE_CLK_B1 -> M_CLK ,
16    pip CLBLM_X47Y76 SITE_CLK_B1 -> M_CLK ,
17    pip INT_X47Y75 GCLK0 -> CLK_B1 ,
18    pip INT_X47Y76 GCLK0 -> CLK_B1 ;
```

Listing 2.1: Example of an XDL design

The logic sites or tiles used by a design are called *instances* in XDL and specified by the keyword *instance* or *inst*. Line 3 introduces an instance named `blink_counter`. The instance is of site type SLICEL and placed on location SLICE_X56Y139. The following three lines describes how different resources of this instance are configured. The format for specifying a resource setting is `<resource>:<identifier>:<setting>`, where the middle part, `<identifier>`, is optional and does not affect bitstream. For example, `AFF:id:#FF` means resource `AFF` is configured as `#FF`(flip-flop). Similarly, `A5LUT::#LUT:O5=0` means resource `A5LUT` is configured in `LUT` mode and its output `O5` is assigned to 0. Configuration value `#OFF` means the resource is not configured. The resources present in the site but not mentioned in the XDL design are either considered as not configured, or they get a default value depending on configuration of connected resources. The XDL design in Listing 2.1 contains another instance, of site type IOB, declared in Line 7.

After instances, netlists (nets in short) are declared. A net is declared by the keyword *net*, followed by the net name. Every net has a source pin and one or more sink pins, where each pin belongs to an instance declared earlier in the XDL design. For example, the inpin in Line 14 is mapped to pin O of instance “LED”. A source pin is connected to sink pins by configuring Programmable Interconnect Points (PIPs) in one or more routing tiles. In XDL, PIPs are specified by keyword *pip* followed by the tile location and the two wires connected in the tile. For example, Line 15 of listing 2.1, `pip CLBLM_X47Y75 SITE_CLK_B1 -> M_CLK`,

specifies a PIP located on tile CLBLM_X47Y75 and connecting the wires SITE_CLK_B1 and M_CLK. A net can have zero pips configured, in which case the net is completely unrouted.

The input to the bitstream generator created in this work is a fully placed and routed design in XDL. A placed and routed XDL design provides details of which tiles on FPGA are used by the design, and how resources in the tiles are configured. Torc provides APIs to read/write XDL files and also traverse and modify all the parts of an XDL design.

2.4 Torc Library

This work uses the Torc [6] library for XDL support, bitstream frame and packet processing, and device exploration. Torc uses 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 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 logic and wiring descriptions for numerous Xilinx architectures. The Bitstream Frames API provides read/write capabilities for configuration bitstreams down to the frame granularity. No information is provided about bits inside the frames, except as documented in the various Xilinx configuration guides. Torc also includes tools for routing, placement, and other CAD functions. This work uses the Physical Netlist API, Device Architecture API, and Bitstream Frame API.

A few utility functions were added to Torc's Virtex-5 and Virtex-7 Bitstream Frames API to support this work. These functions primarily facilitate looking up configuration column 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. In the case of Virtex-5, this is 41 words minus one clock word, divided by 20 CLB tiles, or two words per tile—a total of 64 bits.

One additional function helps to map the interconnect tiles to their associated logic tiles in the configuration space. 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 Virtex-4, Virtex-5, and Virtex-6 architectures, the tile map has interconnect tiles immediately to the left of their corresponding logic tiles. In 7-Series architectures, the interconnect columns alternate between the left and right sides of their logic tiles.

The bitstream generation API created in this work will be release as a part of Torc in future.

2.5 Summary

This chapter discussed FPGA architecture, bitstream structure, XDL file format, and the Torc library. Knowledge of these topics will help to understand this work better.

Chapter 3

Prior Work

Independent bitstream generation has been attempted before and different approaches have been taken for it. The approaches can be put in three categories—bitstream format released, bitstream relocation, and bitstream reverse-engineering.

3.1 Bitstream Format Released

In this category, the details of the bitstream format, including mapping from FPGA resources to bits in the bitstream, are known to the developer of the bitstream generation API/tool. JBits [14] falls in this category. It was developed by Xilinx and Virginia Tech with the aim of providing better software support for run-time reconfiguration. The earlier version of JBits provided APIs to modify existing bitstreams at the level of per configuration for XC4000 and later for Virtex and Virtex2, but it was unable to generate complete bitstreams from scratch. A completely rewritten update extended 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" [5], but it was successfully embedded into a real hardware system and used for the purpose of autonomously modifying itself while running. JBits was not supported for the later FPGA families.

JBits consists of two parts—APIs giving read/write access to the bitstream at the resource level and a library of hardware modules in the form of Java classes. A Java application can use the JBits APIs to get/set bits in a bitstream corresponding to individual resources. Two example functions of the JBits API are given in Listing 3.1. JBits also provided APIs to read back configuration from a device and write a partial bitstream to a device to facilitate dynamic reconfiguration. The library contains parameterizable and relocatable modules of common functions like adder, multiplier, counter, etc. Using JBits requires knowledge of the low-level FPGA architecture.

