A Design Assembly Technique for FPGA Back-End Acceleration

Tannous Frangieh

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

Doctor of Philosophy
in
Computer Engineering

Peter M. Athanas, Chair
Carl B. Dietrich
Wu-Chun Feng
Brent E. Nelson
Patrick R. Schaumont

September 28, 2012
Blacksburg, Virginia

Keywords: Configurable Computing, FPGA Productivity,
Design Assembly Flow, Electronic Design Automation, Design Reuse

A Design Assembly Technique for FPGA Back-End Acceleration

Tannous Frangieh

ABSTRACT

Long wait times constitute a bottleneck limiting the number of compilation runs performed in a day, thus risking to restrict Field-Programmable Gate Array (FPGA) adaptation in modern computing platforms. This work presents an FPGA development paradigm that exploits logic variance and hierarchy as a means to increase FPGA productivity. The practical tasks of logic partitioning, placement and routing are examined and a resulting assembly framework, Quick Flow (qFlow), is implemented. Experiments show up to 10x speed-ups using the proposed paradigm compared to vendor tool flows.

Dedication

I lovingly dedicate this work to my father, mother, sister and two brothers.

Acknowledgments

Thanks to my academic adviser and mentor, Dr. Peter Athanas, for his unceasing support and guidance. Every time I needed guidance, I was in trouble, I needed help or advice, he was there for me.

Thanks to my committee for the insightful comments.

Thanks to Ali Sohanghpurwala for his trustworthy friendship. Ali made this journey much more enjoyable.

Thanks to Karina Kizjakina for her endless love, support and encouragement throughout the writing of this work.

Thanks to Dr. Edwin Robinson and his wife Valarie Robinson for their love. They treated me as their own son.

Thanks to Nataliya Timoshevskaya for being a great friend and for the valuable feedback on the work.

Thanks to Dr. Adolfo Recio, Charles Irick, David Uliana, Karl Pereira, Kavya Shagrithaya, Dr. Krzysztof Kepa, Prabhaav Bhardwaj, Richard Stroop, Ryan Marlow, Shaver Deyerle, Teresa Cervero, and many more great fellow students who became great friends during the

years in Blacksburg.

Thanks to Dr. Neil Steiner and Aaron Wood for their valuable feedback on the work.

Thanks to my parents and family for their endless love and support.

Glossary

API Application Program Interface

ASIC Application-Specific Integrated Circuit

BPR Block Place and Route

BRAM Block Random-Access Memory (RAM)

CLB Configuration Logic Block

CHREC Center for High-Performance Reconfigurable Computing

DSP Digital Signal Processor

EDIF Electronic Design Interchange Format

FPGA Field-Programmable Gate Array

FPT Fixed-Parameter Tractability

GPU Graphics Processing Unit

GUI Graphical User Interface

HDL Hardware Description Language

HLS High-Level Synthesis

HMFlow Hard Macro Flow

HPC High-Performance Computing

ISCAS International Symposium on Circuits and Systems

ISE Integrated Synthesis Environment

ITC International Test Conference

IWLS International Workshop on Logic and Synthesis

LUT Look-Up Table

MCNC Microelectronics Center of North Carolina

MDD Multi-Valued Decision Diagrams

NP-Hard Non-Deterministic Polynomial-Time Hard

PAR Place and Route

PATIS Partial module-producing, Automatic, Timing-aware, Incremental, Speculative

