Module Shaping and Exploration in Rapid FPGA Design and Assembly Workflows

Kevin Lee

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
Paul E. Plassmann
Robert W. McGwier

May 14th, 2015
Blacksburg, Virginia

Keywords: FPGA, Productivity, Rapid Compilation, Modular Workflow
Copyright 2015, Kevin Lee

Module Shaping and Exploration in Rapid FPGA Design and Assembly Workflows

Kevin Lee

(ABSTRACT)

The modular design methodology has been widely adopted to harness the complexity of large FPGA-based systems. As a result, a number of commercial and academic tool flows emerged to support modular design including Hierarchical Design Flow and Partial Reconfiguration Flow, OpenPR, HMFlow, PARBIT, REPLICA, GoAhead and QFlow frameworks. As all of these projects have shown, a modular approach raises the abstraction level, provides clear boundaries for incremental design, reduces placement complexity, and improves productivity. At the physical layer, modules can be compiled into rectangular regions, suitable for placement on the FPGA fabric. Creating a design then becomes the process of placing all of the modules on the FPGA, followed by inter-module routing. FPGAs, however, are not homogenous, and the shape of individual modules could greatly impact overall device utilization. Prior work in modular assembly utilize modules with a single shape and aspect ratio in the assembly process. Due to the increasing size and heterogeneity of contemporary FPGAs, the placement flexibility of such a module is becoming increasingly limited. This thesis introduces a process that exploits offline shape generation and exploration, enabling the selection of shapes using criterias such as resource usage efficiency, placement flexibility, and device utilization. Module shapes can be generated with these criterias in mind while still taking advantage of the reduced placement complexity of modular design and assembly.

Contents

1 Introduction **1**

1.1 Research Contribution 3

1.2 Organization of Thesis 4

2 Background **5**

2.1 Reconfigurable Hardware 6

2.2 Software Model for FPGAs 6

2.3 Hierarchical Design 7

2.4 Existing Workflows 8

2.4.1 HMFlow 9

2.4.2 QFlow 10

2.4.3 TFlow 11

2.4.4 DREAMS 12

2.5 FPGA Tiles and Configuration Bits 13

2.5.1 Artix, Kintex, Vertex, Zynq 14

3 Module Library **16**

3.1 Module Creation 17

3.1.1 Design Entry and Synthesis 18

3.1.2 Shaping and Placements 18

3.1.3 Map, Place, Route, Bitstream 28

3.1.4 Module Registration 29

3.2 Static Creation 32

4 Design Assembly **33**

4.1 Design Entry 33

4.2 Module Relocation and Placement 34

4.3 Routing and Bitstream 35

4.4 Placement Visualization 37

5 Implementation and Results **39**

5.1 Placement and Efficiency Gains 39

5.2 Resource Coverage 44

5.3 GReasy 46

6 Conclusion **50**

6.1 Future Work 51

Bibliography **52**

List of Figures

1.1 Recent Xilinx device resource capacities 2

2.1 HMFlow development workflow [15] C. Lavin, M. Padilla, J. Lamprecht, P. Lundrigan, B. Nelson, B. Hutchings, "HMFlow: Accelerating FPGA Compilation with Hard Macros for Rapid Prototyping," Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on , pp.117,124, 1-3 May 2011. Used under fair use, 2015. 9

2.2 QFlow development workflow [16] T. Frangieh, P. Athanas, A design assembly framework for FPGA back-end acceleration, Microprocess. Microsyst. 2014, http://dx.doi.org/10.1016/j.micpro.2014.02.008. Used under fair use, 2015. . 10

2.3 TFlow development workflow [17] A. Love, W. Zha, P. Athanas, "In pursuit of instant gratification for FPGA design", Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on, pp.1,8, 2-4 Sept. 2013. Used under fair use, 2015. 11

2.4 DREAMS development workflow [18] A. Otero, E. de la Torre, T. Riesgo, Dreams: A Tool for the design of Dynamically Reconfigurable Embedded and Modular Systems, In the Proceedings of the 2012 International Conference on ReConFigurable Computing and FPGA (RECONFIG2012), December 9-11, 2012. Used under fair use, 2015. 13

2.5 CLB, BRAM, and DSP tile distribution across various devices 15

3.1 Floorplanning with a selection of shapes 17

3.2 Selection of shapes with their corresponding placements 19

3.3 PlanAhead script for resource estimation 21

3.4 Resource estimation output from PlanAhead 21

3.5 On-demand module shaping 22

3.6 SQL query optimized for efficiency 25

3.7 SQL query optimized for placements 26

3.8 Console output of sample shape 27

3.9 UCF result for sample shape in Figure 3.8 28

3.10 Module compilation flowchart 30

3.11 Excerpt of placement and shape metadata 31

4.1 Exerpt of design metadata 36

4.2 Visualization of utilized regions on device 38

4.3 Inter-module Connectivity 38

5.1 Six of ten possible offsets for a sub-frame module within a clock frame 41

5.2 Space utilization of two modules 42

5.3 Placements and resource coverage on a Virtex 7 XC7Z045 device 45

5.4 Possible placements in three clock regions for a single module 46

5.5 Interface for GNU Radio 47

5.6 Software defined radio operation 48

List of Tables

5.1 Vertical expansion of clock region size for modern Xilinx FPGA devices. . . . 40

5.2 Shape database generation time and size 44

5.3 Full frame resource efficiency 49

5.4 Sub-frame resource efficiency 49

Chapter 1

Introduction

Ever since the 1980s, Field Programmable Gate Arrays (FPGAs) have seen an increase in popularity for digital hardware design. Known for their power and flexibility, FPGAs are used in a wide range of applications, from ASIC design and testing, to digital signal processing, software defined radio (SDR), computer vision, cryptography, networking, high performance computing (HPC), and even bioinformatics. The innate flexibility stems from the fact that FPGAs are *programmable* hardware devices. Unlike software programs that are compiled down to a set of fixed instructions and executed on a Central Processing Unit (CPU), the underlying hardware within an FPGA is physically configured to implement the desired functionality through the use of Hardware Description Language (HDL) code.

Specifying the functionality of an FPGA device in this fashion however, is non-trivial. In fact, one of the fundamental roadblocks preventing the widespread adoption of FPGA-based heterogeneous computing lies in the low design productivity resulting from digital design and the long compilation cycle [1]. In the software domain, short compilation times and architecture abstractions enable quick evaluation of the resulting code across a wide range of systems, thereby resulting in instant gratification for the developer. In comparison, the digital design turn cycle is prohibitively long, often taking hours to days for large designs.

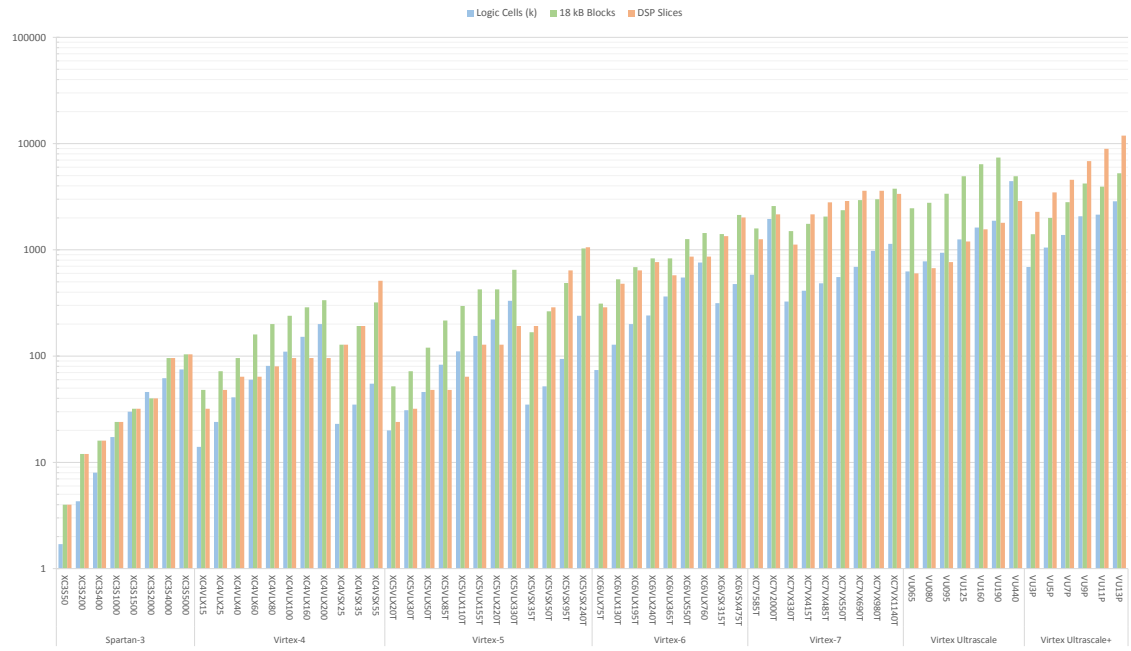


Figure 1.1: Recent Xilinx device resource capacities

Furthermore, unlike programs in the software domain capable of running on any device with the same architecture (x86, ARM, PowerPC, etc..), digital designs are tailored towards a single device with minimal reuse in both new designs and other devices.

The reason behind these digital design limitations is that the synthesis, mapping, placement, and routing at the heart of FPGA compilation is inherently NP-hard. A exponential increase in FPGA device sizes further lengthens the compilation process: FPGA devices have gone from a measly 75K logic cells in the Xilinx Spartan-3 series in 2006 to over 4.4M logic cells in the 2014 Xilinx Virtex UltraScale devices (Figure 1.1). To address the increase in design times, many of the recent developments in FPGA productivity have focused on reducing the level of complexity when building new or incremental designs. Mirroring some of the concepts in the software domain, a popular approach has been to increase the abstraction level during the development process. Instead of compiling the design from scratch, workflows have been proposed that split the design into modular portions that are compiled independently and rapidly assembled at a later stage.

By utilizing this *modular design* methodology, the abstraction level is elevated through the use of components much larger than FPGA primitives. As designs become modular, information from previous compilations can be reutilized to rapidly assemble a new design without having to go through the time-consuming physical compilation process. While the implementation behind modular design workflows have been extensively explored [10][11][13][14][15][17][18], there remains an interesting problem in optimizing the modules resulting from this increased abstraction level. In such workflows, components of a design are separated into *modules* that encapsulate a specific functionality. These modules are then confined into a rectangular region on the target device and can be stitched together to assemble a working design. As devices become larger and less homogeneous however, optimization of such modules becomes increasingly important if the modular design methodology is to be widely adopted.

1.1 Research Contribution

This thesis presents work done to optimize the parameters behind the modules utilized in modular design workflows. Additions were made to TFlow (section 2.4.3), a modular FPGA design and rapid assembly workflow aimed at reducing design compilation times to below the level of which the user's attention is lost [2]. Improvements were made to extend the capabilities of TFlow modules with the use of multiple footprints. These footprints, which are referred to as *shapes*, provide additional flexibility for module selection during the assembly phase of a design. This flexibility manifests itself as a number of quality metrics that can be used to optimize the shape selection criteria; multiple shapes for a FPGA module allows for reduced resource fragmentation when sparse resources such as DSPs and BRAMs are utilized, thereby increasing resource usage efficiency. Placement flexibility is also increased on devices containing heterogeneous resource layouts, resulting in an higher relocatability of a module and, therefore, greater overall device utilization.

1.2 Organization of Thesis

The organization of the thesis is as follows. Chapter 2 provides some background on modular design and methodology, as well as a discussion of related work that utilizes some form of modular design within their workflows. A comparison with the software development cycle is given, with a focus on pre-compilation and rapid assembly. Chapter 3 elaborates on the methodology and limiting constraints in the offline module generation process. Chapter 4 presents the implementation details behind the online rapid assembly phase of a design. The improvements made to the designer interface is given here as well. Chapter 5 presents the results of this work and shows improvements resulting from the consideration of multiple shapes. Finally, Chapter 6 concludes this thesis and presents options for future work.

Chapter 2

Background

Module optimization is an important step in any applications where a set of parameters affect how the final design is put together. Take the classic game of Tetris as an example: if the rotation of shapes were to be prohibited, of the standard seven shapes, only the square and the straight pieces would be able to fill up the entire board without leaving any gaps. If there are an odd number of columns, only a single shape, the straight piece, would be able to achieve this feat. Only by allowing for the use of multiple shapes along with the ability to rotate such shapes, can a board of arbitrary size be completely filled.

This chapter covers the background topics related to this thesis. First, an overview of reconfigurable hardware is presented followed by a description of the general development workflow for FPGAs. The following section elaborates on recent developments aimed at increasing FPGA design productivity through the use of modular design workflows. By breaking the design into modular components, the time required for design assembly can be significantly reduced to allow for near-realtime feedback to the designer. Finally, the underlying fabric making up FPGAs is explored.

2.1 Reconfigurable Hardware

FPGAs occupy a fairly unique segment in hardware development. As hardware devices that are programmable via software, the strength of FPGAs lies in their ability to adapt to various applications without having to change the hardware platform. This configurability is possible due to the use of hardware description languages (HDL) such as Verilog or VHDL. The functionality to be implemented on an FPGA is specified by a designer as HDL and is converted to the low level primitives that make up the underlying fabric of an FPGA through a number of intermediate steps. These steps consists of synthesis, translation, mapping, placement and routing, and bitstream generation. The bitstream file generated in the last step is then used to configure the FPGA with the desired functionality.

2.2 Software Model for FPGAs

A fundamental issue with FPGAs development lies in the time consuming process of turning an HDL design into a bitstream file. Even with older generation devices, larger designs can take anywhere from hours to days to generate a bitstream from HDL. This lengthy turnaround time imposes a heavy penalty on the design process, making FPGAs even less accessible than it already is. Software compilation on the other hand, is orders of magnitude faster, e.g. in applications using interpreted languages like JavaScript, Python, Perl, PHP and other JIT compilers (OpenCL/PyPy), code compilation occurs at run-time in the microseconds to seconds range. Even with more traditional languages such as C or C++, compilation times remain in the seconds to minutes range if precompiled libraries are utilized.