```
1 /* Configure the F LUT of the Slice 0 of row, col to be XOR */
2 set(row, col, Slice0_FLUT, XOR);
3 /* Get the value of the Clock Input at row,col */
4 c = get(row, col, ClockInput);
```

Listing 3.1: JBits API example

Another Java library, called `abits` [3], was released for Atmels FPSLIC series of FPGAs after detailed documentation of the bitstream format for the series was published [15]. The `abits` library, similar to the JBits library, gives access to a bitstream at the resource level. Three applications were created to demonstrate the use of the bitstream API—live debugging, self-timed circuits, and frequency division of layout sensitive signals.

3.2 Bitstream Relocation

Another approach taken for bitstream generation is to create a library of partial bitstreams for modules and routing resources, and stitch these partial bitstreams, after relocation, to generate bitstream for a design. This approach has been used for various purposes, ranging from fast bitstream generation [16], to run-time reconfiguration [1], to creating bitstream IP [17]. A limitation common to all the bitstream generators taking this approach is that they don't support bitstream generation for arbitrary digital functions.

Work by Silva and Ferreira [1] uses the bitstream relocation approach, in that it assembles bitstreams out of discrete components. The components are of very coarse granularity and so the work 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 restrictions to simplify placement and routing and hence improve performance. 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 [18] than it is to the generation of arbitrary bitstreams that this work aims.

A similar approach was taken by Hübner et al. [19] to increase flexibility of run-time reconfiguration. They use partial bitstreams of prerouted modules with interfaces based on LUT-based communication primitives [20]. Configuration data of the modules is extracted from full bitstreams using the JBits API. The modules can be placed on any of the valid locations in allocated vertical slots and connection between modules is made by configuring routing primitives in vertical channels adjacent to the vertical slots. Even though the routing

primitives were of fine granularity, routing for a net between modules was restricted to be within the allocated vertical channels.

The central idea of Horta's work [17] is to create IP cores at bitstream level, or Bitstream IP (BIP) as mentioned in their work. A tool, called PARBIT, extracts partial bitstream from a full bitstream such that the partial bitstream can be placed in a limited region of any device in the family to which the full bitstream belongs. Koch and Teich [21] use a conceptual FPGA layout and bitstream structure to elaborate the idea of extracting partial bitstreams for modules by correlating with a base bitstream, and also shifting of a partial bitstream to relocate the module on the FPGA. Though the conceptual FPGA layout and bitstream structure used by them resemble Xilinx's FPGA tile layout and bitstream structure, they never implemented a bitstream generator for a real device.

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 [22], tFlow [16], and ERDB [23]; all three can generate configuration data for routing PIPs. Others have simply manipulated bitstreams at the frame granularity without providing any generation capability of their own [17].

This thesis is an extension of the recent work by Soni, Steiner, and French [24]. This work takes a similar approach but at much finer granularity of modules. As such, this work supports bitstream generation for arbitrary designs.

3.3 Bitstream Reverse-Engineering

Bitstream reverse-engineering is another way to go about creating independent bitstream generation tool, but it appears to defy the vendor's end-user license agreement. Reverse

engineering bitstreams can help in gathering bitstream details, which are otherwise withheld by vendors, necessary for creating a bitstream generation tool. These details can be used to create a bitstream generation tool like `abits`.

One of the early efforts on bitstream reverse-engineering came from a tool named *debit* by Note and Rannaud [4]. They started with a few assumptions on regularity of Xilinx FPGAs and corresponding bitstreams and applied a cross-correlation algorithm on few selected bitstreams to obtain mapping between the routing resources and bitstream bits. Using this mapping database, they provided a bitstream generation tool, called *xdl2bit*, which could generate configuration data for routing resources and some resources of slices for Virtex2, Virtex4, Virtex-5, and Spartan3. They obtained the bitstream mapping for LUTs form Xilinx tools, but they haven't mentioned how they got the mapping information for other resources in slices. The host site of *debit*, http://www.ulogic.org/trac, was permanently removed from service in summer of 2010.

Though not endorsed by FPGA vendors, reverse-engineering is till attempted by researchers. BIL [25] extended the *debit* work by improving the algorithm to correlate XDL data to bitstream data and expand the bitstream analysis capability. Ding et. al. [26] devised a composite analysis method to create mapping from configuration bits to resource configuration and created a tool named Bit2NCD to convert bitstreams back to NCD files. Even though one of the goals of this work is to create a tool to modify bitstreams, the results focus only on converting a bitstream to an NCD.

Though the research on bitstream reverse engineering focuses on converting bitstreams back to netlists, creating a bitstream generation tool is a possible outcome of such a research. But as reverse engineering of bitstream violates end-user license agreement [27], a bitstream generation tool created out of such a research might not be released.

Chapter 4

Hypothesis and Approach

This chapter first introduces the hypothesis on which this work is based and then gives a overview of the generic bitstream generation approach based on the hypothesis.

4.1 Hypothesis

Before stating the hypothesis, the concept of *micro-bitstream* has to be introduced which is used in the hypothesis.

Definition 1 *A **micro-bitstream** is a building block of configuration data—logic or routing or both—that can be used to compose bitstream of a larger function or design.*

A micro-bitstream may exist as a partial bitstream, as a collection of bitstream frames, or as a collection of sparse frame vectors and may represent a single logical setting, a single routing segment or a mix of both. Regardless of its representation or granularity, a micro-bitstream contains the information necessary to configure some portion of a circuit. The prefix *micro*

stresses that micro-bitstream corresponds to a few settings or small a functional block and not a full design.

Hypothesis 1 *A valid bitstream of arbitrary complexity can be composed by offsetting and logically OR-ing a suitable set of micro-bitstreams.*

The function of a bitstream is to configure resources in an FPGA so that circuit of intended design gets implemented on it. Suppose there exist several micro-bitstreams, each individually configuring a set of resources on FPGA, and these micro-bitstreams are logically OR-ed together into a full bitstream. The set of resources configured by the resultant bitstream will be union of all the sets of resources configured by input micro-bitstreams. Mathematically, the hypothesis can be expressed by the equations given below.

$$
\begin{aligned}
&If\ B = M_1 \| M_2 \| M_3 \| \ldots \| M_k \\
&Then\ R_B = R_{M_1} \cup R_{M_2} \cup R_{M_3} \cup \ldots \cup R_{M_k} \\
&Where, \\
&\qquad B\ is\ full\ bitstream \\
&\qquad M_i\ is\ i_{th}\ micro-bitstream \\
&\qquad R_B\ is\ set\ of\ resources\ configured\ by\ B \\
&\qquad R_{M_i}\ is\ set\ of\ resources\ configured\ by\ micro-bitstream\ M_i
\end{aligned}
$$

If the micro-bitstreams are chosen suitably, the resultant bitstream can represent a valid design. And if a set of suitable micro-bitstreams is created, a full bitstream for an arbitrary digital function can be created by OR-ing a subset of the 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 resources on (or configure them). This was tested non-rigorously in hardware with a simple design: a number of settings were removed from one XDL design and inserted into another XDL design and bitstreams

were generated for both. Neither bitstream worked correctly by itself, but when the two bitstreams were merged by logically OR-ing their frame data, the resultant bitstream had original functionality. The main assumption behind this hypothesis is that configuring a resource leads to some *set bits* in the bitstream, and so logically *OR-ing* bitstreams results in union of resource configurations of all the input bitstreams. Though this hypothesis was tested only on a Xilinx FPGA (`xc5vfx130t`), it may apply to FPGAs from other vendors too.

From the hypothesis it can be deduced that if a number of bitstreams are OR-ed together, the resultant bitstream will contain configuration of all the input bitstreams. Figure 4.1 illustrates the concept with two bitstreams. The deduction can be extended to say that bitstream of a full design can be created by disassembling the design into smaller components, creating a bitstream for each component, and merging bitstreams of the smaller components. Going a step further, a set of Turing-complete primitives can be defined, and a library containing micro-bitstreams of the primitives can be set up. Now, a design can be mapped to the defined primitives, and bitstream for the design can be created by fetching micro-bitstreams of the composing primitives from library and merging them.

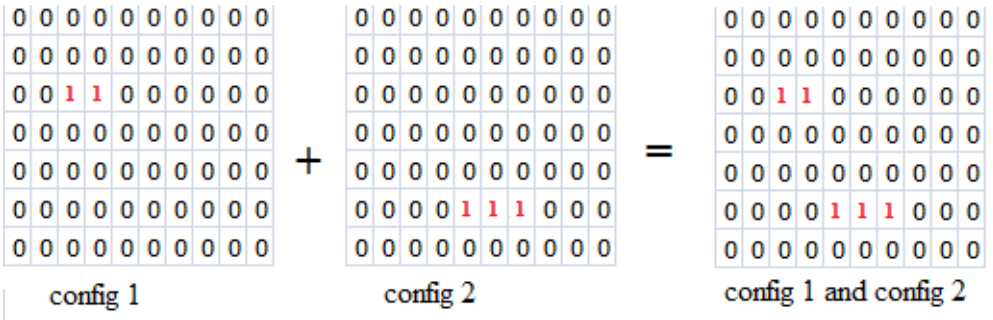


Figure 4.1: Merging bitstreams. Bitstream 1 has configuration 1, bitstream 2 has configuration 2. The resultant ORed bitstream has both configuration 1 and configuration 2.

Creating a library of micro-bitstreams and generating bitstream by merging micro-bitstreams is core idea of this work.

4.2 General Approach

This section gives an overview of the generic process of selecting primitives, creating library of micro-bitstreams, and generating bitstream by merging the micro-bitstreams. Although the steps described here are generic, they are influenced by the work on Xilinx devices.

All the steps involved in the bitstream generation process are shown in Figure 4.2. 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 are coarse-grained. In the general case, a custom device database must be created based on the defined primitive set, and a special mapper, placer, and router must be developed to target the virtual device. This work uses architectural primitives and so the steps in blue are not discussed.

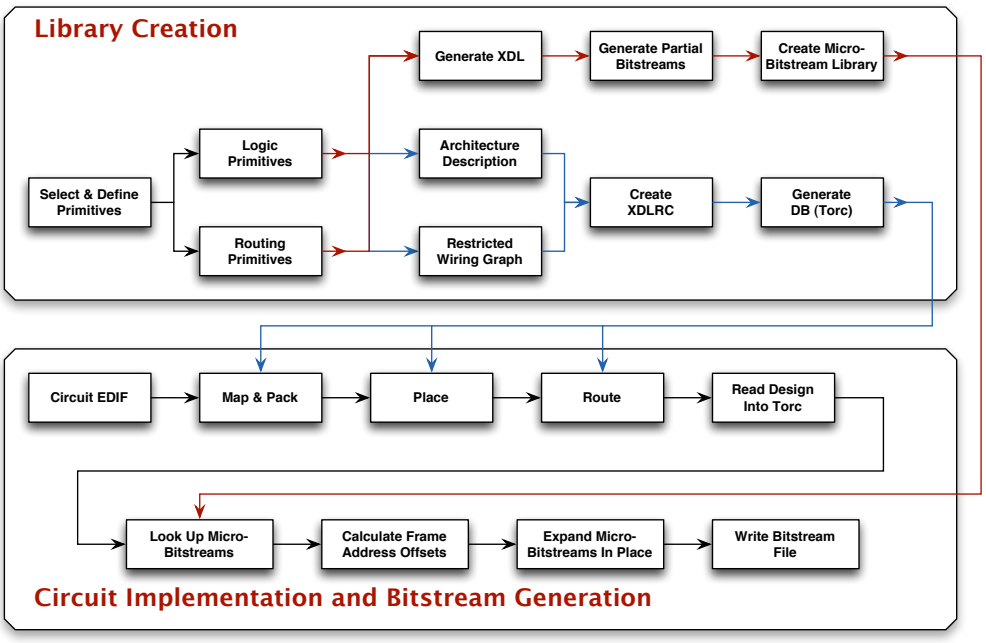


Figure 4.2: General bitstream generation flow

4.2.1 Primitive Selection

The first step in the whole process is primitive selection. The set of primitives should be Turing-complete, i.e. a suitable subset of primitives can be assembled to implement any arbitrary digital function. The other points to consider for primitive selection are granularity—coarse-grained or fine-grained—and feasibility of generating micro-bitstreams using vendor tools. Figure 4.3 shows an example of 2-bit counter implemented with coarse-grained primitives, while Figure 4.4 shows the same circuit implemented with fine-grained primitives.

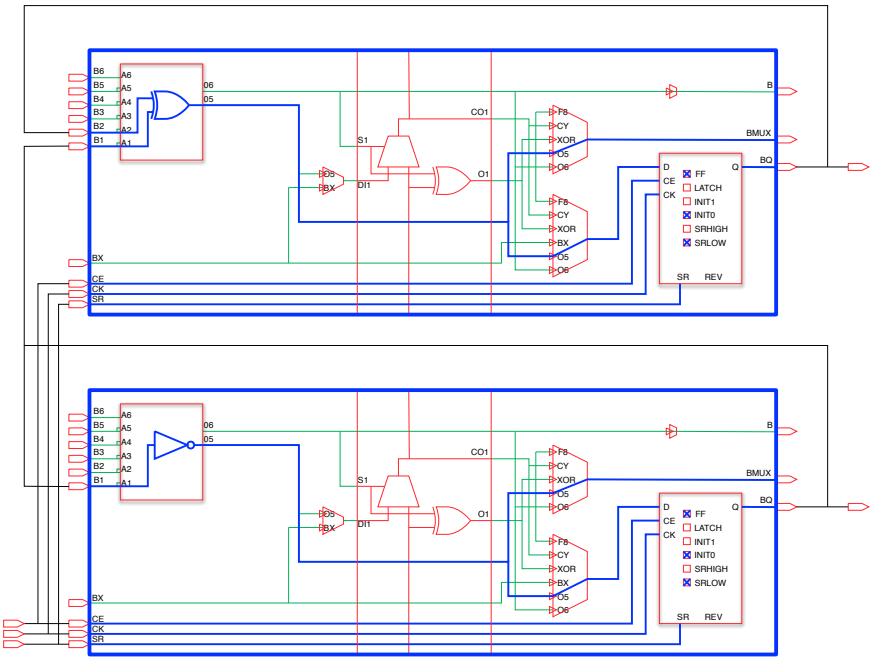


Figure 4.3: Example 2-bit counter built out of coarse-grained blocks. The upper block consists of a preconfigured XOR gate with synchronous and asynchronous outputs. The lower block consists of a preconfigured inverter with synchronous and asynchronous outputs. Both of these would be preconfigured primitives 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.