PR Partial Reconfiguration

qFlow Quick Flow

RAM Random-Access Memory

RAW Raw Architecture Workstation

RPM Relatively Placed Macro

RTL Register Transfer Language

SA Simulated Annealing

SoC System on Chip

TORC Tools for Open Reconfigurable Computing

VPR Versatile Placement and Routing

WoD Wires on Demand

XDL Xilinx Description Language

XUP Xilinx University Program

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

On March 11th 2011, a magnitude 8.9 earthquake hit Japan, with the epicenter tens of miles east of Tohoku. The earthquake triggered a powerful tsunami during which water waves reached up to 133 feet high, and traveled as deep as 6 miles inland [72]. The National Police Agency of Japan reported 15,841 deaths and 125,000 damaged buildings, and the tsunami caused the second largest nuclear disaster after Chernobyl, making the Tohoku earthquake the most devastating Japanese crisis since World War II and the most expensive natural disaster in the world with an estimated economic cost of $235 billion [34]. Among the different aid forms that were provided to Japan, compute hours were donated. Using supercomputers, computers capable of high computing speeds, Japanese scientists were able to accurately predict the tsunami wave heights and inland distance traveled. Unfortunately, it took about 12 hours to finish the total computation [35]. Had the computation been accomplished faster, thousands of life could have been saved and a nuclear disaster could

have been prevented. We live in an era where the ease and speed of computing has direct consequences on our daily lives.

Initially used to implement glue logic, FPGAs today are ubiquitous in automotive, data centers, high performance computers, medical, networking, security and asic prototyping, to name but a few [69, 6]. The speedup and modest power consumption of FPGA-based systems however, are often associated with long wait times before a device configuration is generated. Compared to contemporary alternatives such as microprocessors and Graphics Processing Units (GPUs), FPGA development tools can take hours if not days to generate a configuration, greatly hampering FPGA productivity and risking to limit device adaptation in modern computing platforms.

An FPGA design process consists of two complementary phases: a front-end phase and a back-end phase. A designer models and simulates their problems before hardware implementation, using tools built to describe hardware systems, i.e., the front-end tools. The design model is then synthesized into a structural netlist as the last stage of the front-end process. Today, there exist two classes of front-end tools: graphical (such as LabVIEW, Azido [8], Simulink [56], System Generator [9] and Digital Signal Processor (DSP) Builder [10]) and textual (such as VHDL and Catapult C [29]). Thereafter the back-end tools consume the resulting netlist and turn it into a device configuration. Vendor specific, the back-end tools solve difficult problems associated with transforming the netlist to a FPGA configuration. The algorithms for solving these problems typically do not scale well as the netlist size grows, making this a time intensive process. Due to their long run-times, back-end tools are the major contributor to the FPGA productivity problem. Altera PowerFit Fitter [5] and

Xilinx Place and Route (PAR) [68] are examples of back-end development tools. Table 1.1 highlights back-end compilation times for some FPGA designs.

Table 1.1: FPGA compilation times for sample designs

Design	Platform	Impl. Time (s)
Edge Detection	Xilinx ML 509	270
Gaussian Blur	Xilinx ML 509	350
ZigBee	Xilinx ML 509	441
1 Vector Addition	Convey HC-1	4084
3 Vector Additions	Convey HC-1	4530
Molecular Dynamics (GEM)	Convey HC-1	50400

1.2 Problem Statement

Irrespective of the long run-times, vendor tools excel in the fidelity of their results, while maximizing device utilization and clock speed. In the process of optimizing designs, the tools flatten carefully crafted hierarchical designs, and re-implement verified and implemented logic for every design iteration, incurring longer compilation times. For many FPGA designers, vendor tools serve the purpose of generating highly optimized designs that can fit on smaller, thus cheaper, devices. Yet for other FPGA applications where the speed of configuration is emphasized, FPGA back-end compilation constitutes a bottleneck limiting the number of iterations performed in a day [50]. The fact that back-end tools take hours or days to generate a configuration, shifts the designer focus from the computation itself, to irrelevant

recurring long waiting times, and jeopardizes their progress on solving problems. An analogy with the software world is useful to highlight the problem.

Consider the following hypothetical example.