To draw an analogy to the software compilation process, the traditional compilation process for FPGAs would be the equivalent of compiling the designer's program along with all required libraries down to and including the kernel itself. Needless to say, compilation times would be astronomical compared to the modern compilation process that utilizes pre-

compiled libraries. To achieve a similar compilation time, many of the recent developments in FPGA productivity have focused on reducing the level of complexity when building new or incremental designs. A popular strategy has been to increase the abstraction level during the development process. Rather than compiling the design from scratch, workflows have been developed that break the design into modular portions that can be rapidly compiled independently of the rest of the design.

The underlying architecture of digital design facilitates the same division of functionality that is reflected in the use of software libraries. In the same way that software libraries often use a common Application Program Interface (API), each individual module encapsulates a specific functionality and contains a well-defined interface to allow for independent development and testing. As such, the implementation effort is reduced and the same module can be reused in other designs without recompilation. Like software libraries, modules built in such a way are independent entities that can also be easily replaced as long as its interface remains the same. With the right workflow, a number of these precompiled modules can be linked together to create the final bitstream in a minimal amount of time.

2.3 Hierarchical Design

FPGA vendors themselves have implemented the concept of modularizing designs in their tools. Using the Hierarchical Design (HD) flow, complex designs can be partitioned into smaller pieces with defined interfaces that provide insulation boundaries between individual modules [3]. HD flow preserves module placement and routing, and supports an arbitrary module layout that can be composed using a number of adjacent or overlapping rectangles of varying sizes. HD flow modules, dubbed as *design partitions*, are not without limitations. For example, certain primitives including I/O, configuration and clock resources, are allowed only in the top level and cannot be part of a partition.

Xilinx Partial Reconfiguration (PR) flow also uses HD [4]. In the PR-enabled HD flow, the device is first segmented into a number of partition regions. These regions can then be individually reprogrammed with modules of varying functionality, even when the other regions on the device are still running. The downside to using PR however, is that each partition region can be populated with only a single fixed-size, fixed I/O module at a time. The design can include more than one PR-region, but the combination of multiple modules within a single PR partition is not supported. In the PR flow, certain resources that share the clock region with the PR partition cannot be used by the static part of the design or by another PR region.

Another example of the module-based design methodology is the Isolation Design Flow (IDF) used in High Assurance systems [5][6][7]. IDF aims to increase design fault-tolerance and verification by separating the design modules and connecting them only through trusted interfaces. IDF uses a concept of fences [8][9] to support modular redundancy and segregation by safety level. Similar to the PR flow, IDF uses HD partitions and leaves the design and placement of the modules to the designer.

A major drawback to the HD flow and its derivatives is that there is no support for module relocation. In these flows, a module must be compiled at each of the partition locations before they can be used, even if the compiled modules themselves are logically equivalent. If compared to the software domain, it would be like having to recompile each library at every point in the program that the library is used. Needless to say, this would incur a significant overhead in the compilation process.

2.4 Existing Workflows

As an alternative to vendor provided tools, a number of alternative workflows have been developed to reduce the amount of time required in the design compilation process. While

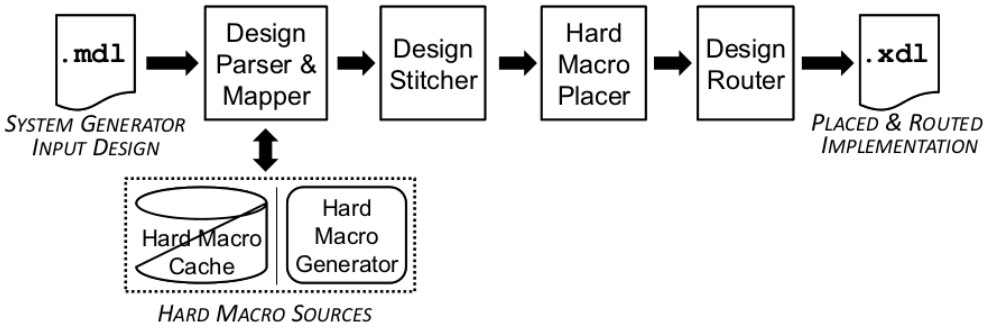


Figure 2.1: HMFlow development workflow [15] C. Lavin, M. Padilla, J. Lamprecht, P. Lundrigan, B. Nelson, B. Hutchings, "HMFlow: Accelerating FPGA Compilation with Hard Macros for Rapid Prototyping," Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on , pp.117,124, 1-3 May 2011. Used under fair use, 2015.

FPGA vendor tools have traditionally focused on achieving the highest quality designs at the expense of long implementation times [1][12], alternative workflows have focused on improving designer productivity by shrinking implementation times through design reuse. The design reuse paradigm often exploits module relocation in order to relax the NP-hard placement problem through the use of coarse-grain placement at the module level.

2.4.1 HMFlow

Hard Macro design flow (HMFlow) uses pre-compiled modules (Hard Macros) in order to accelerate the FPGA implementation time [15]. A hard macro is a pre-placed and pre-routed design that can be positioned in multiple locations within the FPGA fabric. As the design is assembled from hard macros, HMFlow shortens implementation times by skipping design synthesis, mapping, and packing for each module. The placement for these hard macro modules are course grained, and modules can be relocated to any location on the FPGA fabric where the underlying resource pattern is identical. The modules themselves are compiled

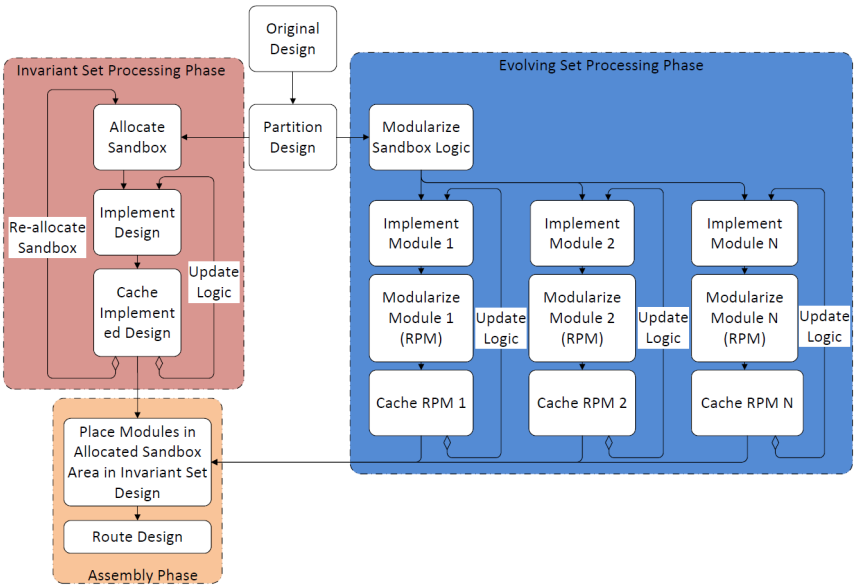


Figure 2.2: QFlow development workflow [16] T. Frangieh, P. Athanas, A design assembly framework for FPGA back-end acceleration, Microprocess. Microsyst. 2014, http://dx.doi.org/10.1016/j.micpro.2014.02.008. Used under fair use, 2015.

using conventional Xilinx tools, and the final design assembly process utilizes a combination of custom as well as vendor provided tools to create the final bitstream.

2.4.2 QFlow

QFlow, short for Quick Flow, is an island-based design assembly framework meant to improve designer productivity [16]. The design paradigm behind QFlow divides a design into two entities: the invariant section (*static*) that is implemented only once and the evolving section (*sandbox*) containing portions of the design that are often reimplemented by the designer. The static portion of the design contains components that will largely remain unchanged in-between designs, such as I/O or memory interfaces. The sandbox region on the other hand contains the modules that are frequently updated by the designer over the course of the design. The development process for QFlow involves four phases: partitioning the design

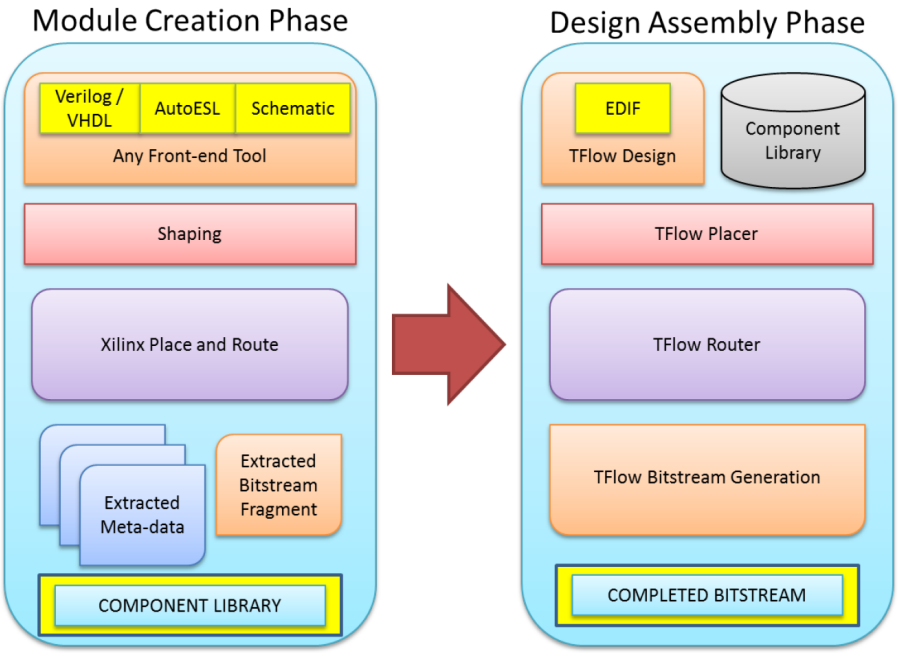


Figure 2.3: TFlow development workflow [17] A. Love, W. Zha, P. Athanas, "In pursuit of instant gratification for FPGA design", Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on, pp.1,8, 2-4 Sept. 2013. Used under fair use, 2015.

into the invariant and evolving sections, building the invariant region, building modules for the evolving region, and assembly of both regions into the final design. Modules to be placed in the sandbox region are independently developed and compiled through the use of conventional Xilinx tools. These modules are then stored in a library for later use. During the assembly phase, each module is placed on the device followed by inter-module routing using the Xilinx router.

2.4.3 TFlow

TFlow, short for Turbo Flow, is a rapid design assembly workflow with the goal of minimizing turnaround time when generating the final bitstream for a design [2][17]. Similar to QFlow, TFlow splits a design into two independent segments for development: the invariant (*static*)

and dynamic (*sandbox*) regions. TFlow however, extends the QFlow paradigm to rapid bitstream-level design assembly aimed at maximizing designer productivity. TFlow is built ontop of TORC, an open source API for manipulation of Electronic Data Interchange Format (EDIF) and Xilinx Design Language (XDL) netlists and Xilinx bitstream packets [19]. In the same manner as QFlow, modules for TFlow are compiled in advance and stored in a module library for later use. During the design assembly phase, rapid turnaround times are achieved by stitching together the precompiled bitstreams from both the static and modules in the module library. Using this workflow, designs can be assembled well within the limit of a person's attention span.

In the first phase of TFlow, modules are built and tested using the standard Xilinx workflow. Once the functionality of the module meets the designer's requirements, a number of unique *implementation images* are built for each module, with each image having a unique set of shapes and placement locations on the target FPGA device. A set of metadata information is extracted for each image as well, and is used later on to speed up the assembly phase.

In the second phase of TFlow, a design composed of a number of previously compiled modules are stitched together into a bitstream in a matter of seconds. Any combination of modules can be used in the design, as long as there are enough resources on the device to contain the modules. No additional overhead is incurred when assembling a design using precompiled modules, but if a module needs to be updated, that particular module needs to be recompiled and reregistred into the module library. With TFlow, turnaround times are minimized through the reuse of modular components, thereby increasing developer productivity.

2.4.4 DREAMS

The design flow for Dynamically Reconfigurable Embedded Platforms for Networked Context-Aware Multimedia Systems (DREAMS) aims to reduce human intervention during the design process in a post-processing stage [18]. Similar to QFlow, DREAMS flow supports the inde-

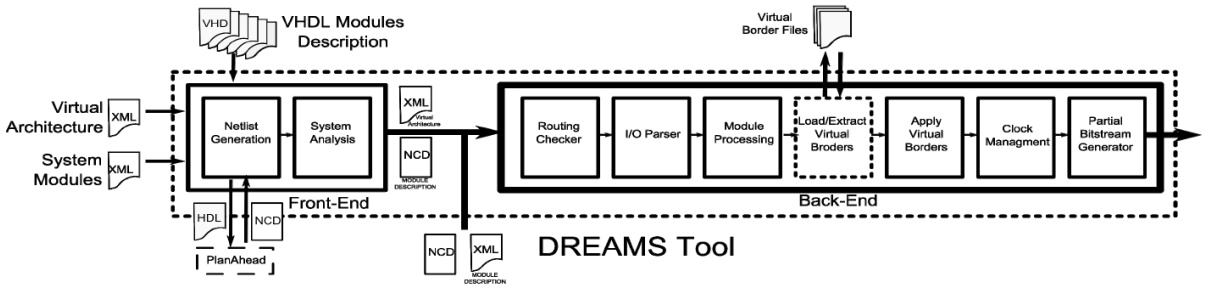


Figure 2.4: DREAMS development workflow [18] A. Otero, E. de la Torre, T. Riesgo, Dreams: A Tool for the design of Dynamically Reconfigurable Embedded and Modular Systems, In the Proceedings of the 2012 International Conference on ReConFigurable Computing and FPGA (RECONFIG2012), December 9-11, 2012. Used under fair use, 2015.

pendent design of both the modules as well as the static system, allowing post-deployment system upgrades and design reuse through interchangeable modules. DREAMS provides module relocation within the device and also across different reconfigurable devices. Modules are built with a specific interface at the boundaries to allow for inter-connectivity. As long as the interfaces are compatible and the modules are properly aligned, inter-device relocation is allowed into any reconfigurable region where the underlying resource layout is identical. Modules must be built with these interfaces in mind, otherwise recompilation of both the modules and the static will be necessary.