Coarse-grained primitives can be basic functional blocks like 1-bit adder, 4-bit multiplier, 2-to-1 multiplexer, counter etc. A coarse-grained primitive can also be formed by combining some logic resources along with routing PIPs, which may not form a meaningful functional

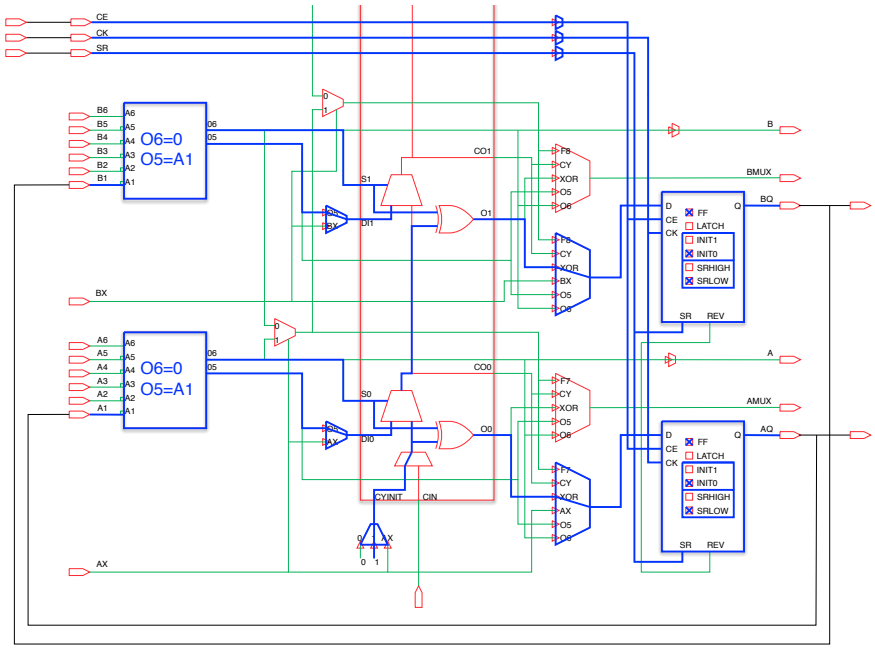


Figure 4.4: Example 2-bit counter built out of fine-grained blocks. The blocks are *architectural primitives*. Arbitrarily complex LUT masks are composed from input passthrough functions according to the equations given. The circuit looks as complicated as the coarse-grained version for this very simple 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.

block by itself but can be used to create a larger design as shown in Figure 4.3. In case of coarse-grained primitives, the design must first be mapped to the primitive set, and subsequently placed and routed within the custom architecture that the primitives define.

Fine-grained primitives map to FPGA architectural primitives. For example a PIP configured in a routing multiplexer or a logic resource setting can be considered fine-grained primitives. Some of the primitives present in Figure 4.4 are—passthrough LUTs, storage elements configured in flip-flop mode, and some multiplexer configurations. Fine-grained primitives are more flexible, make efficient use of underlying architecture, and do not require custom mapping, placing, or routing, so the path shown with blue arrows in Figure 4.2 becomes unnecessary. But generating micro-bitstreams for fine-grained primitives might not

be straight forward. This work uses fine-grained primitives and some of the problems faced while generating micro-bitstreams for fine-grained primitives are discussed in Section 5.2.

4.2.2 Library Creation

After deciding on primitives, micro-bitstreams have to be created for the primitives and stored in a library.

First, the primitives have to be expressed in a format from which a bitstream can be generated. Hardware Description Languages (HDLs), such as Verilog and VHDL, can be used for primitives that represent functional blocks. The non-functional compound primitives and fine-grained primitives have to be expressed in lower level format such as EDIF, BLIF, etc. Some vendors release their custom format to represent mapped design, which can also be used to represent such primitives. XDL, discussed in Section 2.3, is a format from Xilinx to represent mapped and optionally placed and routed design and is used in this work to represent primitives.

Next, primitives have to be generated in the chosen format, mostly in form of files, and then mapped, placed, and routed. Special constraint files can be used to restrict FPGA resources used by the primitives. Xilinx provides User Constraint Files (UCF) for this purpose. In case custom format to represent mapped design is used, these steps might not be necessary. Finally, a bitstream has to be generated for the placed and routed design. Either vendor tools or any custom tools can be used to map, place, and route, but vendor tools *have* to be used for generating a bitstream. Instead of full bitstreams, partial bitstreams [1] can be generated to reduce library size. Since the primitives correspond to a small number of

[1] Partial bitstream contains only a few frames to configure a small portion of FPGA

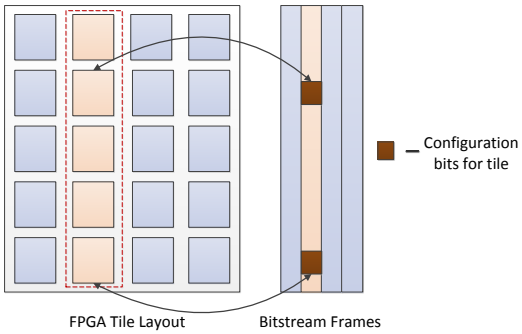


Figure 4.5: Shifting configuration bits

resource settings, even the partial bitstreams will have few set bits and so can be highly compressed. A bitstream generated for a primitive is called *micro-bitstream*.

It is very likely that a primitive can be mapped to multiple locations on an FPGA because of its regular architecture and repetition of resources. FPGAs have multiple instances of a single resource type laid out in a regular fashion. For example in Xilinx device `xc5vlx110t` there are $8,640$ instances of resources in a CLB tile arranged as a matrix of size 160 x 54 [8]. At this scale of resource repetition, a primitive can be mapped to large number of locations on the FPGA. If micro-bitstreams are created for every location of primitive, the library size will become very large.

Regularity of the resource layout on an FPGA can be explored to reduce the library size. As the FPGA architecture is very regular, it can be assumed that the pattern of configuration bits for a particular setting of a resource type will remain same irrespective of the location of the resource. Also, the configuration data for one location can be shifted in the bitstream to map to similar resource in another location. Figure 4.5 shows the idea of shifting of configuration bits. To calculate the amount of shift, knowledge of the tile layout on the FPGA and mapping from tile location to bitstream frame is required. With this knowledge it will be sufficient to generate micro-bitstream for primitives in one location only. During

bitstream merging, the micro-bitstream can be offset to match the location of the primitive in the input design.

4.2.3 Bitstream Generation

The basic idea of bitstream generation is to first map an input design to defined primitives, traverse the primitives, fetch micro-bitstreams for each primitive from library and merge them with a base bitstream as shown in Figure 4.6. The base bitstream can be empty or have some static design. If coarse-grained primitives are used, there is need for special device databases and special mapping, placement, and routing tools. Figure 4.2, placed at the beginning of this section, shows bitstream generation process for coarse-grained primitives. If the primitives selected are architectural primitives, then the mapping part will not be required.

If the library stores micro-bitstreams for primitives on only one location, offset calculation of configuration data will be required. As stated earlier, this offset calculation requires knowledge of the tile layout on the FPGA and mapping from tile location to bitstream frame. Figure 4.5 shows the idea of shifting of configuration bits.

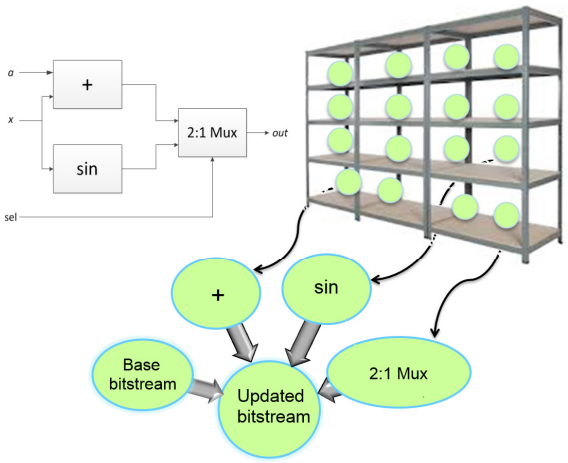


Figure 4.6: Micro-bitstream merging. The circuit on top-left contains three primitives—an adder, a sine function, and a 2:1 multiplexer. A bitstream for the circuit can be generated by fetching micro-bitstreams of each of the three primitives from library and merging them, optionally with a base bitstream.

Chapter 5

Implementation Details

This chapter gives details of all the steps of bitstream generation—primitive selection, library generation, and bitstream merging— as implemented in this work. As this work uses architectural primitives the steps of creating custom architecture, mapping, placing, and routing are not required. Figure 5.1 gives an overview of the whole process.

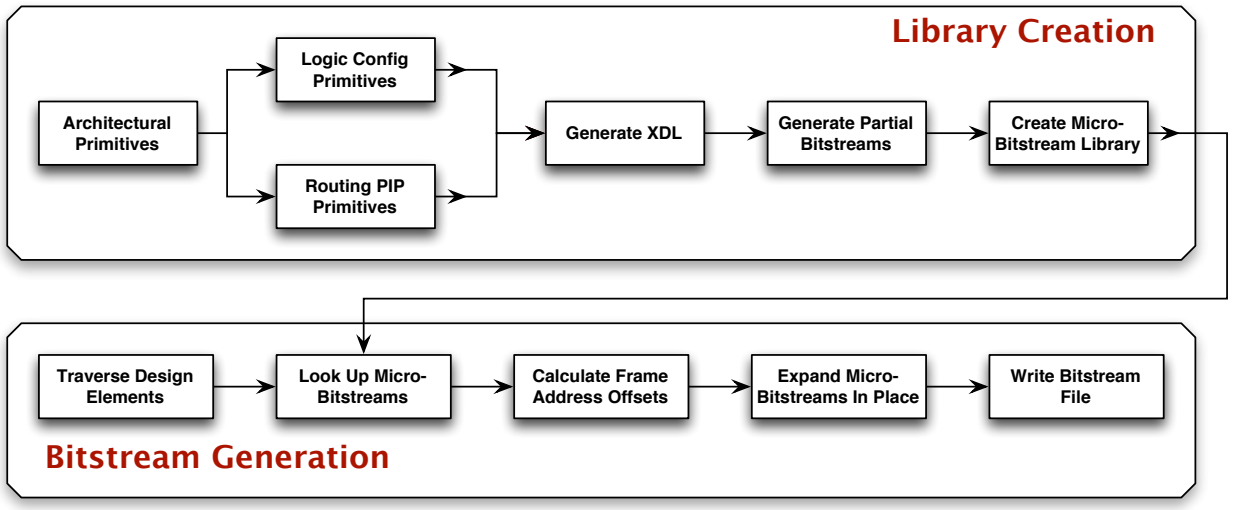


Figure 5.1: Simplified bitstream generation flow. The top half shows library creation process and the bottom half shows bitstream merging process.

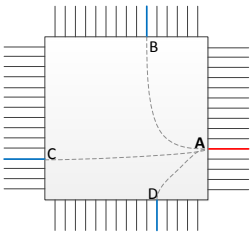
5.1 Primitive Selection

For this work primitives of finest granularity were selected, such that they map to the architectural primitives. The primitives will be discussed in details in this section. As the routing resources and logic resources have different characteristics, the two will be discussed separately.

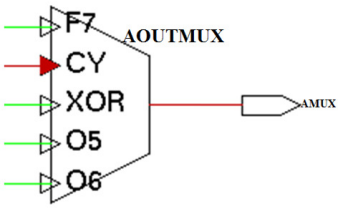
5.1.1 Routing Primitives

In a routing tile, a configured PIP connects a sink wire to a source wire of the tile. Figure 5.2a shows a routing tile where the red wire on left is a sink wire and it can be connected to three source wires. There is a separate PIP to control each of the three connections. Usually, multiple PIPs across multiple routing tiles have to be configured to route a net. In an XDL design, nets are bounded by a source and one or more sink pins, and are composed of routing PIPs. Only the routing PIPs affect the bitstream.

For routing resources, every PIP is considered a primitive. The INT and CLB routing tiles are supported in this work.



(a) The sink wire on right can be connected to three source wires.



(b) A multiplexer resource in logic site SLICEL.

Figure 5.2: Routing and logic primitives

5.1.2 Logic Primitives

Logic sites have variety of resources and configuration options. Each configuration option of a resource is considered a primitive. For example, the logic site SLICEL contains a configurable multiplexer called AOUTMUX. This multiplexer can be configured to pass any of the five input signals as shown in Figure 5.2b. Each of the five possible configurations of the resource AOUTMUX is a primitive. Site types supported for logic primitives are—SLICEL, SLICEM, RAMB, and DSP48E.

Selection of primitives was influenced by multiple criteria, including usage in typical designs and absence of peculiar configurations: 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. BRAM and DSP primitives were similarly selected because of their prevalence.

5.1.3 Unsupported Resources

Some resources are configured with complex or arbitrary strings, while others are configured by complex code. For example, the Virtex-5 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.

5.2 Library Creation

Micro-bitstream library creation is a multi-step process. To begin with, an XDL design is created for each primitive. The XDL designs are converted to NCD format and then to corresponding bitstreams using Xilinx tools. Each bitstream is compared to a reference bitstream and commonalities are discarded, so that what remains is the effect of the XDL primitive. The resulting micro-bitstreams are compressed and stitched together into a library. Each step is discussed in details in the following subsections.

5.2.1 XDL Generation

As stated before, an XDL design is created for every primitive. For the logic primitives, a reference XDL design is also created for every resource with the resource turned off. Torc provides APIs to read/write XDL files and also iterate over tile types, the sites and routing PIPs within a tile, the resources in a site, and the possible settings of a resource.

Before generating XDL designs for primitives, the target device and the tile location for primitives have to be decided. Generating a library for just one device of a FPGA family works for all other devices of the family as long as the supported resources are present in the chosen device. This is possible because the height of a clock region in an FPGA, and consequently height of a bitstream frame, is the same for all the devices of a family, and the micro-bitstreams are stored only for the first tile of a column in a clock region. For a given FPGA family, the smallest possible device should be chosen for generating a library as the process of creating NCDs and bitstreams is faster for a smaller device. For the Virtex-5 family device `xc5vlx110t` was used and for the Virtex-7 family `xc7a110t` was used.

The chosen set of primitives can be mapped to multiple locations on an FPGA, but micro-

bitstreams are generated for only one location. The tile location of micro-bitstreams should be such that the offset calculation becomes simple during bitstream merging. In this work, micro-bitstreams correspond to first tile in a tile column within a clock region. Location of the tile for primitives can be arbitrary as after generating a bitstream for the primitives, configuration data can be shifted during micro-bitstream compression to make it correspond to the first tile.

The process of XDL generation is different for routing tiles and logic sites and each will be discussed separately.

Routing Primitives

An XDL design generated for a routing primitive contains a dummy instance with no configuration and a fake net with just one PIP. Listing 5.1 shows an example of an XDL design for a routing primitive. Even though this XDL design is invalid, the Xilinx tools provide options to convert such a design to a bitstream.