> *Edwin is a professor in geology. He uses a Linux workstation to build a gravity model using a high-level language such as C++. Every time he modifies the source code of his model, he compiles the code and run the resulting model. When using complex datasets, it takes Edwin less than half a minute to compile and generate gravity response numbers. If we assume that he repeats this process every 10 minutes, over an 8-hour working day Edwin can have a total of* $8\ hours/10.5\ minutes \approx 45$ *compilations/day. Halfway through his work, an update to the C++ compiler forced a developer to compile all libraries called in their program, every time the program itself is compiled. Unlucky Edwin uses in his model many library calls, including system or kernel calls, and now it takes him roughly two hours to compile and generate gravity response numbers. Suddenly, he dropped from 45 compilations/day to roughly 4 compilations/day only!*

Today, a scientist working on FPGA development faces the same problem that Edwin had after his compiler's update. Any change to the computing logic triggers a sequence of unnecessary and time consuming implementation jobs before a configuration is generated. If FPGAs were ever to become a trend in computing platforms, it is crucial to decrease the time to generate a configuration.

1.3 Proposed Work

Traditionally, FPGA development flows implemented a design while optimizing for speed and area, and ignoring any delays in configuration generation. This work presents a way of viewing FPGA development, in which FPGAs are treated as flexible hardware capable of quick adaptations to computational needs, not only as reconfigurable devices operating optimized high-speed designs. In this paradigm, a design is split into two complimentary sets based on logic variance: a mostly invariant interfacing logic set and an evolving logic set. Compared to a holistic design implementation by current FPGA development methodologies, the work implements each of the two sets separately. As the name indicates, the invariant set is less likely to change over the design process, thus less frequently re-implemented. The evolving set is oftentimes re-implemented during design iterations. Through separation of concerns and re-use, the approach speeds up the implementation process and put the designer's focus on the creative part (algorithms and computations) of the design process.

Let's return back to the software analogy, the evolving set maps to Edwin's modeling program and the invariant set maps to the libraries called by that program. Over the period of Edwin's work, one can assume that called libraries are invariant and rarely require any compilation. Edwin's program on the other hand is being constantly updated, calling for frequent compilations. By linking to the already compiled libraries, instead of re-compiling them for each iteration, Edwin went from 4 to 45 compilations/day, achieving an order of magnitude higher. Moreover, he was able to focus on what is of most relevance to his work by spending his time developing the gravity model and the corresponding algorithm.

1.4 Why Does This Matter?

Decreasing FPGA compilation time has favorable implications on the configurable computing development process. A faster compilation process increases the number of compilations/day, therefore decreasing the time and money spent on a project. Apart from saving time and money, a fast compilation offers unprecedented capabilities in engineering systems such as autonomous, reliability and debugging systems. Consider the following analogy. Hypothetically speaking, if we were to drastically speed up the learning process of a human brain, mastering the Chinese language, grasping Einstein's Theory of Relativity, and learning how to fly a fighter jet could all turn into an afternoon exercise. FPGAs are but small digital brains. As their compilation time (or digital learning process duration) approaches zero, endless possibilities in science and engineering are presented; An autonomous system capabilities won't be limited to pre-existing hardware configurations, a reliability system will quickly adapt to unfamiliar scenarios preventing disasters and potentially saving lives, and signal probing boils down to a mouse hover in a Graphical User Interface (GUI), to name but a few.

1.5 Research Contributions

The realization of this work produced the following contributions:

1. A proposed paradigm for FPGA development and design assembly. A clear contribution is the partitioning of a design and the application of different implementation approaches based on logic variance (Chapter 4).

2. An implementation framework for FPGA designs based on the concepts from the proposed design assembly paradigm (Chapter 5).

3. An evaluation of the proposed design assembly in light of the developed framework and a comparison against current FPGA development models and vendor flows, using a set of benchmark designs covering a multitude of design domains (Chapter 6).

The findings of this research are also applied to the Center for High-Performance Reconfigurable Computing (CHREC)-funded projects "V2-11: Back-end Productivity Enhancements for High-Performance Computing (HPC)" and "V2-12: End-to-End Tool Flow for FPGA Productivity". The projects intend to provide a familiar environment that aids in the productivity of FPGA design development, thus enabling members to improve the productivity of their design process.