2.5 FPGA Tiles and Configuration Bits

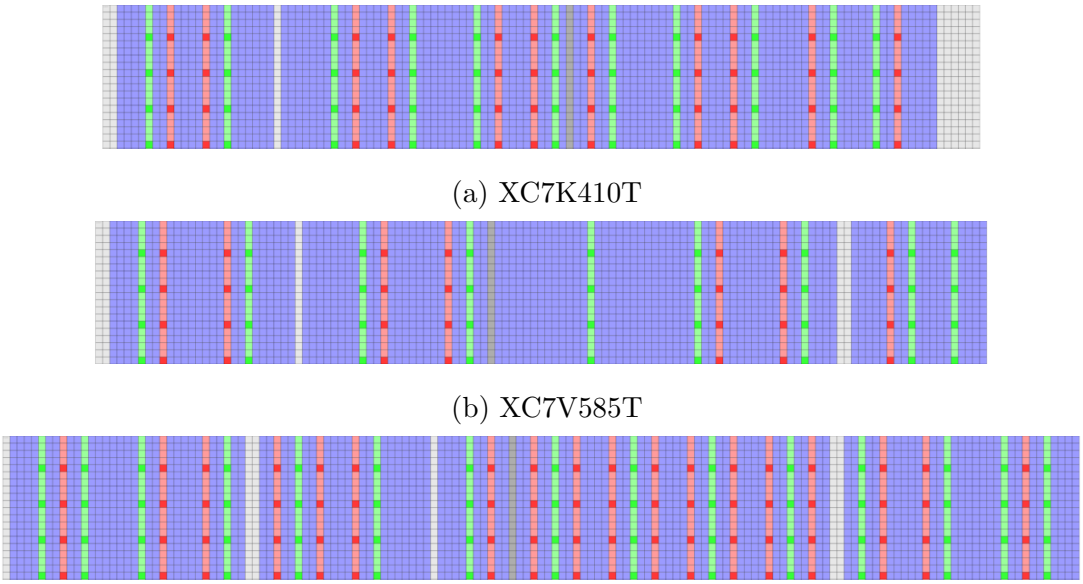
Modules built in the workflows mentioned in the previous sections are relocatable within the device to other locations where the underlying resource pattern remains identical. The internal configuration of an FPGA is specified through a bitstream file that is generated at the end of either the vendor or third-party workflow. This bitstream then contains information on how every component within the FPGA is configured, from the values in the look up tables (LUTs) to the internal routes that interconnect components. While the direct association

of a specific component to the bitstream file is proprietary, the general structure of the bitstream is not. As such, if the general structure of the bitstream is known, modules that are constrained to a portion of the device can be represented as a pattern of bits. By offsetting this pattern of bits by a specific amount, the module can be relocated to other places on the device.

Bitstreams for Xilinx devices are split into frames that are each a *clock region* in height. Each clock region is served by a horizontal spine that contains the clock routing resources used by the components in that region. The height of the clock region varies across devices as well, ranging from 16 rows on the Spartan 6 series devices to 50 rows on the Series 7 devices. Most vendor modular tools use these clock regions to delineate placement boundaries for modules since working with fractional frames presents certain difficulties. In addition to clock regions, resources within the FPGA fabric is separated into columns spanning the width of the device. Each column contains a specific resource, ranging from CLB tiles (blue), BRAM tiles (green), DSP tiles (red), and miscellaneous tiles (grey) for other resources. The distribution of these resources however, is nonuniform across the device. Detailed later on, this heterogeneous nature of resource distribution results in limited reuse of a module across a device. Shown in Figure 2.5, the resource layout varies widely across device families, resulting in limited placement locations for precompiled modules.

2.5.1 Artix, Kintex, Vertex, Zynq

To complicate things further, in the most recent devices from Xilinx, a number of new FPGA families were introduced to address the range of system requirements, ranging from low-cost, high-volume applications to high-capacity, high-performance applications. The Artix-7 family is targeted towards low-cost, low-power, small form-factor applications. The Kintex-7 family is optimized for the best price-performance ratio, and the Virtex-7 family aims to provide the highest system performance and capacity. In addition, to traditional



(a) XC7K410T

(b) XC7V585T

(c) XC7VX415T

Figure 2.5: CLB, BRAM, and DSP tile distribution across various devices

FPGA devices, Xilinx has also introduced a new Zynq family that include a dedicated ARM CPU alongside the FPGA logic resources. The addition of the ARM code allows the device to be used as an embedded Linux platform, further increasing the processing capabilities of the device while also allowing the device to reprogram itself on the fly.

Chapter 3

Module Library

The first phase in the TFlow design workflow is to generate the modules to be stored in the module library. The components generated in this phase comprise of the invariant portion of the design (the *static*) as well as the evolving portion(the *sandbox*). The modules within the sandbox region of the design encapsulate a specific functionality of the designer's choosing, and are often updated over the course of the development cycle. Modules generated as part of the sandbox region are stored locally alongside its corresponding metadata. This metadata is later used in the assembly phase of the design workflow.

In this thesis work, improvements were made to the module creation process to allow modules to take on multiple shapes. In previous work, each module assumed a single shape, leading to a limited number of placements on the device. Although this was sufficient for proof-of-concept work, a single shape per module does not suffice in large designs where high module density is desired. To achieve greater device coverage, better resource efficiency, and greater placement flexibility, a larger selection of shapes is required (Figure 3.1).

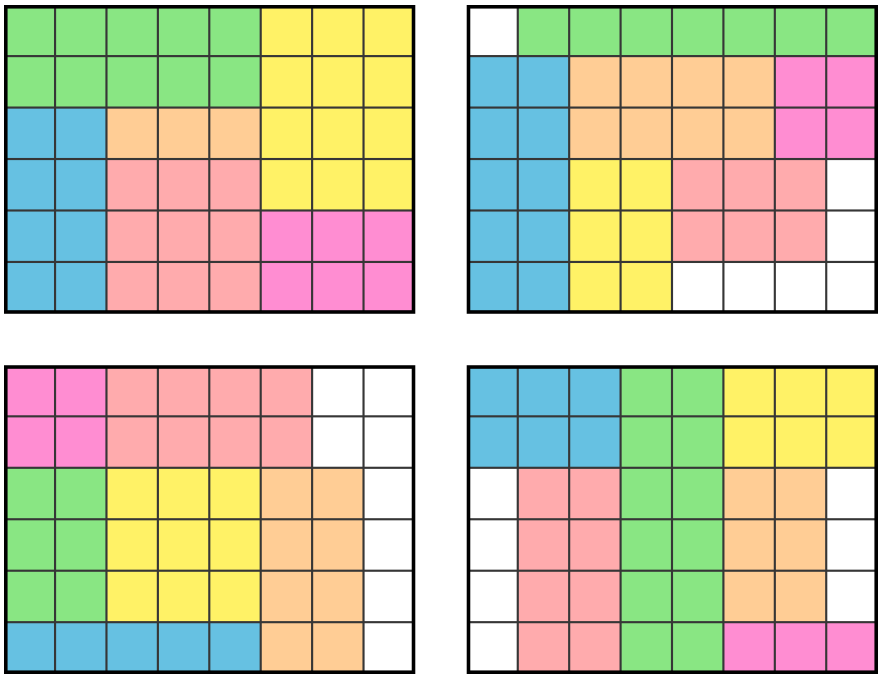


Figure 3.1: Floorplanning with a selection of shapes

3.1 Module Creation

By breaking down the design down into modular components, the TFlow paradigm gives the designer the ability to test and to verify each module before registering it into the TFlow module library. In addition to easier verification, the individual modular components of the design are much smaller in both size and complexity, resulting in orders of magnitudes faster compilation. The module compilation process is inherently parallel as well, as there are no dependencies in-between modules being built. The compilation process utilizes vendor tools for synthesis, map, place, route, and bitstream generation along with TFlow tools for metadata creation. To generate the corresponding metadata for each module, TFlow utilizes the TORC library, an an open-source API that provides the capability to read, write, and otherwise manipulate EDIF and XDL files. TORC also contains the wiring and logic information for major Xilinx devices; The resources provided by TORC are used in both the module generation and assembly phases of TFlow.

3.1.1 Design Entry and Synthesis

The first stage of the module design process is to specify the desired functionality in HDL. Since TFlow has the capability to dynamically route modules during the assembly phase, the only requirement the designer has is to specify a user-defined naming convention for the top-level ports. The port naming convention for the modules is irrelevant for TFlow as the assembly connection graph is also specified by the designer. The only ports that needs to have consistent naming across all modules are the clock lines, which are separately routed by TFlow. The source HDL files are stored in a user-defined location, and the module build process is automated through the use of Makefiles. To compile the modules, the source HDLs are first passed through Xilinx Synthesis Tool (XST). After synthesis is complete, the resulting Xilinx Netlist File (NGC) is converted to an EDIF file for further processing.

3.1.2 Shaping and Placements

Once the module has been synthesized, the constraints of the module needs to be determined before the compilation can proceed. These constraints, termed *shapes*, specify the physical region on the FPGA device that the placer will attempt to constrain the module into. Since these constraints specify the physical placement and location of the module, the resulting shapes can vary widely in terms of size, aspect ratio, resource utilization, and placement locations. Shown in Figure 3.2, a module requiring 200 CLBs and 10 BRAMs has a number of efficient shapes, with each shape having its own set of placements on the device. As the configuration bit pattern within the FPGA fabric is repetitive in the vertical direction and occasionally in the horizontal direction, modules can be relocated elsewhere on the device to locations that have the same underlying configuration bit pattern, thereby keeping the same I/O interface and physical resource footprint. TFlow utilizes this property of FPGAs by creating a module library consisting of each module compiled once for each desired shape and then relocating the internal configuration of the module during the assembly phase.

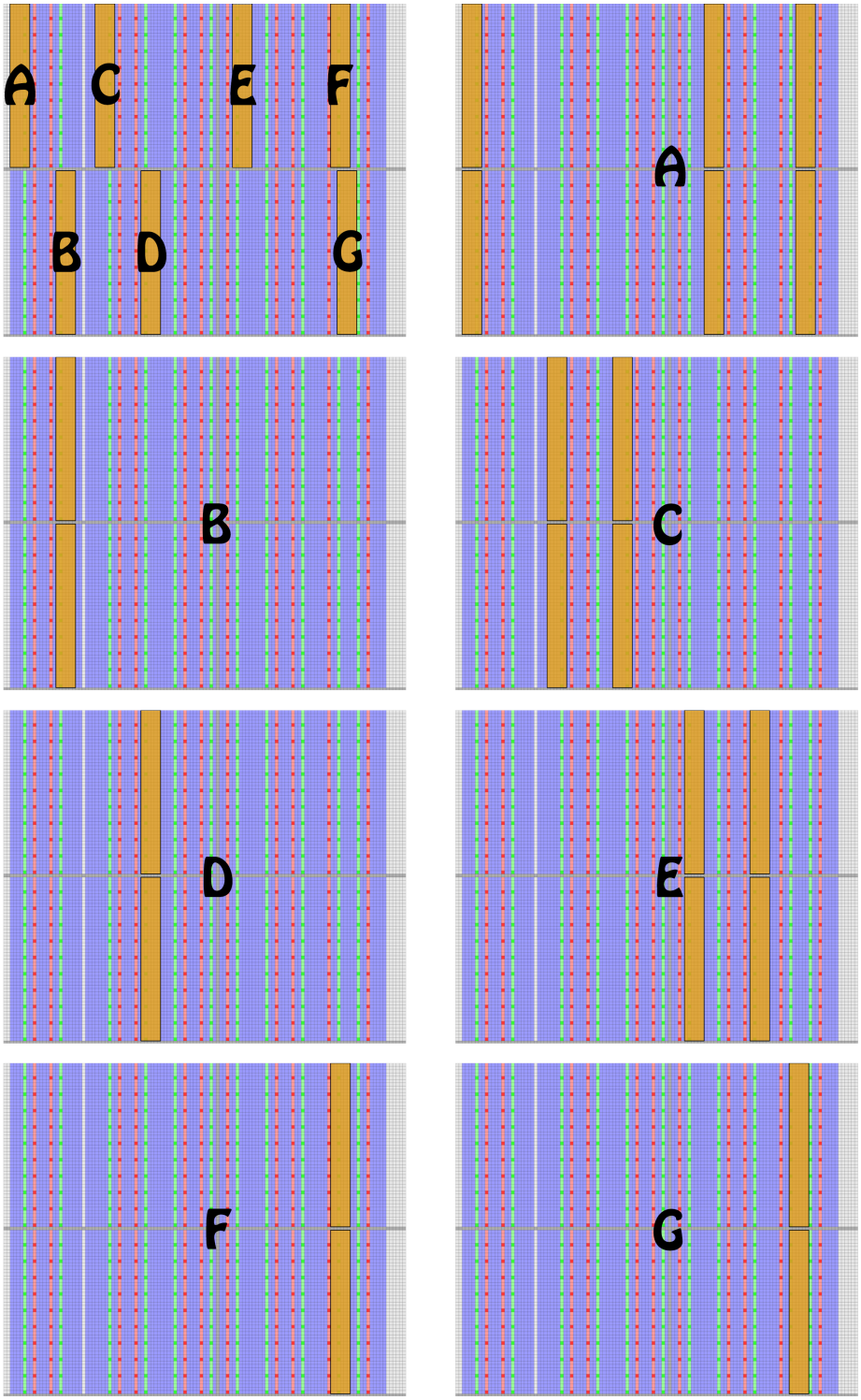


Figure 3.2: Selection of shapes with their corresponding placements

The synthesized design in EDIF format is first passed through Xilinx's PlanAhead utility to obtain an estimation of the resource requirement. This is done using the TCL script shown in Figure 3.3 with sample results from the script shown in Figure 3.4. The output from PlanAhead is then parsed for the estimated quantity of registers (*FLOP_LATCH*), look-up tables (*LUT*), digital signal processors (*DSP/MULT*), and block memory (*BRAM/BMEM*). Additionally, the number of Bus Macros (*BM*) bits used for the input/output ports of the module is retrieved from the EDIF file. The number of required CLBs for the module is computed as a function of the estimated number of registers, LUTs, and BMs. The number of DSPs and BRAMs required is taken directly from the output of PlanAhead. The resulting values, the base resource requirement for the module, is then passed into the `ModuleGenerator` program to compute a list of the 'top' shapes for the module.