```
1 design "Virtex-5-INT-routing-EN2END1-IMUX_B15" xc5vfx130tFF1738-2 v3.2;
2 inst "dummyInst" "SLICEL", placed DUMMY SLICE_X1Y1;
3 net "dummyNet",
4     outpin "dummyInst" BQ,
5     inpin "dummyInst" B4,
6     pip INT_X1Y199 EN2END1 -> IMUX_B15 ;
```

Listing 5.1: XDL for a routing primitive

Torc APIs were used to cover the supported routing tile types, and iterate through all the PIPs available in the routing tile. For each tile type a tile location has to be provided in which the PIP will be configured. Torc provides a list of all tiles present on a device and the first tile of each tile type was selected for placement. To get all the PIPs in a routing tile, first the source wires are iterated over, and for each source wire all its sinks are visited.

Each source-sink pair corresponds to a PIP. Below is the pseudo code for generating XDLs for routing tiles.

```
For all supported routing tile types
  Get first tile of the type
  For every source wire in tile
    Get all sinks of the source wire
    For each sink
      Generate XDL with source-sink PIP.
```

Listing 5.2: Pseudo code for genrating XDLs for routing primitives

Logic Settings

XDL design files are generated corresponding to each setting of every configurable resource present in the supported logic sites. So for resource `AOUTMUX` present in site SLICE, shown in Figure 5.2b, five XDL files are generated, one each for the configurations `F7, CY, XOR, O5,` and `O6`.

Xilinx tools treat logic settings differently from routing PIPs. Even though a single PIP in a design does not make sense, bitstream generated for such a design contains configuration bits corresponding to the PIP. But in case of logic settings, if a resource setting's associated resources are not instantiated in the design, Xilinx tools don't generate configuration bits for the resource. In other words, if a resource setting doesn't make sense, Xilinx tools ignore it. This condition was interpreted from the finding that for some resources, micro-bitstream was same for all the settings. In some cases, micro-bitstreams were empty for all the settings. The resource `AFFMUX` present in site SLICEL is a good example to show the problem of dependencies. This configurable multiplexer drives the input of flip-flop `AFF`,

shown in Figure 5.3, but if AFF is not instantiated, then setting of AFFMUX has no effect upon the bitstream regardless of its setting.

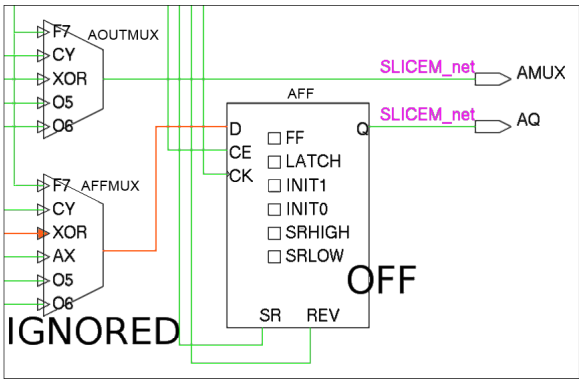


Figure 5.3: Logic resource dependency. AFFMUX setting gets ignored if AFF is not configured.

To properly generate micro-bitstreams for resources with dependencies, XDL design must include the dependency. Some logic resources depend upon other logic resources, while some depend upon the presence of a net driving or being driven by the resource. To counter these situations, the concept of a *harness* design was used which is described next.

Logic Site Harnesses

The harness XDL design contains a 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, all of the primary site resources—all LUTs and flip-flops in the case of slices— are instantiated, 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. One harness design was created for every supported logic site type.

Harness design creation is a manual process. Resources of a site have to be analyzed to find how they are inter-connected and which resources might have dependencies. The rule

of thumb is to configure all resources in a site. Setting of some resources have to be fine tuned to make sure the dependent resource's output is used and also make sure Xilinx tools generate bitstream for the harness. The warning messages generated by the Xilinx *xdl* tool while converting an XDL file to a NCD file is a good way to find out if any resource might be ignored by *bitgen*. The harness for SLICEL of Virtex-5 family is given in Appendix A.

The harness XDL serves as the basis for generating XDLs for the logic configurations. For every resource, an XDL is generated with the resource turned off which serves as "reference XDL" for all the settings of the resource. Then XDLs are generated for every configuration of the resource. The rest of the configurations of the harness remains in-tact. Below is the pseudo code for generating XDLs for logic settings.