1.6 Organization of the Document

The rest of the dissertation is organized as follows. Chapter 2 identifies theoretical limits when solving Non-Deterministic Polynomial-Time Hard (NP-Hard) problems. The FPGA design flow and use models along with literature review are summarized in Chapter 3. Chapter 4 details the contributed approach. Chapter 5 highlight algorithms and implementation details. Benchmarks and performance analysis are presented in Chapter 6. Conclusions and future work are discussed in Chapter 7.

Chapter 2

Incremental Design for Planar Logic

The circuit design process begins with a circuit specification and through several transformations turns the specification into a two-dimensional physical implementation. The circuit specification is first turned into a netlist of logic components. The logic components are then mapped to geometric shapes that are placed on a two-dimensional physical plane and connected together to form the physical circuit representation.

The process of implementing a design for a contemporary FPGA requires the specification of millions of switches to be set. A designer usually models a circuit using a Hardware Description Language (HDL). The HDL model then goes through a number of transformations before a final configuration is generated. The transformation from a circuit model to a configuration is an automated process, and a complex problem that consists of solving difficult sub-problems. Algorithms to solve such problems are usually computationally expensive, yielding to long times to generate a configuration from a circuit model.

This chapter examines the compilation process in the light of identifying aspects that will minimize the computational load for incremental compilation. A model of a netlist is presented, and the back-end processing, specifically place and route, is described in terms of operations on the netlist model. The chapter highlights offline and run-time processing techniques for the placement and routing problems, and concludes with a discussion on the run-time complexity for the two problems in light of the aforementioned techniques.

2.1 FPGA Flows: A Closer Look

The FPGA flow is divided into seven stages: Design entry and logic synthesis, technology mapping, mapping optimization including logic folding and fanout optimization, primitive packing, primitive block placement, routing, and physical bitstream generation. During design entry and logic synthesis, a circuit model is consumed and transformed into a netlist of logic gates. Technology-independent logic optimization is applied to the netlist during this phase. The resulting netlist is technology mapped, i.e., represented using FPGA device primitives. Mapping optimization is applied to the mapped netlist to reduce the circuit size and delay. The optimized netlist of primitives is then packed into primitive blocks that are placed on a device and routed. The placed and routed netlist is finally turned into a device configuration bitstream. Figure 2.1 presents the different stages of a FPGA flow. Since the emphasis of the work is on back-end acceleration, in particular place and route stages, all other stages are mentioned only for completeness, and will be mostly ignored for the rest of this chapter.

Place and route stages are governed by computationally demanding algorithms. The run-time complexity of such algorithms grows exponentially in the problem size. For such problems, addressing the question of how to decrease the run-time of the computation reduces to addressing the question of how to reduce the problem size itself at run-time, thus the two questions:

1. For placement, how can we decrease the number of blocks primitives to be placed at run-time?

2. For routing, how can we reduce the number of nets to be routed at run-time?

Before addressing the two questions, a netlist model needs to be presented.

2.2 Netlist Model

A netlist of blocks primitives (or gates) can be represented as a directed graph $G(V, E)$, where the set of vertices V are the blocks primitives in the netlist and the set of edges E makes the nets connecting the blocks. Figure 2.2 is a directed graph representation of a small netlist where the set of blocks primitives is $V = \{LB1, LB2, LB3, LB4, LB5, LB6, LB7, LB8\}$, and the set of nets is $E = \{n1, n2, n3, n4, n5, n6, n7, n8, n9, n10, n11, n12\}$.