Prior Work

Module shaping for workflows mentioned in Section 2.4 generally used an iterative approach towards determining what shape to use for each module [20][22][21]. The resource requirements for each module is first computed through a vendor tool such as PlanAhead. Taking these resources into account, the shaper iterates through the device fabric to find acceptable starting points for a module. BRAM and DSP tiles are prioritized as they are relatively sparse in comparison to CLBs. Once an acceptable starting location is found, the shape is expanded until a undesired resource is encountered or the edge of the device has been reached. Once the requirements for BRAM/DSPs are met, the shaper continues to expand the shape horizontally until the CLB requirements are also met. Workflows that utilize this iterative shaper methodology for determining module shapes typically do so to optimize for minimal wire length between interconnected modules. By placing interconnected modules close together, optimal timing performance can be achieved (Figure 3.5). To optimize the placement for such modules, techniques such as linear programming have been utilized to find the optimal set of placements for all modules in the design [23].

```
create_project -force project_1 project_1 -part <DEVICE>
set_property design_mode GateLvl [get_property srcset [current_run]]
set_property edif_top_file <MODULE>.ndf [get_property srcset [current_run]]
add_files -norecurse <MODULE>.ndf
add_files -norecurse .
import_files -force -norecurse
open_netlist_design -name netlist_1
report_stats -tables {primitive} -file <MODULE>.stats -format table
exit
```

Figure 3.3: PlanAhead script for resource estimation

```
Primitive Statistics
====================

Primitive type Count
---------------------
FLOP_LATCH     5033
LUT            1499
MUXFX          4
CARRY          571
IO             162
BMEM           9
MULT           82
DMEM           3260
OTHERS         159
```

Figure 3.4: Resource estimation output from PlanAhead

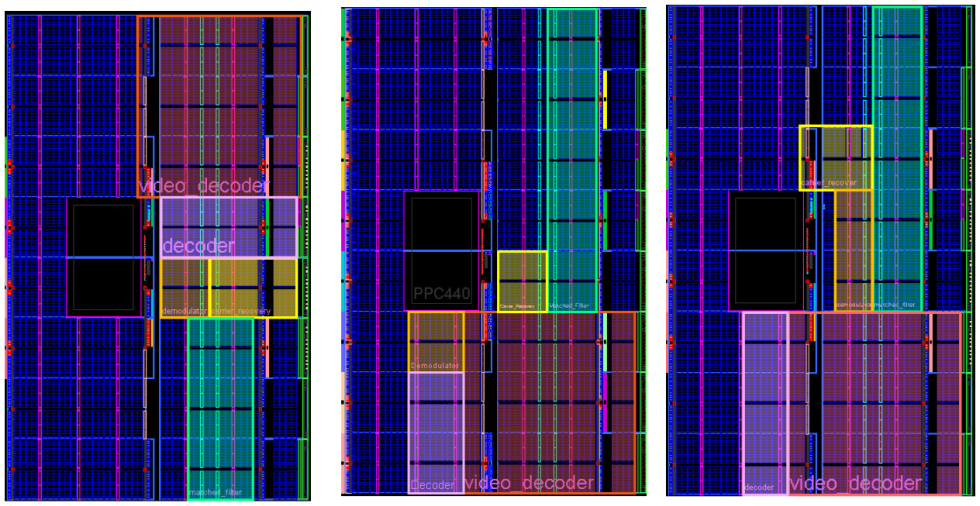


Figure 3.5: On-demand module shaping

There were a number of issues with this approach, with the main problem being the lack of relocatability of such modules when built with a predetermined placement. As modern FPGAs are becoming larger and increasing heterogeneous, a single shape becomes increasingly limited in terms of possible placements on the FPGA fabric. Module generation time is also an issue with this method as each module is built after the optimal shape and placement is found. This extra time is added to the design compilation process, further hindering developer productivity. The resulting shapes are also specific to the module's placement and association with other modules being used in the design, leading to limited reuse and reduced effectiveness from caching previously generated shapes.

In TFlow, the module library implementation was also incapable of supporting multiple shapes per module. Due to limitations imposed by the structure of the module library, TFlow tools in the assembly phase are designed to only utilize one set of placed ports, PIPs, and nets for each module. As such, this limits a module to a single shape. In order to provide support for multiple shapes for each module, it was determined that both the TFlow tools as well as the metadata schema had to be revised for multi-shape support.

New Developments

The `ModuleGenerator` tool developed in this work aims to improve the shape generation process to allow for the creation of multiple efficient shapes for a given module. As such, the revised tool takes in a number of parameters regarding a module and return a listing of top shapes meeting those requirements. In addition to the estimate of required resources, a number of optional parameters can be specified as well. These optional parameters include boundary restrictions, number of shapes to return, granularity of modules, and additional tolerances for each of the resources. The additional boundary restrictions can be used to restrict the device area where modules are allowed to be generated, while the extra tolerances allow for additional tiles to be allocated to help relieve routing congestion. The granularity of the shapes can be specified as well, allowing for the height of the shape to be a multiple of either the clock region or the minimum tile height of internal resources. Shape information is queried from a local database that is populated on the initial run for each device. If the database does not exist, or if the database does not contain the device requested, the `ShapeGenerator` tool is run to populate or append to the database.

The `ShapeGenerator` tool takes a minimal number of parameters, only the device and the granularity for the shapes needs to be specified. The tile information for the device is pulled from TORC and consists of the tile type, index, location, and name. This information is then assembled into a 2D array consisting of only CLBLL, CLBLM, DSP, and BRAM tiles to allow for fast iteration. Once the tile array is created, the shaper iterates over the array computing every possible shape. Computing all possible shapes for a module in this fashion is inherently a combinatorial problem. In a given FPGA device with $n \times m$ tiles, there are

$$\sum_{i=1}^{n}\sum_{j=1}^{m} i \times j = \frac{n(n+1)}{2} \times \frac{m(m+1)}{2}$$

possible rectangular shapes, and enumerating all possible shapes has a complexity of $O(n^2 \times m^2)$. As shown later, this results in a fairly lengthy runtime but the downsides are offset by the fact that it only needs to be run once per device. Each computed shape is stored in a

SQLite table named after the FPGA device and is identified by the following information:

- Shape boundaries (bottom and top X/Y coordinates, width, height)
- Contained resources (CLBs, BRAMs, DSPs, Miscellaneous)
- Aspect ratio, granularity, location relative to clock column
- Hash identifying the resource layout of given shape

To store the large amount of shape information generated by ShapeGenerator tool, SQLite was selected as the database of choice. In addition to being an embedded database engine, SQLite offers a fully self-contained SQL implementation in a minimal footprint. Furthermore, the database file format used by SQLite is designed to be cross-platform, allowing the full set of TFlow tools to run on devices running an embedded Linux platform.

In addition to the table containing all possible shapes, a separate table is created to hold placement information for each unique shape. This placement table stores only the unique hash of a shape, the granularity, and the number of placements for each unique shape. By storing both the individual shapes as well as an overview of the placement count for that shape, a wide variety of filtering options are available. Due to the large size of the database, a number of pragmas are also set to improve querying performance by loading the entire database into memory and increasing cache and paging sizes. SQL transactions are also used to reduce database insert overhead. Once all the shapes have been appended to the database, a number of indexes are created to allow for fast, efficient queries.

Once the shape database is populated, queries can be constructed to find the 'top' shapes for a given module. Two options are made available to the developer for shape selection. The first option instructs the shaper to focus on shape efficiency. The efficiency for shapes is computed as a product of extra, unused resources and a penalty value. Abundant resources, such as CLBs incur a lower penalty value while sparse resources, such as BRAMs and DSPs,

```
1 PENALTY_QUERY = ( <CLB_PENALTY> * clb_count + <BRAM_PENALTY> * bram_count +
      <DSP_PENALTY * dsp_count + <MISC_PENALTY> * misc_count )
2 RESOURCE_QUERY = ( clb_count >= <MIN_CLB> AND bram_count >= <MIN_BRAM> AND
      dsp_count >= <MIN_DSP> )
3 CONSTRAINT_QUERY = ( loc_x_min >= <MIN_X> AND loc_y_min >= <MIN_Y> AND loc_x_max
      < <MAX_X> AND loc_y_max < <MAX_Y> )
4 CREATE TEMP VIEW <DEVICE>_temp AS SELECT hash, <PENALTY_QUERY> AS penalty FROM
      <DEVICE> WHERE granularity=<GRANULARITY> AND <RESOURCE_QUERY> AND
      <CONSTRAINT_QUERY>
5 SELECT hash, placements FROM <DEVICE>_placements WHERE granularity=<GRANULARITY>
      AND hash in (
6     SELECT hash FROM <DEVICE>_temp GROUP BY hash ORDER BY penalty ASC LIMIT
          <NUM_SHAPES>
7 ) ORDER BY placements DESC LIMIT <NUM_SHAPES>
```

Figure 3.6: SQL query optimized for efficiency

incur a higher penalty value. Specialty tiles containing non-programmable logic such as dedicated I/O or embedded processors incur an even higher penalty value to prevent them from being used in generated shapes. The first stage in the efficiency query is to compute the efficiency metric for all shapes in the database (Figure 3.6, line 4). This returns a temporary table containing two columns: the hash string and penalty value computed for each shape that meets the resource and constraint requirements.

From this table, a query is made to retrieve the shapes with the lowest penalty values, and therefore having the highest resource efficiency (Figure 3.6, line 6). The table returned from this query is configured to only return one shape per unique hash value, with the size of the table limited to the number of shapes desired. Once the most efficient shapes for the module are found, a third query is made to retrieve and sort the shapes by the number possible placements for each shape (Figure 3.6, lines 5,7). The resulting table from this query is a list of the most efficient unique shapes for the module, sorted by the number of placements for each shape.

```
1 RESOURCE_QUERY = ( clb_count >= <MIN_CLB> AND bram_count >= <MIN_BRAM> AND
      dsp_count >= <MIN_DSP> )
2 CONSTRAINT_QUERY = ( loc_x_min >= <MIN_X> AND loc_y_min >= <MIN_Y> AND loc_x_max
      < <MAX_X> AND loc_y_max < <MAX_Y> )
3 SELECT hash, placements FROM <DEVICE>_placements WHERE granularity=<GRANULARITY>
      AND hash in (
4     SELECT hash FROM <DEVICE> WHERE granularity=<GRANULARITY> AND
          <RESOURCE_QUERY> AND <CONSTRAINT_QUERY> GROUP BY hash;
5 ) ORDER BY placements DESC LIMIT <NUM_SHAPES>
```

Figure 3.7: SQL query optimized for placements

The second option provided for shape selection puts a focus on maximizing the number of possible placements, irregardless of the efficiency as long as the resource requirements are met. This option results in shapes with more overall placements, but comes with the trade-off of less efficient shapes. A query is first made for all unique shapes that meet the resource and constraint requirements (Figure 3.7, line 4). A second query is then executed to sort the returned shapes by the number of possible placements (Figure 3.7, Lines 3,5). The resulting table contains a list of unique shapes that have the highest number of possible placements on the device while still meeting the resource requirements of the module.

After the list of top shapes have been found, the shaper tool iterates through each unique shape and pulls a single instance of that shape on the device to build the User Constraint File (UCF). The UCF is used later on in the compilation process to restrict where the module is placed on the device. The location of the shape at this point is irrelevant as TFlow will relocate the module as necessary during the assembly phase. Once an instance of the shape is pulled from the database, the details for the shape is written to the console to provide the user with a visual aid (Figure 3.8) of what the shape looks like and the resource pattern making up the shape. The TORC API is used to find the site boundaries for the resources contained in the shape. The resulting constraints are then written to a UCF named after the module, appended by an incremental ID for each returned shape (Figure 3.9).