```
1 For all supported logic site types
2   Get harness for the site type
3   For every resource in the site
4     Generate XDL with resource set off
5     For every configuration of the resource
6       Generate XDL with resource configured
```

Listing 5.3: Pseudo code for generating XDLs for logic primitives

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. Valid operators are '~' (NOT), '+' (OR), '*' (AND), and '@' (XOR). Virtex-5 slices have four fracturable LUTs—named A, B, C, and D—that can generate separate functions of five and six inputs.

An example of LUT setting in XDL is `D6LUT::#LUT:O6=~(A1*A2)`, also shown in Figure

5.4. Here, the D-LUT is configured in LUT mode, as opposed to RAM or ROM mode, to implement the Boolean equation `~(A1*A2)` assigned to output `O6`.

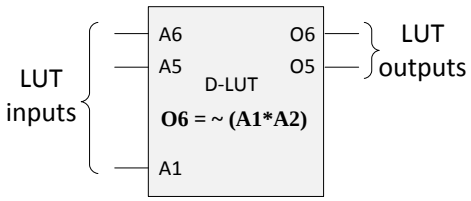


Figure 5.4: A D-LUT in site SLICEL

There is no feasible way to generate micro-bitstreams for all possible Boolean equations with the LUT inputs and given operators. 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 run-time by applying the equation in bit-wise fashion to these micro-bitstreams. The same is done for output `O5`. XDLs generated for LUTs do not use harness design as incompatible equations for the two outputs—`O5`, and `O6`— result in faulty micro-bitstreams. Xilinx tools generate configuration bits for LUTs even without the harness.

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. A 6-input LUT has 64 memory bits and can therefore take on 2^{64} values. Instead of generating a micro-bitstream for each of possible value of LUT memory, the Logic Allocation (LL) file, that *bitgen* creates when given the '`-l`' (ell) flag, was used. 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
...";
```

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.

Compound Resources and Exceptions

There are cases where a group of bits is controlled by more than one logic resource. This is not a dependency, where one resource must be enabled for another one to affect the bitstream, but rather a case where the bitstream values truly are determined by two or more separate resources. For example, in DSP sites, the resources AREG and ACASCREG jointly affect the same set of bits in the bitstream. In such cases, XDL designs are generated for each combination of settings of the related resources. In the above example, resource AREG can take three settings (0, 1, and 2) and resource ACASCREG also takes the same three settings. A total of nine XDL designs are generated for the compound resource composed of AREG and ACASCREG, though Xilinx User Guide [11] mentions some restrictions on what combinations are legal.

No formal method was devised in this work to detect compound resources. Such resources were detected only by accident and the specification given in Xilinx User Guides [11]. To

exhaustively test all combinations of logic resource settings would be intractable. Compound resources that are supported behave like super-primitives.

Another exception occurs when resources that are configured off, set bits in the bitstream. For example, when a DSP site is instantiated, resources `LFSR_EN_SET` and `TEST_SET_P` both set bits in the bitstream, even when turned off. This is problematic because the approach of generating micro-bitstreams presupposes that unused resources have no impact upon the bitstream. To deal with such resources, concept of reference setting was used—a setting of the resource that does not set bits in the bitstream. While generating XDL designs for logic resources, a reference XDL design is also created in which the resource is turned off. But for resources like `LFSR_EN_SET` and `TEST_SET_P`, the reference XDL design had the reference configuration of resource which does not set bits in the bitstream.

5.2.2 Micro-bitstream Generation

All the XDL files generated in the previous step are converted to NCD files using the Xilinx *xdl* tool and the NCD files are then converted to bitstreams using the Xilinx tool *bitgen*. The bitstreams corresponding to resource configuration are compared to respective reference bitstream and the their diff is compressed and saved. Since the XDL designs of primitives are invalid, option *-force* has to be used with the *xdl* tool to force generation of NCD. For the same reason option *-w* has to be used with *bitgen* to disable Design Rule Checker (DRC).

The reference bitstream varies with the resource type. For routing resources, i.e. PIPs, reference bitstream is an empty bitstream. A reference bitstream for a logic resource is created from the reference XDL which has the resource turned off, or set to reference configuration that does not produce configuration bits. A bitstream corresponding to a resource configu-

ration is termed as “primary bitstream”. Figure 5.5 shows the primary bitstreams and the reference bitstream for resource AOUTMUX.

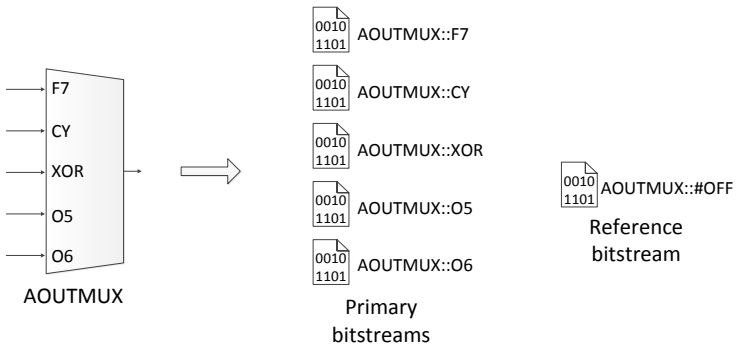


Figure 5.5: Primary bitstreams and reference bitstream for resource AOUTMUX

Comparison of bitstreams is done bit-by-bit and the difference is stored frame wise. If a frame in the primary bitstream has set bits not present in reference bitstream, then the frame is stored with only those set bits. The frames are identified by an index normalized to the first frame in the current column, and the column index is not stored in the library. That means data stored in library are independent of tile column index. The set of frames obtained after diffing should only contain configuration bits corresponding to the resource configuration.

The resultant micro-bitstream (or set of frames) consists of sparse binary data that is highly compressible, typically by two or more orders of magnitude. As such, the frames are compressed before storing in library. Every bit that was set in the primary bitstream but not in the reference bitstream is represented as a very compact <frame index, word index, bit index> tuple in a 32-bit word. Here, frame index refers to the relative index of a frame within a column. Word index is relative to frame, and bit index is index of bit within the word.

A frame contains configuration data for all the tiles in a column within the corresponding

clock region. Depending on the location of a tile within a column, the configuration data for the tile shifts within the frame, but the bit pattern remains the same. It was assumed that this shift of configuration bits is proportional to the tile position within the column, which was verified to be true later. So, in the tuple of <frame index, word index, bit index> word index and bit index depend on the tile location within the column. Configuration data is stored for only the first tile in a column and during bitstream merging the word index and bit index are offset as required.