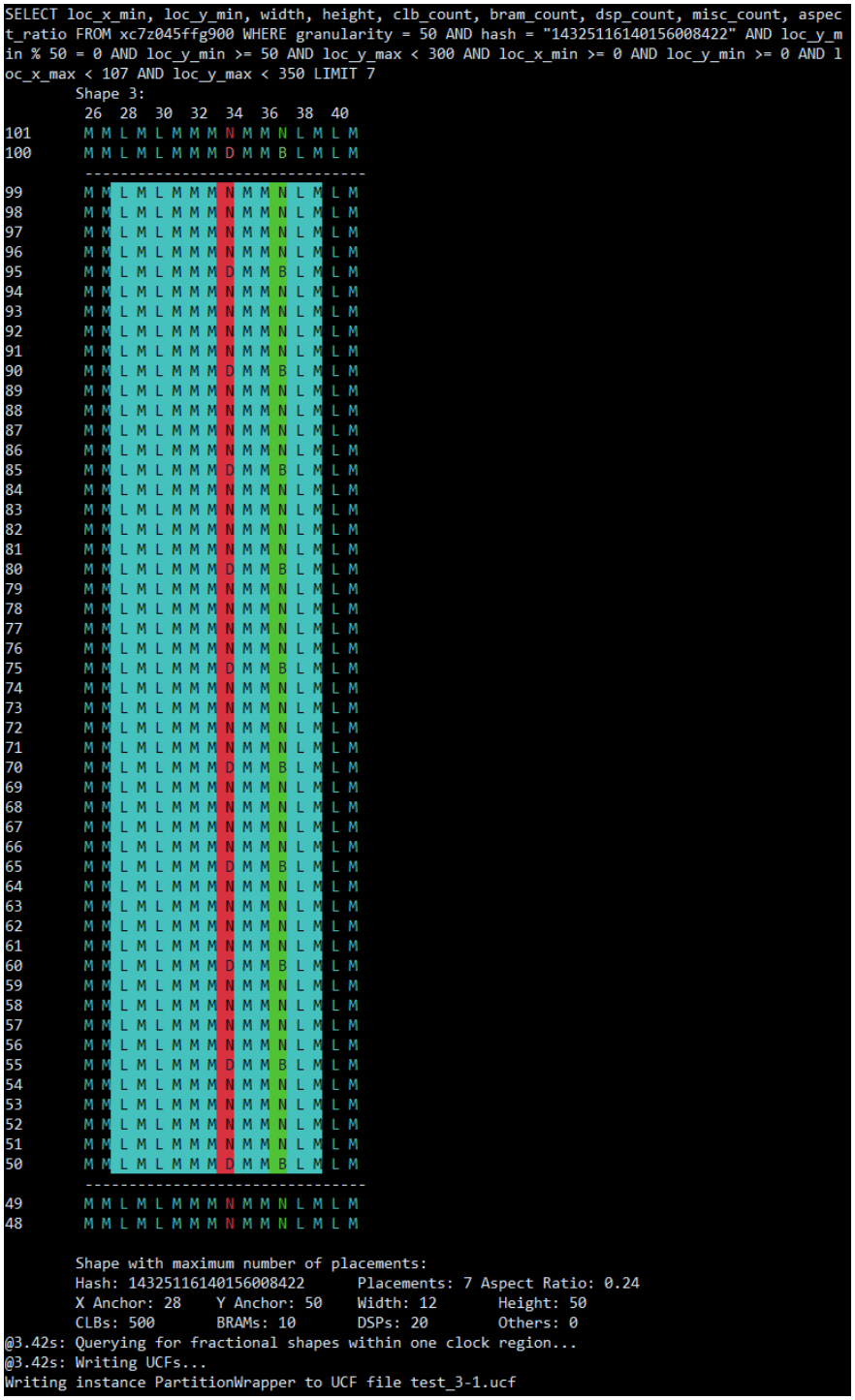


Figure 3.8: Console output of sample shape

```
1 INST "PartitionWrapper" AREA_GROUP = "pblock_test_1-1";
2 AREA_GROUP "pblock_test_1-1" RANGE=SLICE_X48Y50:SLICE_X67Y99;
3 AREA_GROUP "pblock_test_1-1" RANGE=RAMB18_X3Y20:RAMB18_X3Y39;
4 AREA_GROUP "pblock_test_1-1" RANGE=RAMB36_X3Y10:RAMB36_X3Y19;
5 AREA_GROUP "pblock_test_1-1" RANGE=DSP48_X3Y20:DSP48_X3Y39;
```

Figure 3.9: UCF result for sample shape in Figure 3.8

3.1.3 Map, Place, Route, Bitstream

Once the UCFs defining the shapes are created, the module compilation process continues through the translate, map, place and route, and bitgen stages. A number of build scripts are used to automate the full compilation process, giving the designer the benefits of the TFlow workflow without having to dive into the details. The TFlow workflow utilizes Xilinx Partitions for compiling modules to ensure routing constraints are enforced. The EDIF generated in post-synthesis stage is first modified to work with Xilinx Partition followed by compilation through the Xilinx tools with TFlow utilities inserted in at specific points in the build process to retrieve metadata on the module being built. This metadata includes information on the location of each used port, routed nets, PIPs, and the boundaries of the module. The TFlow module compilation process is shown in Figure 3.10. An important point to note is that the module compilation process is heavily parallelizable. Once the design is synthesized, the remaining compilation steps can be executed in parallel to allow for minimal turnaround times.

During the build process, the list of possible placements for a shape is further refined in a pre-placement phase. Rather than looking at the top level resources such as CLBs, BRAMs, and DSPs, the pre-placer inspects the tile pattern within the module's boundaries using the TORC API. Using this tile pattern and TORC, the pre-placer iterates over the device tiles to find other placement locations on the device that contain an identical configuration bit pattern. Once all possible placements are found, routes within the placements are checked for

compatibility. The resulting placements that satisfy all resource checks are then appended to the metadata file for the module.

3.1.4 Module Registration

Once a number of modules are compiled with their corresponding shapes, they are added into the *module library* where they can then be used by any design on which the device was built. To add in capability for multi-shape modules, this library structure was modified to allow the assembly stage to discern between modules utilizing different shapes. The resulting structure of the library is a flat hierarchy of folders containing the individual modules postfixed by the ID number of the prior generated shape that it used. The folder for each module is created only if it successfully compiled, which can be checked by looking for the existence of the bitstream file in the module's build folder. If such a file exists, the module is known to have been successfully compiled and additional actions are taken to add the module to the library.

After a module is successfully compiled, the metadata associated with the module is first updated to include the ID of the shape that the module was built with. This is done by iterating through the build folder for each module, reading the shape ID from the folder name, and finally updating the shape and placement ID fields in the metadata file with the associated shape ID. The module's shape ID as used here acts as a pointer to associate the the shape with the module that uses it. An excerpt of the metadata XML file containing the placement and shape information as well as the shape ID is shown in Figure 3.11 with the shape ID found on lines 14 and 40. Once the metadata file is updated, the module is registered in the library by copying the NCD, BIT, XDL, and XML files from the module's build folder to its corresponding folder in the module library.

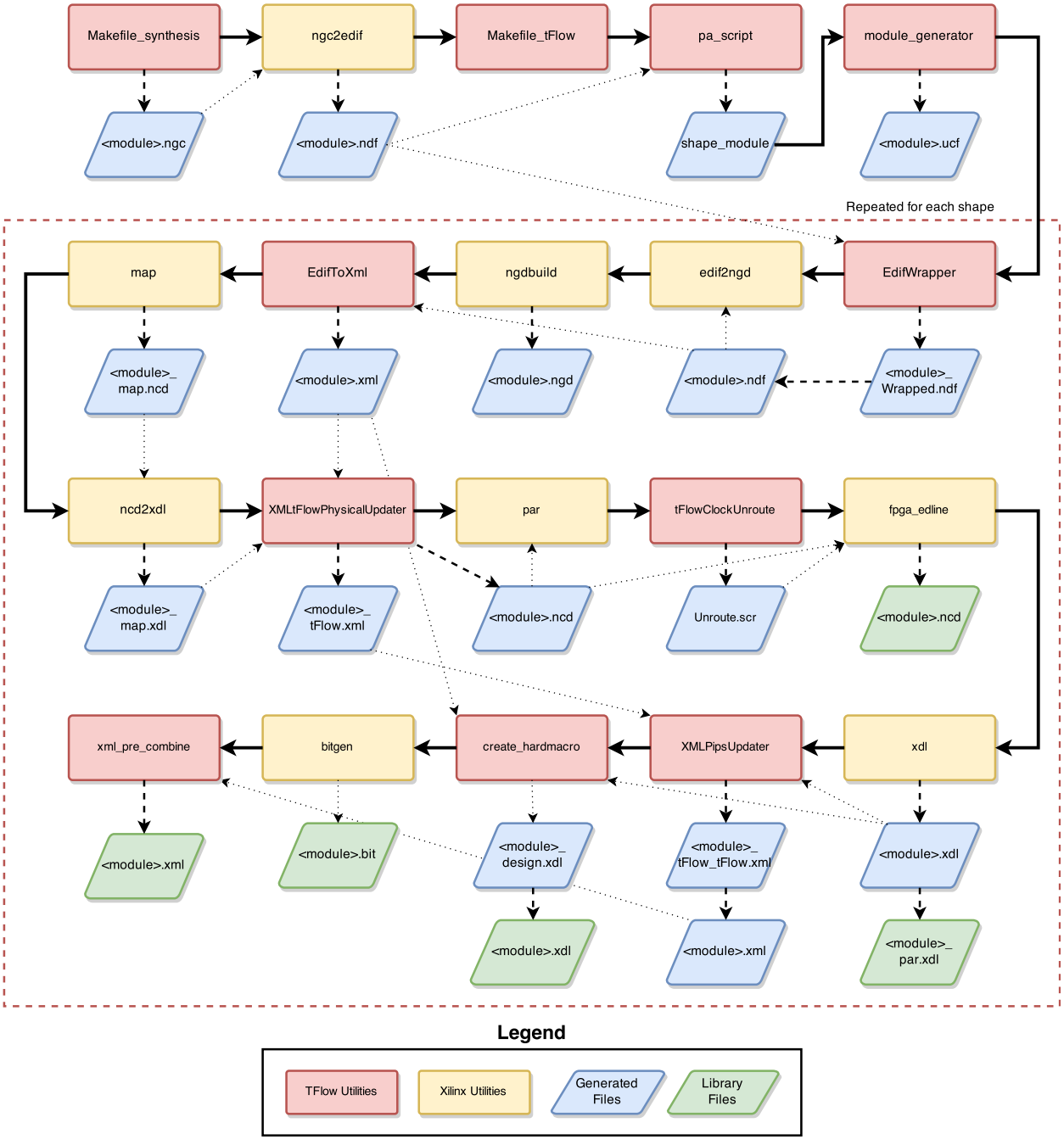


Figure 3.10: Module compilation flowchart

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
!DOCTYPE boost_serialization>
<boost_serialization signature="serialization::archive" version="9">
<xmlmodule class_id="0" tracking_level="1" version="0" object_id="_0">
    ...
    <mPlacements class_id="14" tracking_level="0" version="0">
        <count>7</count>
        <item_version>0</item_version>
        <item class_id_reference="2" object_id="_51978">
            <mX>43</mX>
            <mY>300</mY>
            <mIndex>-1</mIndex>
            <mType></mType>
            <mID>noc_block_axi_fifo_loopback_reg_1</mID>
        </item>
        ...
    </mPlacements>
    <mShapes class_id="15" tracking_level="0" version="0">
        <count>3</count>
        <item_version>0</item_version>
        <item class_id="16" tracking_level="1" version="0" object_id="_51985">
            <mBoundaries class_id="17" tracking_level="0" version="0">
                <count>1</count>
                <item_version>0</item_version>
                <item class_id="18" tracking_level="1" version="0" object_id="_51986">
                    <sw_x>-3</sw_x>
                    <sw_y>0</sw_y>
                    <ne_x>6</ne_x>
                    <ne_y>49</ne_y>
                </item>
            </mBoundaries>
            <mAnchor class_id_reference="2" object_id="_51987">
                <mX>43</mX>
                <mY>50</mY>
                <mIndex>0</mIndex>
                <mType>BRAM_R</mType>
                <mID></mID>
            </mAnchor>
            <mID class_id_reference="1" object_id="_51988">
                <mName>noc_block_axi_fifo_loopback_reg_1</mName>
            </mID>
        </item>
        ...
    </mShapes>
    ...
</xmlmodule>
```

Figure 3.11: Excerpt of placement and shape metadata

3.2 Static Creation

The process for creating the static is similar to the module creation process. Rather than encapsulating a specific functionality as the modules aim to do, the static region aims to combine the invariant part of the design such as I/O ports, memory interfaces, embedded processors, and other static logic. The steps involved in the static creation mirror those of the module creation process; the HDL design is synthesized, mapped, placed, routed, and a bitstream is created for use in the assembly phase. The static however is required to leave a *sandbox* region empty to allow for modules to be placed. Once the static is built, a metadata file is created for the static containing information such as boundaries and port information.

Chapter 4

Design Assembly

This chapter discusses the second phase in the TFlow design workflow. In this phase, a design comprised of a number of individual modules along with the static is first specified by the designer. These modules, generated in the previous phase, are pulled from the module library and combined with the static to create a programmable bitstream for the target FPGA in a matter of seconds. As long as the desired modules were pregenerated in the first phase, TFlow can be utilized to generate the bitstream for a full design with minimal turnaround time.

4.1 Design Entry

The design specifying the modules and associated connectivity graph is entered as an EDIF file. As the EDIF file format is well documented, TFlow is compatible with any front-end tool that is capable of generating data in this manner. An example front-end tool is GReasy, a TFlow enhanced version of GNU Radio. While GNU Radio is normally tailored towards software developers who have limited knowledge of HDL, the addition of TFlow allows designers to take advantage of the computing power and flexibility of FPGAs in their designs

without having to manually write HDL code. In GReasy, a module library can be pre-generated and provided to the designer as a set of graphical blocks. Once the designer has assembled their design using GNU Radio's graphical interface, the module and connectivity information is exported as an EDIF file to be used by TFlow.

4.2 Module Relocation and Placement

Once the EDIF file specifying the design is generated, TFlow proceeds to the module assembly phase. In the TFlow assembly phase, a separate metadata file is created to contain design information used by TFlow for rapid module assembly. The metadata file reflects the structure of the EDIF and XDL files and is comprised of both the static design as well as all the modules to be placed on the device. In order to support multi-shape modules, modifications had to be made to the way modules were specified in the design metadata file. Previous works assumed a single shape for each module, and as such each module were assumed to be identical in terms of internal tiles and PIPs. As such, the design metadata file contained only a single copy of tiles and PIPs for each module. With the addition of multi-shape support, each module had to be 'dereferenced' to store their own set of resource information. To do this, the XML schema that TFlow uses to store metadata had to be modified to allow for greater flexibility.

When the design metadata file is first created, entries for each module exists only as a placeholder. Each module included in the design has a number of parameters: a unique ID, anchor point, and a reference to the resource information making up that module (Figure 4.1, lines 8-26). As the module resource information is initially empty, the corresponding module from the library is first copied in. The copying process is done over two stages: the placement locations and shapes are first copied prior to the placement algorithm execution. After each module in the design has been placed on the device, the associated tiles and PIPs are copied over to the metadata file.

As all of the possible relocation locations for each module was included in the module compilation phase, placement of the modules simply involves reading the previously computed placement information from the metadata file of each module. Once the placement data is retrieved, modules that overlap with the static portion of the design are first removed by examining the module's boundaries at each placement location. Once this is done, the actual placements for each module is determined through the use of placement algorithms. A number of placement algorithms have been implemented in TFlow, including brute force, first valid, limited random, and full random. For each possible placement combination, the quality of the placement is computed by computing the Manhattan distance between each module in the connection graph. The placement combination with the best routing metric is selected as the final placement and the final placement information is recorded in the design metadata file.

Once each module has a selected shape and placement, the corresponding tiles and PIPs for each module are updated to reflect the new coordinates. In the module generation phase, tile and PIP coordinates are saved relative to an anchor point located somewhere within the module. With multiple shapes, the coordinates for each tile will differ between shapes. An additional stage in the TFlow compilation workflow was added at the end of the place and route phase to update all tile coordinates for each module to reflect it's coordinate from post-relocation.

4.3 Routing and Bitstream

Once modules are placed and the metadata updated to reflect the module's actual location, TFlow proceeds to the inter-module routing phase. In place of using vendor tools for routing, TFlow utilizes a custom router primarily focused on execution speed, following by routabiity and timing performance. Once the PIPs required for the inter-module routing have been extracted, TFlow moves to the final stage to assemble the full design.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<!DOCTYPE boost_serialization>
<boost_serialization signature="serialization::archive" version="9">
<xmlModule class_id="0" tracking_level="1" version="0" object_id="_0">
    <mName class_id="1" tracking_level="1" version="0" object_id="_1">...</mName>
    <mAnchor class_id="2" tracking_level="1" version="0" object_id="_2">...</mAnchor>
    <mPips class_id="7" tracking_level="0" version="0">...</mPips>
    <mChildren class_id="8" tracking_level="0" version="0">
        <count>3</count>
        <item_version>0</item_version>
        <item class_id="9" tracking_level="1" version="0" object_id="_7">
            <mName class_id_reference="1" object_id="_8">
                <mName>FFT0</mName>
            </mName>
            <mAnchor class_id_reference="2" object_id="_9">
                <mX>-1</mX>
                <mY>-1</mY>
                <mIndex>-1</mIndex>
                <mType></mType>
                <mID></mID>
            </mAnchor>
            <mModule class_id_reference="0" object_id="_10">...</mModule>
            <mHash class_id_reference="6" object_id="_420">...</mHash>
            <mHardMacroLevel>0</mHardMacroLevel>
        </item>
        ...
    </mChildren>
    <mPorts>
        <item class_id_reference="11" object_id="_839">
            <mName class_id_reference="1" object_id="_840">
                <mName>ce1i_tdata</mName>
            </mName>
            <mDirection>0</mDirection>
            <mBusIndex>0</mBusIndex>
            <mTiles>
                <count>0</count>
                <item_version>0</item_version>
            </mTiles>
        </item>
        ...
    </mPorts>
    <mShapes>...</mShapes>
    <mNets>...</mNets>
    <mClockNets>...</mClockNets>
</xmlModule>
</boost_serialization>
```

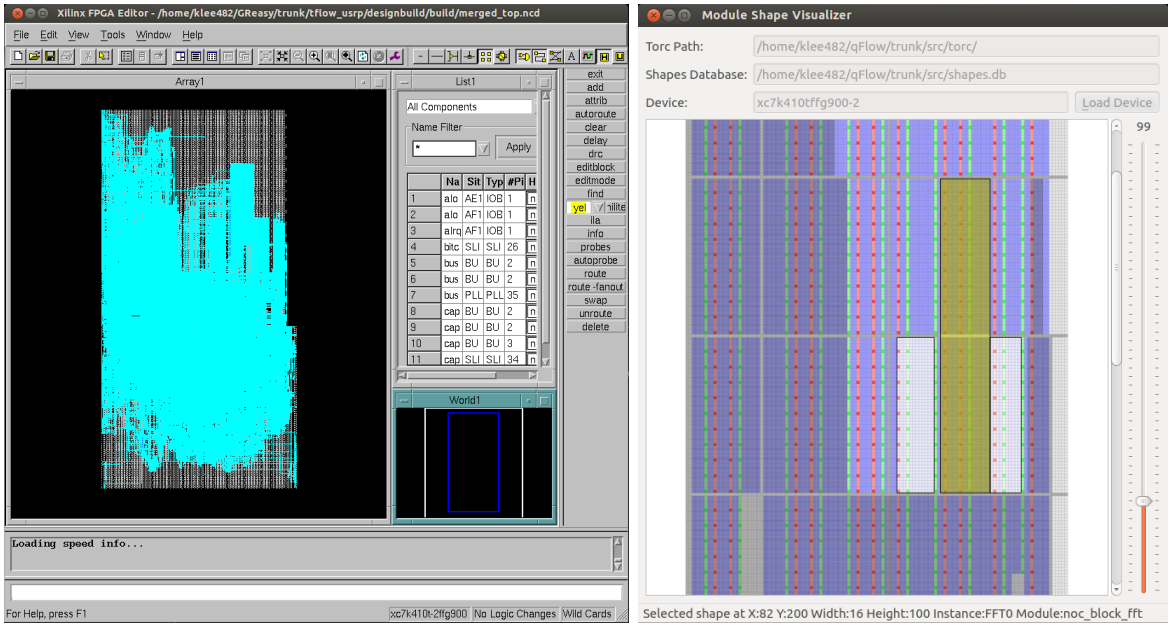
Figure 4.1: Exerpt of design metadata

In the bitstream stitching stage of the assembly process, the final bitstream for the FPGA device is created. The static bitstream is combined with the bitstreams of each module used along with the micro-bitstream for each inter-module and clock route. By stitching together the design via bitstreams, total assembly time is minimized.

4.4 Placement Visualization

One of the previous downsides with using TFlow is that the bitstream stitching methodology for rapid design assembly does not allow a designer to visualize the resulting set of placed modules. In the vendor compilation workflow, the NCD used to generate the bitstream is passed to a tool such as fpga_editor for visualization (Figure 4.2a). Creating this NCD however, involves running a lengthy debug extension in TFlow to create an XDL representation of the design that can be further converted into a vendor recognizable NCD design. These vendor tools however, require significant expertise and domain knowledge to use, and is thus a significant barrier to overcome for designers who lack such knowledge.

To remedy this, a custom visualizer tool was developed to read and parse the design metadata file and show the results of the assembled TFlow design in an intuitive way (Figure 4.2b). In addition to visualizing the static and sandbox regions of the device, the visualization tool also shows the resource layout at the tile granularity, allowing the user to see the underlying pattern of CLBs, BRAMs, and DSPs. Inter-module connectivity is also shown for designs where modules are connected in a pipeline fashion to provide the designer with feedback on module placement (Figure 4.3).



(a) FPGA editor (b) Custom visualization tool

Figure 4.2: Visualization of utilized regions on device

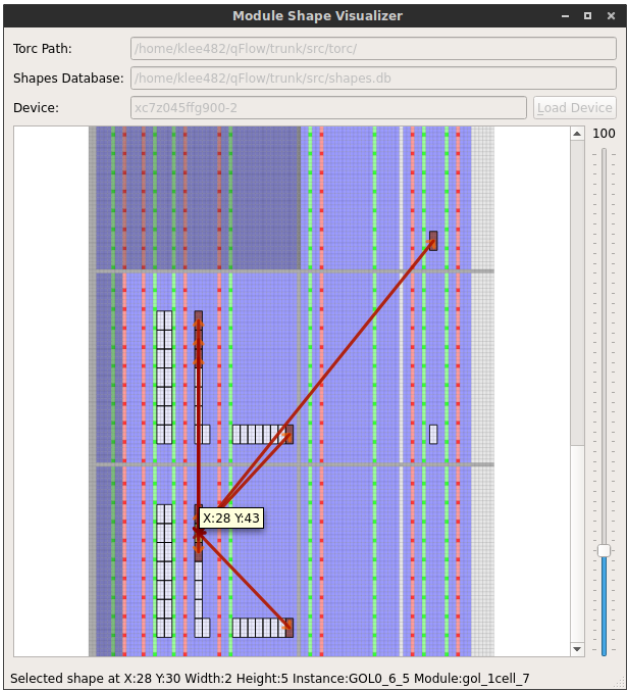


Figure 4.3: Inter-module Connectivity

Chapter 5

Implementation and Results

The unique paradigm of TFlow gives designers the ability to rapidly assemble FPGA designs without having to go through the traditionally time consuming compilation process. By breaking the design space into two parts and utilizing the modular design methodology, turnaround times for building a single module is reduced and designs can be rapidly assembled when provided a library of such modules. With the additional support for multiple shapes, modules gain an increase in terms of resource efficiency as well as placement flexibility. To measure the benefits resulting from the use of additional shapes for each module, shaping data was generated for a number of devices. The results were analyzed, with a focus placed on the improvements in module resource efficiency, placement locations, and overall device coverage.

5.1 Placement and Efficiency Gains

As mentioned in Section 2.5, modules build for PR are typically limited in granularity to the height of a clock region in the device. This, however, results in resource allocation inefficiencies within the module. With modules that span a full clock region, resource usage

Table 5.1: Vertical expansion of clock region size for modern Xilinx FPGA devices.

Xilinx FPGA Family	CLB	BRAM	DSP
Spartan-6	16 rows	4 rows	4 rows
Virtex-4	16 rows	4 rows	8 rows
Virtex-5	20 rows	4 rows	8 rows
Virtex-6	40 rows	8 rows	16 rows
Series-7	50 rows	10 rows	20 rows

is rounded up to a multiple of the resource count in a single clock frame. For example, in a device with 50 CLBs and 10 BRAMs in a single clock frame, a module utilizing 202 CLBs and 4 BRAMs would take at minimum 250 CLBs and 10 BRAMs, resulting in 20% extra CLBs and 140% extra BRAMs. Any extra unused resources becomes wasted as device primitives can only be allocated to one module at time on the device.

Furthermore, in the latest Xilinx FPGAs, there also exists a new shared clocking mechanism between adjacent columns of tiles. When relying on Xilinx Partitions to create modules, every module must have an even number of columns, as one clock line is shared between every two columns. This can cause additional wasted resources, as the resource granularity has shrunk again. For example, a module that only requires 40 CLBs in a device with a frame height of 50 CLBs would be able to consist of a single CLB frame. However, this new restriction forces it to contain an another adjacent frame, which may contain additional CLBs, BRAMs, or DSPs. These additional resources may not be utilized by the module, but with this PR requirement they are reserved and cannot be used by any other module.

In older devices, resource wastage was less of a concern due to the smaller frame height of the device (Table 5.1). For example, the Virtex 5 has a frame height of twenty CLBs whereas the Series-7 generation of programmable devices (Virtex-7, Zynq etc) have a frame height of fifty CLBs. In terms of DSPs and BRAMs, the Virtex-5 has four BRAMs and eight DSPs in a single frame while the Virtex-7 has ten BRAMs and twenty DSPs (Every BRAM36 can be also used as two BRAM18s). As devices continue to increase in size, inefficiencies resulting

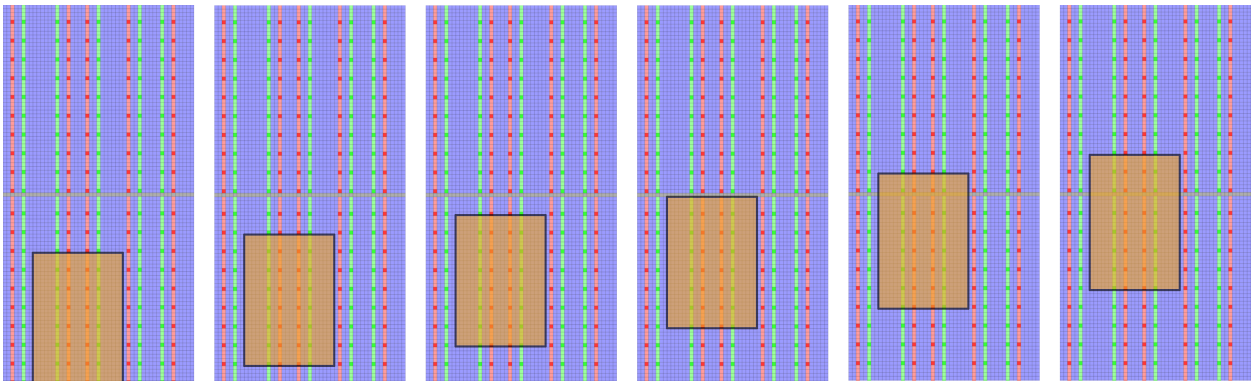
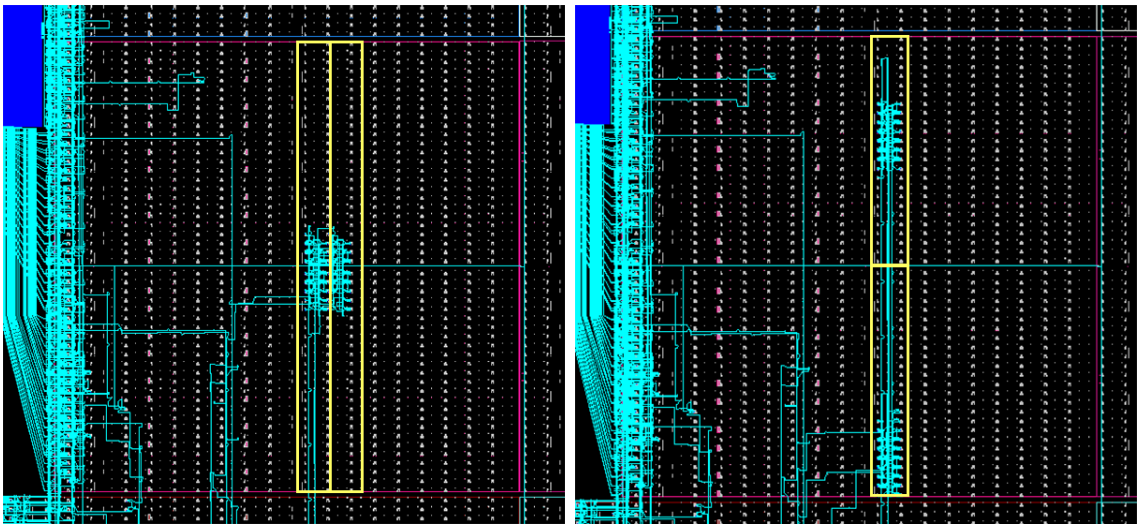


Figure 5.1: Six of ten possible offsets for a sub-frame module within a clock frame

from the height of the clock region will only continue to increase. Unlike BRAMs and DSPs, inefficiencies in terms of extra CLBs is acceptable as the ratio of CLBs to other resources is orders of magnitude greater. In fact, extra CLBs are often desired in a module to ease internal routing congestion. Inefficiencies in terms of BRAMs and DSPs on the other hand, poses no such benefits and only limit the resource usage efficiency on the device.