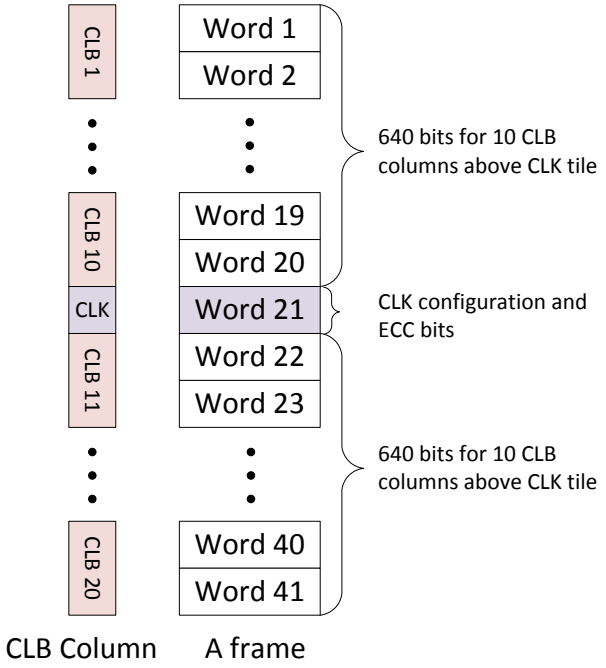


Figure 5.6: Mapping of frame bits to a CLB column

For example, in the case of Virtex-5 FPGAs, a bitstream frame consists of 41 32-bit words out of which the middle word is for ECC and CLK tile configuration [12]. In a column of CLB tiles there are 20 CLB tiles in one clock region. Assuming that rest of the 40 words of a frame are evenly distributed over 20 CLB tiles gives two words (or 64 bits) per CLB tile—see Figure 5.6. This assumption can be extended to deduce that shifting a CLB tile by one row in a column will shift configuration data by two words in the respective frame.

Similar calculation can be done for DSP and BRAM columns also. This logical assumption was tested to be true after implementation.

Micro-bitstreams stored in library correspond to the first tile of a column in a clock region.

For LUTs in RAM/ROM mode and BRAM's memory and parity initial value configuration, the tuple information (<frame index, word index, bit index>) is directly obtained from the LL file, which is generated by *bitgen* when '-l' options is provided. The LL files were generated for the first tile of column so that bit shifting is not required when using data from the LL file.

5.2.3 Library Organization

The compressed micro-bitstreams are stitched together into a single file for convenience and speed. Internal structure of the library file is as follows:

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

In the above file structure, micro-bitstream data is an array of the <frame index, word index, bit index> tuples.

5.3 Bitstream Generation

Bitstream generation process was shown in Figure 5.1 at the start of this chapter. Since this process essentially merges micro-bitstreams together, the tool created in this work is called *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 is 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.

5.3.1 Design Traversal

Bitmerge uses Torc APIs to traverse the XDL design, first visiting all the instances and then all the nets. In every instance all the resource settings are traversed, and in every net all the PIPs are traversed. Every placed instance gets a tile location, so all the resource settings in the placed instance belong to the same tile location. Whereas, in a net each PIP gets its own tile location. The frame set and offset calculation is done for every instance and every PIP.

5.3.2 Resource Processing

Processing is simple for most of the resources—read the configuration, look up corresponding compressed micro-bitstream in library and merge with base bitstream. For some resources, a few extra steps have to be taken. Processing of different types of resources is described below.

Routing PIPs

Processing of PIPs is straightforward—fetch the micro-bitstream from the library based on parent tile type, source, and sink.

Logic Settings

Processing of most logic settings is also straightforward—fetch the micro-bitstream from the library based on parent tile type, site type, the resource, its setting. Site type is present in instance declaration and tile type can be obtained from the tile location.

There are some special cases in logic resources.

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: `A6LUT::#LUT:O6=A1`, `A6LUT::#LUT:O6=A2`, `A6LUT::#LUT:O6=1`, and so on. The desired function is formed by applying the Boolean expression to the appropriate micro-bitstreams. This concept is better understood with an example.

Consider the LUT setting `A6LUT::#LUT:O6=(A1*A2)@(A3+A4)`. Here output `O6` of A LUT is assigned to Boolean equation `(A1*A2)@(A3+A4)`. The library does not contain micro-bitstream for this equation, but it contains micro-bitstreams for each input variable, i.e. A1, A2, A3, and A4. The micro-bitstream for the equation is formed by fetching the micro-bitstream of each input and performing the Boolean operations on the micro-bitstreams as

present in the equation. Mirco-bitstreams are first expanded to a frame set to do the Boolean operation at the frame level. This can be expressed in form of an equation.

$$M_{(A1*A2)@(A3+A4)} = (M_{A1} * M_{A2}) @ (M_{A3} + M_{A4})$$
$$Where,$$
$$M_X \; is \; micro-bitstream \; of \; variable \; X$$

To generate the configuration bits for a LUT equation, *bitmerge* parses and evaluates the expression with code generated by *Bison* and *Flex*. When the code encounters a variable or literal in the equation, it fetches the corresponding micro-bitstream from the library, expands it to 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. The unary NOT operator cannot be implemented directly, because *bitmerge* does not know which frame bits to invert. It instead implements inversion by XOR-ing the current set of stack frames with the micro-bitstream for logic 1. XOR-ing is applied in bitwise fashion, and the result is pushed back onto the stack. When the parsing completes, the only set of frames remaining on the stack is merged with the base frame set.

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`. Some examples of resources which have hex string as configuration setting are—LUTs

in RAM/ROM mode, BRAM's data and parity initialization settings, and DSP's `MASK` and `PATTERN` resources.

Special Case: Compound Primitives

As explained in Section 5.2.1, there are a few cases of compound primitives, where two or more resources jointly affect a set of frame bits. When one of the resources of compound primitive is encountered, other resource of the primitive are also gathered and the corresponding micro-bitstream is fetched based upon the combined settings of the individual resources.

5.3.3 Frame Address And Offset Calculation

Micro-bitstream of a primitive fetched from library contains tuples of the form <frame index, word index, bit index>. Here, frame index is relative to the first frame of a column in a bitstream, word index relative to the first word in the frame, and bit index within the word. From the XDL design the tile index of the primitive can also be identified. With the information of the tile index and micro-bitstream tuples, the location in a bitstream has to be identified where the micro-bitstream should be merged.

Given a tile index, Torc can return set of all the frames present in the corresponding bitstream column. This frame set is same for all the tiles in a column within a clock region. Frame index from a micro-bitstream tuple can be directly used in the frame set to identify the frame which should be modified by the tuple. The word index and bit index of tuples correspond to the first tile of the tile column in clock region and have to be offset as per tile location in design. The offset calculation was explained earlier in Section 5.2.2.

5.3.4 Bitstream Merging

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

An option to clear bits in bitstream, as opposed to set bits in bitstream, is also provided in the bitstream generation API. This option has been provided keeping the fault tolerance application in mind. A typical scenario of a fault tolerance application would be to unplace instances from faulty sites, unroute all nets connected to the unplaced instances, and then place and route the design. In such a scenario, instead of generating full bitstream for the new design, the bits for the resources which were removed from design can be cleared out, and then bits for the newly introduced resources can be set.

5.4 Code Structure And Usage

A high level discussion of the code structure and class hierarchy will be helpful if anyone wants to extend this work. The code is written in C++ and will eventually become part of the Torc library.

5.4.1 Bitstream Merging Code

Class hierarchy of bitstream merging code is shown in Figure 5.7. The base class `Assembler` is an abstract class but most of the implementation goes into that class. The derived

class `VirtexAssembler` is also abstract and serves to contain some data types, like the Frame word type, common to all Virtex families. The leaf classes—`Virtex5Assembler` and `Virtex7Assembler`— contain family specific attributes and implementation. The code to parse LUT equation was created by *Bison* and *Flex* and is encapsulated in a separate class named `LutParser`. A factory class, not shown in the figure, was also created to facilitate easy creation of family specific class objects.

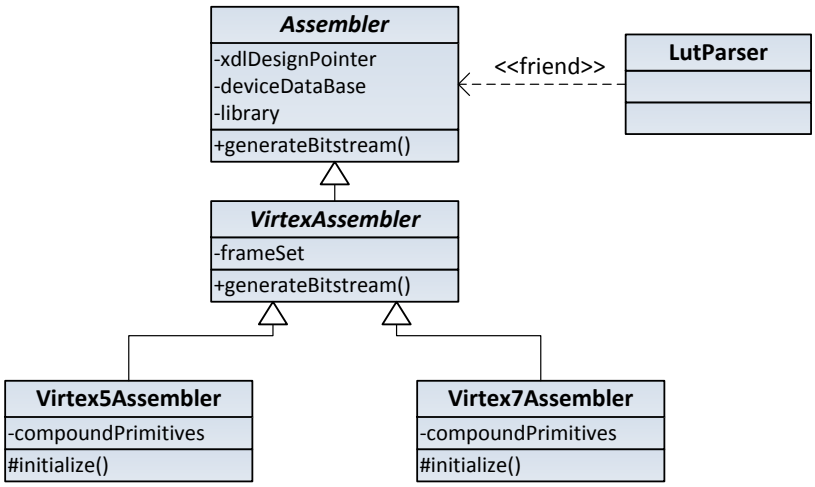


Figure 5.7: Class hierarchy for bitmerge

The base class implements tasks which should be same across all families. Some of the tasks implemented in base class include traversing XDL design, iterating over the resource settings in an instance and PIPs in net, parsing LUT equations and hex strings, and loading the library. Classes for families contain attributes that are expected to change with the family and also code for object initialization and bitstream manipulation.

When this code gets added to Torc, micro-bitstream libraries for Virtex-5 and Virtex-7 will also be added. So, anyone who wants to use this open-source bitstream generation API for Virtex-5 or Virtex-7 families will primarily use the bitstream merging API.