To address the increasing frame height with newer devices, the frame height modular constraint must be relaxed. By doing so, a reduction in the granularity of primitives is possible; however, components such as BRAMs and DSPs can take up many tiles worth of resources relative to CLBs. Thus, modules with these components have a minimum granularity that is dependent on the tile height of the resources. To maintain consistency, this size can be used as the new modular granularity. For the Series-7 devices, the largest resource is a BRAM36 with a height of five tiles. With this new granularity, the height for a module can be reduced from the size of a clock frame to a minimum of one BRAM36, two BRAM18s, five CLBs, or two DSPs per column. With the use of such subframe modules, high resource efficiency can be achieved even as the size of a clock frame continues to increase.

For the Series-7 devices, such a increase in the granularity results in a 10× increase in the number of placements for a module as each module can be shifted within a clock region by an offset matching the minimum granularity (Figure 5.1). A module built with the granularity



(a) Non-fractional placement (b) Fractional placement

Figure 5.2: Space utilization of two modules

of a full clock region can be shifted within the device to any other clock region; the number of placements for such a module matches the number of clock regions on the device. In contrast, modules built with sub-clock region granularity can have an order of magnitude more options for placement. Due to this increase in resource efficiency, a greater number of modules can be placed in a single clock region as well. (Figure 5.2). It is important to note however, that while a subframe module has more overall placements, many of the placements will overlap with each other. The main benefits then of sub-frame granularity, and therefore more placements, lies in the increase in placement flexibility for individual modules.

Increasing the granularity introduces some downsides as well. When a module is built with sub-frame granularity, it can only be relocated to other clock regions while keeping the same offset. This is due to the horizontal clock tiles that lie in between clock regions, and as such each offset of a module within a clock region must be built as a unique shape for that module. Because of this, moving from full clock frame to sub-clock frame granularity increase the number of shapes that each module is built with. If a module was built with a number of different shapes, with each shape having a number of offsets within a clock region, the

number of possible placements can increase by an order of magnitude or more. While this increases the algorithm complexity when computing the optimal placement for each module, by virtue of the increased granularity that the modules operate at, computing all possible placements for a given set of modules on a device is computationally trivial. Even a brute force methodology that checks every possible placement combination is achievable in a short amount of time.

Another issue that arises with the use of multiple shapes lies in the time required to build each set of module. With the modular nature of the workflow, every shape for each unique module is run through its own compilation cycle (Figure 3.10). While the amount of time required to compile each module varies according to its size, compilation times remain fairly consistent when building the same module with various shapes. As such, when the shapes for a module are built in a sequential fashion, the time required to build a module with all its shapes scales linearly with the number of shapes that the module is built with. There are however, no dependencies in the compilation process in between modules being built. As such, the time required for the compilation of all modules can easily be reduced by building the modules in parallel. By parallelizing the module compilation process, the turnaround time can be reduced to the maximum amount of time required to compile a single module.

It is important to note that with TFlow, the module compilation process is also done in advance. Each module is built in the offline phase of development, the final assembly time is unaffected by the module compilation time. As the library of precompiled modules is created in advance, lengthy compilation times for the modules is largely irrelevant.

The use of sub-frame granularity incurs additional penalties as well, namely in the database size and creation time. While the generation time is less of a concern (as the database is only created once), the use of sub-frame granularity increases the total number of shapes to sort through as well as the size of the database by two orders of magnitude (Table 5.2).

Table 5.2: Shape database generation time and size

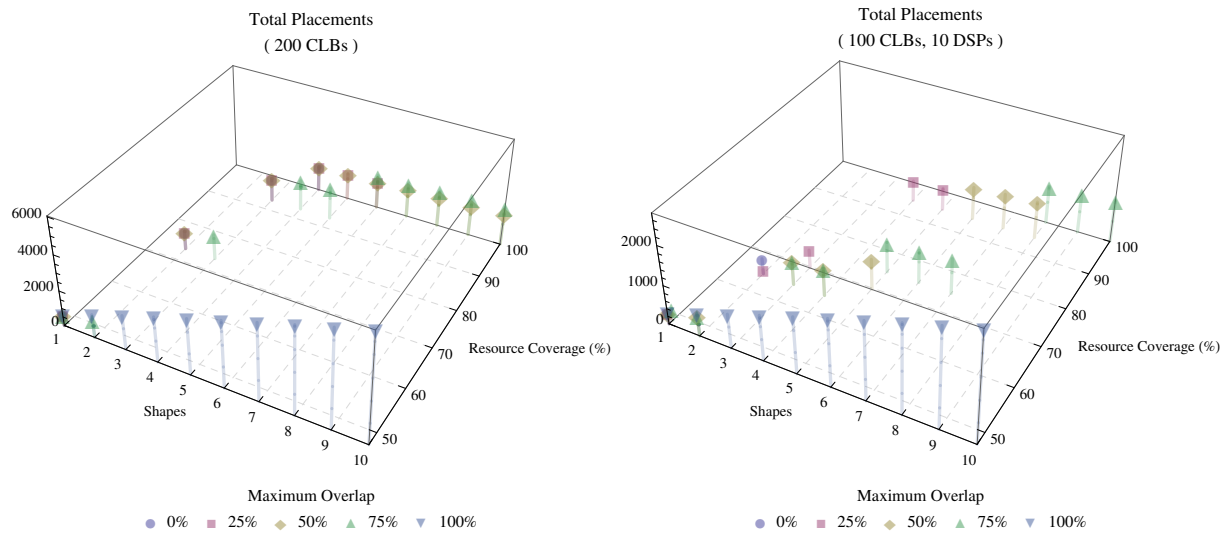
		Full Frame			Sub-Frame		
Device	Size	Time	Shapes	Size	Time	Shapes	Size
xc7z010	30×100	0.066s	56 (52)	34KB	1.23s	11.8K (1.04K)	2.6MB
xc7z020	75×150	1.44s	1.9K (1.2K)	525KB	15.9s	152K (51.9K)	37MB
xc7z030	78×200	2.15s	3.8K (2.5K)	1.1MB	39.8s	312K (132K)	79MB
xc7z045	106×350	8.04s	19.7K (9.8K)	4.9MB	387s	1.8M (622K)	437MB
xc7k160	72×250	2.63s	4.9K (2.9K)	1.3MB	57.5s	413K (173K)	104MB
xc7k325	94×350	6.54s	15.5K (7.8K)	3.9MB	279s	1.37M (526K)	346MB
xc7k410	122×350	11.3s	26.2K (11.9K)	6.5MB	589s	2.32M (775K)	579MB
xc7k480	123×400	15.2s	33.5K (12.9K)	8.2MB	853s	3.01M (859K)	744MB

(*) indicates unique shapes

5.2 Resource Coverage

With the heterogeneous nature of resource distribution on modern FPGA devices, the top shapes for a given module have a high chance to overlap as they can be shifted horizontally by a number of tiles. When building modules with sub-frame granularity, further overlap between shapes occurs as modules can be vertically shifted. Due to these overlaps, simply increasing the number of shapes for a module does not necessary improve the device utilization of the module. To improve device utilization then, a new metric was introduced into the shape selection process to find the optimal number of shapes to cover the entire device. This metric, the *resource coverage* of a module, is defined as the utilization of resources that the module contains in relation to the total resource count on the device. To give an example, the resource coverage of two modules on a Virtex 7 device is examined as an increasing number of shapes is taken into consideration.

For a module that uses only 200 CLBs, the top shape allows for only 50% of the CLBs on the device to be utilized (Figure 5.3a). This limitation arises from the heterogeneous nature



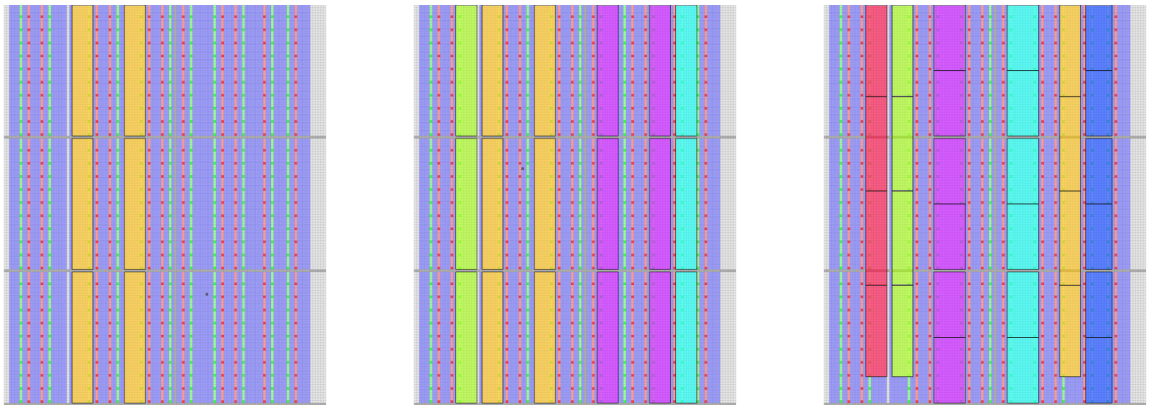
(a) Module with 200 CLBs

(b) Module with 100 CLBs, 10 DSPs

Figure 5.3: Placements and resource coverage on a Virtex 7 XC7Z045 device

of the resource distribution; the resource pattern making up the top shape only exists in half the columns on the device. If the number of shapes for the module is increased, the resource coverage for the module remains constant as the shaper would prefer to vertically shift the module, gaining more placements in the process. If only a limited amount of overlap is allowed however, the resource coverage of a module increases from 50% to 70%, 90%, and 100% with two, three, and four shapes respectively. The same pattern persists in modules that use both CLBs and DSPs or BRAMs (Figure 5.3b). With the additional resources, a greater number of resource patterns exist and as such more shapes are needed before full resource coverage can be achieved on the device.

A visual representation of the increase in device coverage and placements for a module utilizing 202 CLBs and 7 BRAMs on a Kintex 7 device is shown in Figure 5.4. A module with a single shape has six placements on the device, with each shape using 350 CLBs and 10 BRAM tiles. For all six placements combined, 16% and 18% of the CLBs and BRAMs respectively on the device are utilized (Figure 5.4a). With four shapes, 18 placements are possible while CLB and BRAM utilization increases to 48% and 55% respectively (Figure



(a) 6 Placements (b) 18 Placements (c) 30 Placements

Figure 5.4: Possible placements in three clock regions for a single module

5.4b). If sub-frame granularity is enabled, six shapes increases the placement count to 30, while also increasing the CLB utilization to 56% and BRAM utilization to 74% (Figure 5.4c).

5.3 GReasy

To test TFlow with multi-shape support in a real-world application, modules were built for GReasy, a software extension to GNU Radio designed to augment the platform with FPGAs without reducing developer productivity [24]. GNU Radio is an open source tool designed to allow users to test and prototype radio designs through an interactive drag and drop interface (Figure 5.5). By augmenting GNU Radio with TFlow, GReasy gives radio designers, who may not be familiar with HDL or the FPGA design process, the ability to accelerate their designs on an FPGA without the long development cycle normally associated with digital design. In doing so, GReasy takes full advantage of the modular design methodology offered by TFlow: within the GReasy environment, a library of FPGA-based processing modules encapsulating a specific function are created in advance and provided to the designer.

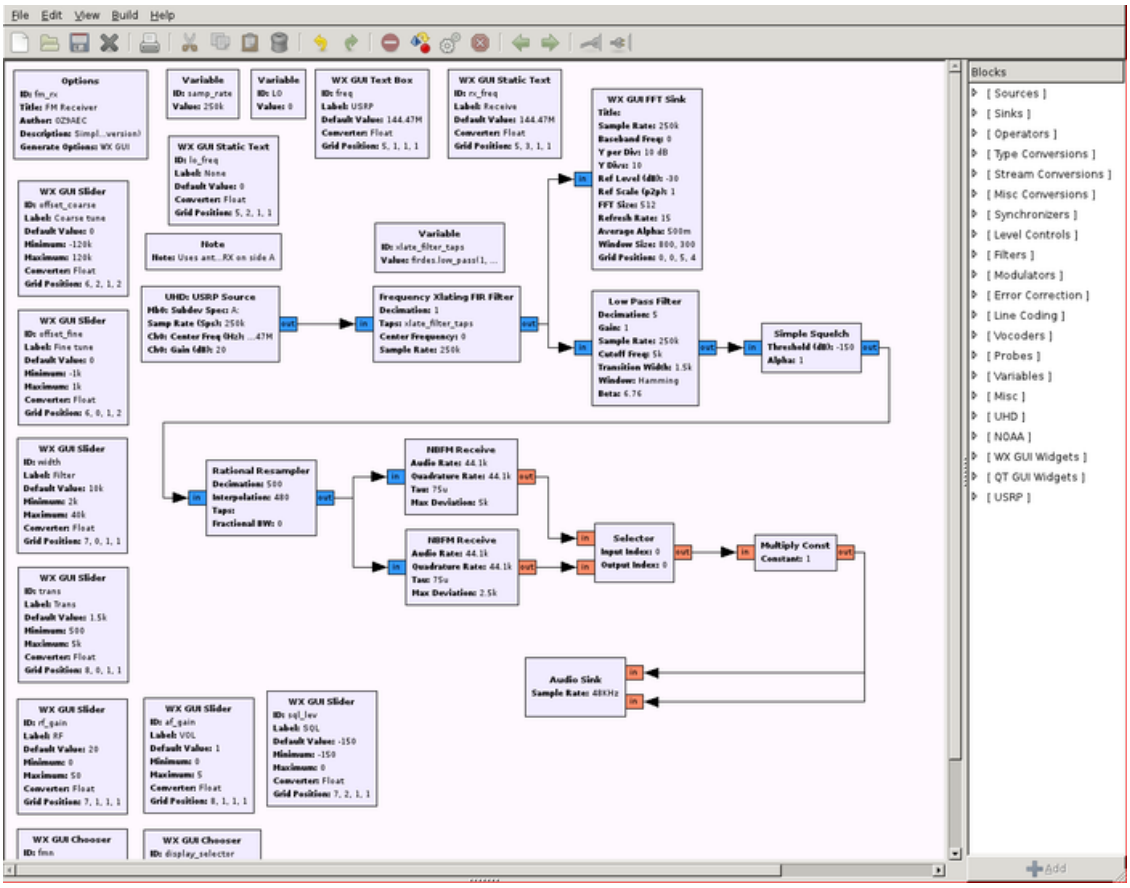


Figure 5.5: Interface for GNU Radio

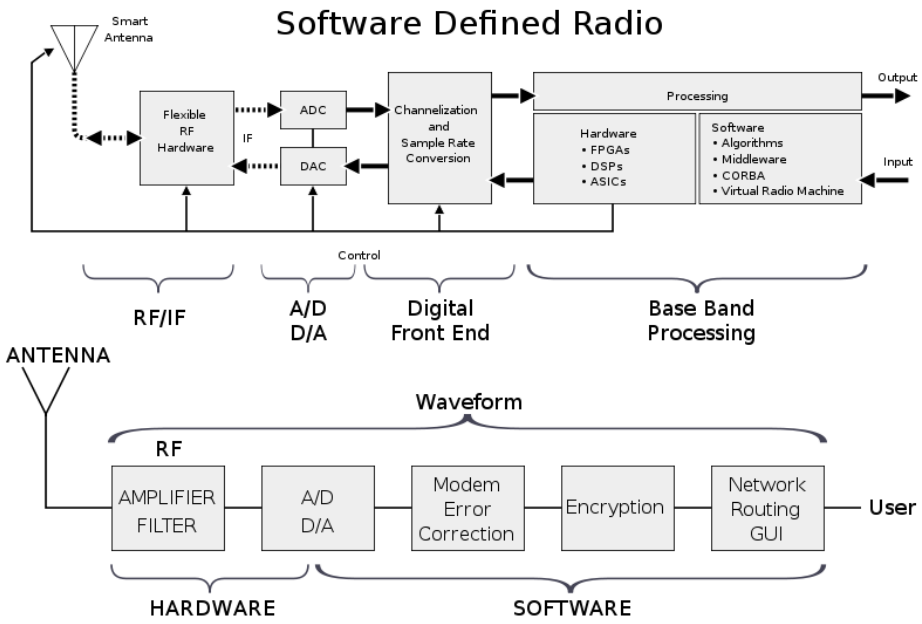


Figure 5.6: Software defined radio operation

Software defined radio in itself is a natural fit for FPGA acceleration. As a high bandwidth real-time application, the flowgraph for SDR designs can often be divided into a number of individual processing components such as filters, encryption/decryption blocks, and other DSP algorithms (Figure 5.6). These components in turn can have their functionality encapsulated into individual FPGA modules (radio blocks) for use by TFlow. As the front-end to TFlow, GReasy takes the designer's input and generates an EDIF from the GNU Radio flowgraph. This EDIF is then fed into TFlow, resulting in a fully assembled and operational design within a matter of seconds.

To explore the benefits of multi-shape modules within GReasy, tests were performed to compare the placement and resource efficiency gains when building FPGA-based radio blocks on the Kintex 7 XC7K410T device. Table 5.3 contains the the results returned by the shaper tool when building modules that occupy a full clock region. The Δ column contains the amount of extra BRAMs and DSPs that are allocated but unused by the module, resulting in inefficiencies ranging from 27% to 1900%. Table 5.4 contains the results when modules

Table 5.3: Full frame resource efficiency

	Required			Full Clock Frame				
Module	CLB	BRAM	DSP	CLB	BRAM	DSP	Δ	Time
Decimate by 16	168	0	4	250	0	20	**16**	4.2s
DES Encryption	495	1	0	500	20	0	**19**	4.2s
FFT	830	18	48	850	20	60	**18**	4.2s
Keep One in N	202	7	0	250	10	0	**3**	4.2s
Vector IIR	234	13	4	350	20	20	**23**	4.2s
FIR Filter	656	9	82	1050	30	120	**59**	4.2s

Table 5.4: Sub-frame resource efficiency

	Required			Sub-Clock Frame				
Module	CLB	BRAM	DSP	CLB	BRAM	DSP	Δ	Time
Decimate by 16	168	0	4	180	0	8	**4**	12.9s
DES Encryption	495	1	0	495	11	0	**10**	12.8s
FFT	830	18	48	870	18	48	**0**	13.6s
Keep One in N	202	7	0	245	7	0	**0**	12.9
Vector IIR	234	13	4	245	14	14	**11**	14s
FIR Filter	656	9	82	735	21	84	**14**	13.4

are built at the minimum sub-frame granularity. Resource wastage in this case is heavily reduced, with most modules requiring a minimum amount of extra tiles and some requiring no extra BRAMs or DSPs beyond what they require.

Shape query times were computed for each module using a system containing an Intel i7-2600 with 8MB cache and 12GB of RAM. Query times remained constant across a board, with little impact from the size of the module. Shapes built with sub-frame granularity took slightly longer to query due to the increased size of the database, but otherwise minimal time is added to the module generation process.

Chapter 6

Conclusion

With the increased abstraction achieved through the modular design methodology, rapid design assembly can be achieved through the use of pre-compiled modules alongside the right workflow. With the added consideration for multi-shape modules, individual modules can be made significantly more efficient while device resource utilization can also be increased as well. For statically placed modules, this results in better resource efficiency and therefore greater module density. For relocatable modules, the number of possible placement locations and total resource coverage can be drastically improved, while still keeping the module's footprint to a minimum. With a larger selection of shapes, designs utilizing a large number of modules have more flexibility when computing the set of optimal placements. Taking advantage of the offline nature of the compilation process, a large library of modules can be created in advance and provided to the user. For module recompilation, turnaround time is minimized through the use of parallel compilation. With the multi-shape augmentation to TFlow, designers gain the ability to operate at a higher abstraction level; productivity is increased without compromising the flexible and configurable nature of FPGAs.

6.1 Future Work

Future work that can build from this new module shaping capability is manyfold. Library management tools could be built containing a set of modules with their resource requirements and possible shapes. Shape selection can also be integrated with tools such as PlanAhead. PlanAhead already has the potential to perform resource estimation, but does not extend this capability into shape selection. User selection of shapes could also be added as a stand-alone application, so that placement locations and other options can be customized by the end user. The user can then analyze the placements and their metrics accordingly. Feedback from the placement algorithm can also be incorporated into the generated shapes, allowing for additional metrics such as internal module performance, route timing analysis, and spatial accessibility. The implementation of the shape database can be improved upon as well; reducing the metadata overhead of each shape will decrease the size of the database while adding in support for R* trees would allow for directional and geospatial queries.

Bibliography

[1] M. Wirthlin, B. Nelson, B. Hutchings, P. Athanas, and S. Bohner, "Future Field Programmable Gate Array (FPGA) Design Methodologies and Tool Flows," Air Force Research Laboratory, Wright-Patterson Air Force Base, OH, Tech. Rep. July, 2008.

[2] A. Love (2015), "A Modular Flow for Rapid FPGA Design Implementation," Master's Thesis, Virginia Polytechnic Institute and State University, Blacksburg, Virginia.

[3] Xilinx, "Hierarchical Design Methodology Guide," Xilinx User Guide UG748 (v 14.1 and later), Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_4/
Hierarchical_Design_Methodology_Guide.pdf, Xilinx, 2012.

[4] Xilinx, "Partial Reconfiguration User Guide", Xilinx User Guide UG702 (v 14.4), Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_4/
ug702.pdf, Xilinx, 2012.

[5] Xilinx, "Single Chip Crypto Lab Using PR/ISO Flow with the Virtex-5 Family", Xilinx Application Note XAPP1135 (v1.1.3), Available: http://www.xilinx.com/support/documentation/application_notes/xapp1135-scc-priso.pdf, 2013.

[6] J. D. Corbett, "The Xilinx Isolation Design Flow for Fault-Tolerant Systems", Xilinx White Paper WP412 (v1.1), Available: http://www.xilinx.com/support/documenta-

tion/white_papers/
wp412_IDF_for_Fault_Tolerant_Sys.pdf, 2013.

[7] E. Hallett, "Developing Secure and Reliable Single Device Designs with Xilinx 7 Series FPGAs or Zynq-7000 AP SoCs Using the Isolation Design Flow",Xilinx Application Note XAPP1086 (v1.2), Available: http://www.xilinx.com/support/documentation/application_notes/xapp1086-secure-single-fpga-using-7s-idf.pdf, 2013.

[8] M. McLean, J. Moore, "FPGA-based single chip cryptographic solution", Military Embedded Systems, 2007.

[9] T. Huffmire et. al., "Moats and Drawbridges: An Isolation Primitive for Reconfigurable Hardware Based Systems", In Proc of IEEE Symposium on Security and Privacy (SP'07), pp.281,295, 2007.

[10] C. Beckhoff, D. Koch, J. Torresen, "Go Ahead: A Partial Reconfiguration Framework," Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on , vol., no., pp.37,44, 2012.

[11] A. Wold; D. Koch; J. Torresen., "Component based design using constraint programming for module placement on FPGAs," Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC), 2013 8th International Workshop on , vol., no., pp.1,8, 10-12 July 2013.

[12] B. Nelson, M. Wirthlin, B. Hutchings, P.r Athanas, S. Bohner, Design productivity for configurable computing, ERSA, 2008. pp. 5766.

[13] E. L. Horta, J. W. Lockwood, S. T. Kofuji, "Using PARBIT to Implement Partial Run-Time Reconfigurable Systems", Lecture Notes in Computer Science Volume 2438, 2002, pp 182-191, 2002.

[14] H. Kalte, G. Lee, M. Porrmann, and U. Ruckert, "REPLICA: A Bitstream Manipulation Filter for Module Relocation in Partial Reconfigurable Systems", In Proceedings of the

19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 3 - Volume 04 (IPDPS '05), Vol. 4., Washington DC, USA, 2005.

[15] C. Lavin, M. Padilla, J. Lamprecht, P. Lundrigan, B. Nelson, B. Hutchings, "HMFlow: Accelerating FPGA Compilation with Hard Macros for Rapid Prototyping," Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on , pp.117,124, 1-3 May 2011.

[16] T. Frangieh, P. Athanas, A design assembly framework for FPGA back-end acceleration, Microprocess. Microsyst. 2014, http://dx.doi.org/10.1016/j.micpro.2014.02.008.

[17] A. Love, W. Zha, P. Athanas, "In pursuit of instant gratification for FPGA design", Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on, pp.1,8, 2-4 Sept. 2013.

[18] A. Otero, E. de la Torre, T. Riesgo, Dreams: A Tool for the design of Dynamically Reconfigurable Embedded and Modular Systems, In the Proceedings of the 2012 International Conference on ReConFigurable Computing and FPGA (RECONFIG2012), December 9-11, 2012.

[19] N. Steiner, A. Wood, H. Shojaei, J. Couch, P. Athanas, and M.French, "Torc: Towards an Open-Source Tool Flow," in Proceedings of the 19th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA 2011 (Monterey, California), February 27March 1, pages 4144, 2011.

[20] C. Beckhoff, D. Koch, J.Torreson, "Automatic Floorplanning and Interface Synthesis of Island Style Reconfigurable Systems with GoAhead", in Architecture of Computing Systems ARCS 2013, Prague, Czech Republic, Springer Berlin Heidelberg, 2013.

[21] C. Bolchini, A. Miele, and C. Sandionigi, "Automated Resource-Aware Floorplanning of Reconfigurable Areas in Partially-Reconfigurable FPGA Systems," in FPL, 2011, pp. 532538.

[22] K. Vipin, S. A. Fahmy, "Architecture-Aware Reconfiguration-centric Floorplanning for Partial Reconfiguration", In the Proceedings of the 8th International Conference on Reconfigurable Computing: Architectures, Tools and Applications (ARC'12), Hong Kong, China, Springer Berlin Heidelberg, 2012.

[23] M Rabozzi, J. Lillis, M. D. Santambrogio, "Floorplanning for Partially-Reconfigurable FPGA Systems via Mixed-Integer Linear Programming," Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on , vol., no., pp.186,193, 11-13 May 2014

[24] R. L. Marlow (2014), "Making Radios with GReasy: GNU Radio With FPGAs Made Easy," Master's Thesis, Virginia Polytechnic Institute and State University, Blacksburg, Virginia.