5.4.2 Library Generation Code

The class hierarchy of library generation code is shown in Figure 5.8. As in case of bitstream merging classes, the base class `AssemblerLibGen` here too is an abstract class and implements most of the tasks. Some of the tasks implemented in the base class are—generating XDL files for logic and routing primitives, converting XDLs to bitstreams, and stitching micro-bitstreams to one file.

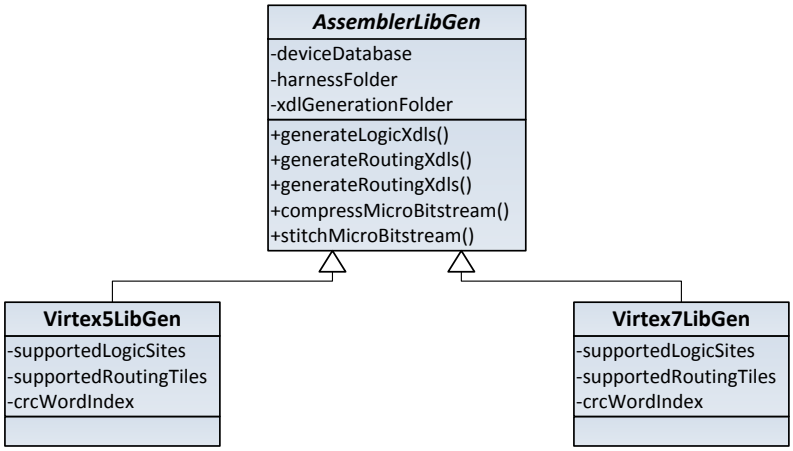


Figure 5.8: Class hierarchy for library generation

The derived classes have family specific attributes such as list of supported logic site types, list of supported routing tile types, list of compound resources and a map of reference setting for the logic resources. They also implement the functions related to bitstream reading/writing.

Chapter 6

Results and Analysis

This chapter presents evaluation of *bitmerge* in terms of resource coverage, range of supported devices, fidelity, portability, extensibility, run-time performance, and library size. A comparison of *bitmerge* with previous works on some of these points is provided in Table 6.1.

6.1 Resource Coverage

In Virtex-5 family, *bitmerge* supports logic sites 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 and CLB tiles. These routing tiles cover the majority of the routing resources in any device. In Virtex-7 family, only SLICEM and SLICEL are supported among logic sites and for routing, INT and CLB tiles are supported. An attempt was made to support the BRAM sites, but the bitstreams generated by *bitmerge* for BRAM sites in Simple Dual Port (SDP) mode do not match with *bitgen's* bitstreams. Support for the DSP site was not tried because of time constraint.

	Bitmerge	Bit2NCD	JBits	Partial Bitstreams [1]
Resource Coverage	*Logic*: CLB, BRAMs, and DSP. *Routing*: INT and CLB	*Logic*: CLB, BRAMs, and DSP. *Routing*: INT and CLB	*Logic*: CLB, BRAMs, and DSP. *Routing*: INT and CLB	Module dependent
Supported Devices	All Virtex-5 and Virtex-7 family devices.	Spartan-3, Spartan-3E, Virtex-II, Virtex-4, Virtex-5 series	Xilinx XC4000 and Virtex family.	Virtex-II Pro
Method	Merging micro-bitstreams	Reverse-engineering	Bitstream format available	Bitstream module relocation
Granularity	Fine	Fine	Fine	Coarse
Embeddable	Yes	N/A	Yes	Yes

Table 6.1: Comparison with previous works.

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 vendor privileged data. A simple solution is to configure any unsupported resources in a base bitstream that is then imported by *bitmerge*.

6.2 Validation and Fidelity

The micro-bitstream library was validated by creating test designs in XDL, generating bitstreams for those designs, and comparing the bitstreams to the corresponding *bitgen* output. A tool was created to do word-by-word comparison of the configuration data of two input bitstreams. Torc does not currently calculate and update frame ECC values, so "clock" word at the center of each frame is ignored when comparing bitstreams.

Individual XDL designs were created for every supported logic site and for a sample of routing PIPs. Some XDL instances for logic sites and PIPs for routing tiles were taken from real designs, while some were custom created to try different resource settings. For the smaller designs (containing 2-3 instances and 15-20 PIPs), *bitmerge's* bitstreams matched *bitgen's*. A design of Reed-Solomon encoder, taken from OpenCores [28], was also tested for validation and *bitmerge's* output did not match *bitgen's* for this design. The design contained 345 instances and 823 nets after removing instances and PIPs of unsupported tile types. While it can be deduced that *bitmerge* does not generate configuration data correctly for all resource configuration, it is difficult to quantify fidelity in terms of a percentage of the supported resources. This is because the mismatched configuration bits cannot be mapped back to FPGA resources, and so it is difficult to identify which resource setting resulted in wrong configuration data.

6.3 Extensibility

This work initially supported the Virtex-5 family devices; later support for the Virtex-7 family devices was added. Extending support to other architectures will require the creation of a suitable harnesses for every logic site and addition of family specific classes for both library generation and bitstream merging. Adding new classes will need little effort as most of the implementation is done in the base class. Creating a harness design will require substantial work as inter-dependency of the resources of a site have to be analyzed and some trials might be required to find the useful combination of resource configuration. Supporting routing PIPs will not require any extra effort.

Harness designs of one family of FPGAs cannot be used for another family as resources in a site might differ. For example, the logic site SLICEL in the Virtex-7 family contains more

resources than its Virtex-5 family counterpart; there are four extra flip-flops and the output of the flip-flops is tied to inputs of multiplexers within the site. A particular setting of the multiplexers ensures that the setting of these extra flip-flops is not ignored by the Xilinx tools. The harness for the logic site BRAM also differs for the two families. For Virtex-5 family, the harness for BRAM contains an instance of BRAM with no configuration, but such an harness did not work for the Virtex-7 family.

Extending support to a new family is dependent on the Torc library and Xilinx tools also. Torc should have a database for the family architecture, support reading/writing of a bitstream of the family, and be able to map from the FPGA tile index to the bitstream column. Support for the Virtex7 family was added only after these features were added to Torc for the family. Xilinx tools are used during the creation of a micro-bitstream library. The next version of the Xilinx tools might be stricter with invalid designs. For example, the tools might not generate a bitstream for a design with just one PIP in it even with the option to ignore Design Rule Check (DRC). If such situations arise, it will be a good idea to use coarse-grained primitives.

6.4 Portability

Implementation of *bitmerge* is dependent on, and later will be a part of the open source library Torc, which uses the C++ standard library, and the open source libraries *Boost* [29] and *zlib* [30]. As the source code of all the dependencies of *bitmerge* is available, it can be theoretically compiled for any platform. The tool *bitmerge* has been compiled and run on several desktop machines with different flavors of Linux and also on a Linux based embedded platform. The embedded platform was built on XUPV5 development board [7] which has a Virtex-5 family FPGA—`xc5vlx110t`. The MicroBlaze Soft Processor [31]

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

Table 6.2: *Bitmerge* performance results.

core was instantiated on the FPGA and Linux kernel and *bitmerge* were cross-compiled for MicroBlaze using Xilinx tool chain.

6.5 Runtime Performance

Runtime performance was tested 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 `xc5vfx130t` and includes 20,518 configurable instances with 405,431 logic settings, and 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 6.2 shows the runtime performance of *bitmerge*. License agreement terms preclude benchmarking *bitmerge* against Xilinx software, but is was noted that *bitmerge* compares quite favorably, particularly when the data originates in XDL form.

6.6 Library Size

The micro-bitstream library for Virtex-5 family is 547 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. The RAMB initialization and

parity data occupies a little over half of this space. The Virtex-7 library is slightly over 1 MB, which includes the BRAM initialization and parity data but does not include micro-bitstreams for BRAM and DSP sites. The reason for increase in Virtex-7 library size is that the tile columns are pair-wise mirrored and so every tile type has two variants—left and right.

In the future, this data will be compressed using *gzip*, as Torc already does for its device databases.

Chapter 7

Conclusion

An open-source bitstream generator has been described here that does not require reverse-engineering of tools or configuration bitstreams. The motivation originates from two possible applications: 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.

Generating bitstream directly from an XDL design will remove the overhead of converting the XDL design to NCD and also the overhead of *bitgen's* DRC and full bitstream generation, which are involved with the vendor's tool flow. Also, if changes to an XDL design are done using Torc APIs, making bitstream changes from inside Torc will amortize the overhead of program start-up, database initialization, and file I/O.

Making bitstream changes from inside an embedded system will allow us to perform dynamic tuning, or to change the system autonomously while it is running. A stand-alone fault tolerance application can make good use of embedded bitstream generation capability.

The approach taken in this work is to create a library of micro-bitstreams corresponding to architectural primitives, and combine them into arbitrarily complex designs by expanding

and OR-ing their frame contents after relocation. The input is a placed and routed design in XDL format, with an optional base bitstream, and the output is a new or modified bitstream.

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

The micro-bitstream library currently supports Xilinx Virtex-5 and Virtex-7 devices, but the 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 a site harnesses to enable dependent resources.

This work presented a bitstream generation API in C++ which will be a part of the open-source Torc library. This API takes a placed and routed XDL design as input and based on the XDL data, modifies the configuration frames directly. The bitstream generator was also run on an embedded system based on `XUPV5-LX110T` development platform.

7.1 Future Work

The following enhancements and improvements can be tried out to stretch the concept of open-source bitstream generation by merging micro-bitstreams.

1. Applications to demonstrate the usefulness of this work will boost research in field of open-source bitstream generation. A simple fault tolerance application using the bitstream generation capability in an embedded system was planned as a part of this thesis but it is still a work-in-progress. Completing the application will help the cause.

   Also, once this work is released as a part of Torc, other researchers might use the API in their applications.

2. To increase the fidelity of *bitmerge*, harness designs should be created for every resource instead of one common harness for all the resources in a site. For example, the BRAM site in Virtex-7 family has different restrictions for the two modes of operation—True Dual Port (TDP) and Simple Dual Port (SDP)—which results in incorrect bitstreams. Different harness for the two modes of BRAM might help to generate proper micro-bitstreams for its resource configurations.

3. This idea of creating library of micro-bitstreams can be tried with coarse-grained primitives. It will be interesting to observe what granularity of primitives can still implement a wide range of digital functions and if coarser primitives can help in improving fidelity and extending the resource coverage.

Bibliography

[1] 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*, pp. 80–87, 2010.

[2] J. Rose, J. Luu, C. W. Yu, O. Densmore, J. Goeders, A. Somerville, K. B. Kent, P. Jamieson, and J. Anderson, "The vtr project: Architecture and cad for fpgas from verilog to routing," in *Proceedings of the 20th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 77–86, ACM, 2012.

[3] 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*, pp. 45–54, 2007.

[4] 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*, pp. 264–264, 2008.

[5] N. J. Steiner, *Autonomous Computing Systems*. PhD thesis, Virginia Tech, March 2008.

[6] 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`.

[7] Xilinx Inc., "Xilinx University Program XUPV5-LX110T Development System." `http://www.xilinx.com/univ/xupv5-lx110t.htm`, January 2012.

[8] Xilinx Inc., "Virtex-5 FPGA User Guide," March 2012.

[9] Xilinx Inc., "Virtex-4 FPGA User Guide (v2.6)," December 2008.

[10] C. Beckhoff, D. Koch, and J. Torresen, "The xilinx design language (xdl): Tutorial and use cases," in *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2011 6th International Workshop on*, pp. 1–8, 2011.

[11] Xilinx Inc., "Virtex-5 FPGA XtremeDSP Design Considerations," January 2012.

[12] Xilinx Inc., "Virtex-5 FPGA Configuration User Guide," October 2012.

[13] Xilinx Inc., "7 Series FPGAs Configuration," January 2013.

[14] 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.

[15] W. S. Gosset, "Atmel AT40k/94k Configuration Format Documentation." Posted to `comp.arch.fpga` and archived with message-id 20050812150910.29614.qmail@nym.alias.net.

[16] A. Love, W. Zha, and P. Athanas, "In Pursuit of Instant Gratification for FPGA Design," in *Field-Programmable Logic and Application (FPL 2013). International Conference on*, September 2013.

[17] E. L. Horta and J. W. Lockwood, "Automated method to generate bitstream intellectual property cores for Virtex FPGAs," in *Field Programmable Logic and Application* (J. Becker, M. Platzner, and S. Vernalde, eds.), vol. 3203 of *Lecture Notes in Computer Science*, pp. 975–979, Springer Berlin Heidelberg, 2004.

[18] 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*, pp. 117–124, 2011.

[19] M. Hübner, C. Schuck, M. Kiihnle, and J. Becker, "New 2-dimensional partial dynamic reconfiguration techniques for real-time adaptive microelectronic circuits," in *Emerging VLSI Technologies and Architectures, 2006. IEEE Computer Society Annual Symposium on*, vol. 00, pp. 6 pp.–, 2006.

[20] M. Hübner, T. Becker, and J. Becker, "Real-time lut-based network topologies for dynamic and partial fpga self-reconfiguration," in *Integrated Circuits and Systems Design, 2004. SBCCI 2004. 17th Symposium on*, pp. 28–32, 2004.

[21] D. Koch and J. Teich, "Platform-independent methodology for partial reconfiguration," in *Proceedings of the 1st conference on Computing frontiers*, pp. 398–403, ACM, 2004.

[22] 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*, pp. 513–516, 2007.

[23] K. Kepa, 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*, pp. 195–200, 2011.

[24] R. Soni, N. Steiner, and M. French, "Open-source bitstream generation," in *Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on*, pp. 105–112, 2013.

[25] F. Benz, A. Seffrin, and S. Huss, "Bil: A tool-chain for bitstream reverse-engineering," in *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, pp. 735–738, 2012.

[26] Z. Ding, Q. Wu, Y. Zhang, and L. Zhu, "Deriving an NCD file from an FPGA bitstream: Methodology, architecture and evaluation," *Microprocessors and Microsystems*, vol. 37, no. 3, pp. 299 – 312, 2013.

[27] Xilinx Inc., "End User License Agreement." `http://www.xilinx.com/support/documentation/sw_manuals/end-user-license-agreement.txt`, March 2012.

[28] OpenCores. `http://www.opencores.org`.

[29] Boost. `http://www.boost.org`.

[30] zlib. `http://www.zlib.net`.

[31] Xilinx Inc., "MicroBlaze Soft Processor Core." `http://www.xilinx.com/tools/microblaze.htm`.

Appendix A: Harness for site SLICEL of Virtex-5 family

```
1
2  # ==========================================================
3  # This is a harness circuit. The purpose of this XDL is to retain the
4  # individual setting of some muxes which gets removed during XDL to NCD conversion, when
       applied individually.
5  # time: Wed 20th June
6
7  # ==========================================================
8
9  design "harness_slicel" xc5vfx130tff1738-2 v3.2;
10
11 inst "SLICEL" "SLICEL",placed CLBLM_X1Y38 SLICE_X2Y100  ,
12   cfg " A5LUT:SLICEL.A5LUT:#LUT:O5=A1 A6LUT:SLICEL.A6LUT:#LUT:O6=A1
13        ACY0::O5 AFF:SLICEL.AFF:#FF AFFINIT::INIT0
14        AFFMUX::XOR AFFSR::#OFF AUSED::0
15   B5LUT:SLICEL.B5LUT:#LUT:O5=A1 B6LUT:SLICEL.B6LUT:#LUT:O6=A1
16        BCY0::O5 BFF:SLICEL.BFF:#FF BFFINIT::INIT0
17        BFFMUX::XOR BFFSR::#OFF BUSED::0
18   C5LUT:SLICEL.C5LUT:#LUT:O5=A1 C6LUT:SLICEL.C6LUT:#LUT:O6=A1
19        CCY0::O5 CEUSED::#OFF CFF:SLICEL.CFF:#FF
20        CFFINIT::INIT0 CFFMUX::XOR CFFSR::#OFF CLKINV::CLK COUTMUX::#OFF
21        COUTUSED::0 CUSED::0
22   D5LUT:SLICEL.D5LUT:#LUT:O5=A1 D6LUT:SLICEL.D6LUT:#LUT:O6=A1
23        DCY0::O5 DFF:SLICEL.DFF:#FF DFFINIT::INIT0
24        DFFMUX::XOR DFFSR::#OFF
```

```
25    PRECYINIT::0  CARRY4:SLICEL.CARRY4:#OFF  CYINITGND:SLICEL.CYINITGND:#OFF "
26    ;
27
28  net "SLICEL",
29    outpin "SLICEL" A,
30    outpin "SLICEL" AMUX,
31    outpin "SLICEL" AQ,
32    outpin "SLICEL" B,
33    outpin "SLICEL" BMUX,
34    outpin "SLICEL" BQ,
35    outpin "SLICEL" C,
36    outpin "SLICEL" CMUX,
37    outpin "SLICEL" COUT,
38    outpin "SLICEL" CQ,
39    outpin "SLICEL" D,
40    outpin "SLICEL" DMUX,
41    outpin "SLICEL" DQ,
42    inpin "SLICEL" A1,
43    inpin "SLICEL" A2,
44    inpin "SLICEL" A3,
45    inpin "SLICEL" A4,
46    inpin "SLICEL" A5,
47    inpin "SLICEL" A6,
48    inpin "SLICEL" AX,
49    inpin "SLICEL" B1,
50    inpin "SLICEL" B2,
51    inpin "SLICEL" B3,
52    inpin "SLICEL" B4,
53    inpin "SLICEL" B5,
54    inpin "SLICEL" B6,
55    inpin "SLICEL" BX,
56    inpin "SLICEL" C1,
57    inpin "SLICEL" C2,
58    inpin "SLICEL" C3,
59    inpin "SLICEL" C4,
60    inpin "SLICEL" C5,
61    inpin "SLICEL" C6,
62    inpin "SLICEL" CE,
63    inpin "SLICEL" CIN,
```

```
64    inpin "SLICEL" CLK,
65    inpin "SLICEL" CX,
66    inpin "SLICEL" D1,
67    inpin "SLICEL" D2,
68    inpin "SLICEL" D3,
69    inpin "SLICEL" D4,
70    inpin "SLICEL" D5,
71    inpin "SLICEL" D6,
72    inpin "SLICEL" DX,
73    inpin "SLICEL" SR
74    ;
```

Listing 1: Harness for site SLICEL of Virtex-5 family