A successful placement computes a legal placement for every primitive block in the graph, with an exponential run-time complexity in the graph order, i.e., the number of vertices $|V|$ [44] [61] [52]. A successful routing finds a wire for every net connecting a source to a sink (or sinks) block, with an exponential run-time complexity in the graph size, i.e., the number

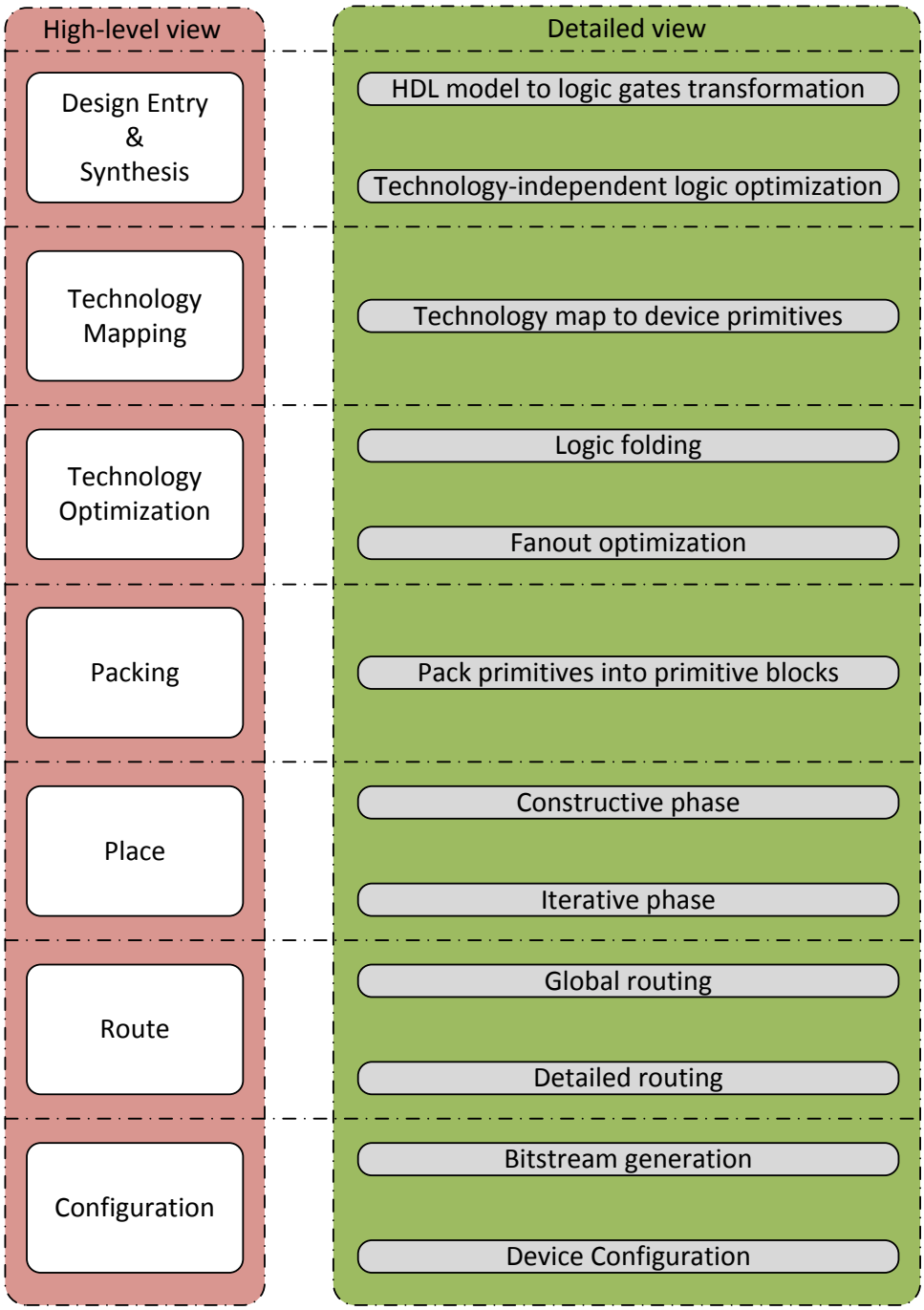


Figure 2.1: FPGA tools flow

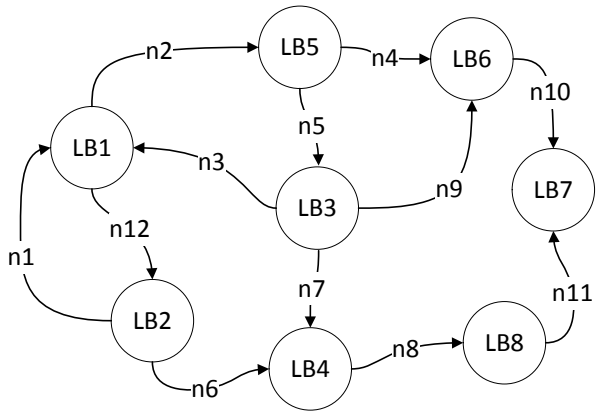


Figure 2.2: A graph representing a netlist of blocks primitives

of edges $|E|$ [67]. Thus, a decrease in the graph order and size yields a faster placement and routing.

2.3 Offline & Run-time Processing

All design relies on some sort of hierarchical description and FPGA design is no exception. In a hierarchical design, a design is described in terms of logical groupings, which in turn are described in logical sub-groupings and so on. In FPGA design, we call a logical grouping a module, and the use of a module at any level of the hierarchy an instance of that module.

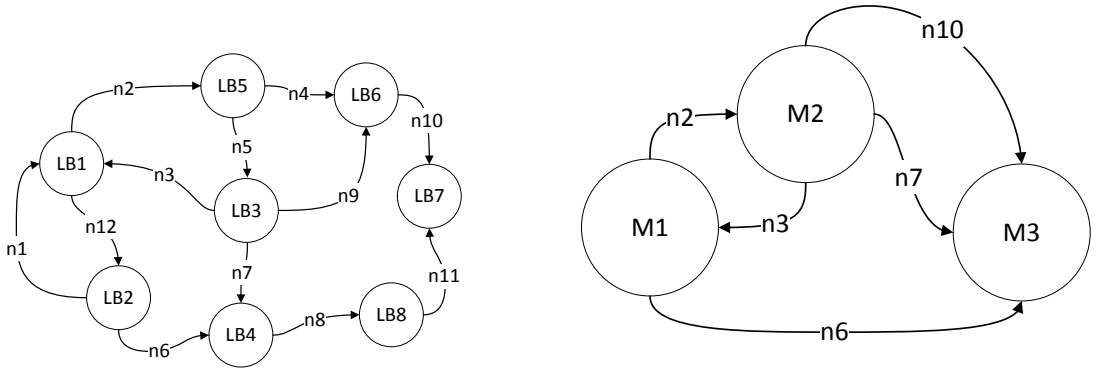
Hierarchy in FPGA designs tends to be functional, i.e., grouping logic performing a specific function. The same functional grouping can be viewed as a separation of concern mechanism, stating that given a level of hierarchy with any two modules A and B, the implementation process of A is independent of that of B. The aforementioned reasoning suggests an incremental approach to implementation; during a design implementation, every module can be

implemented independently of others, i.e. offline processed, then the final solution would be attained by stitching the different modules at run-time. Table 2.1 shows how the different design flow stages can be modified to incrementally implement a design.

Table 2.1: Preprocessing and run-time processing for an FPGA design flow

Stage	**Offline Processing**	**Run-time Processing**
Synthesis	Modules synthesis	Synthesized modules stitching
Mapping	Modules mapping	Mapped modules stitching
Mapping Optimization	Mapped optimization	Optimized modules stitching
Packing	Intra-module packing	Packed modules stitching
Place	Intra-module block placement	Coarse module placement
Route	Intra-module routing	Inter-module routing
Configuration Generation	Module configuration generation	Modules configuration stitching

An incremental approach to implementation can be explained as a clustering operation on the graph representation of a netlist. Figure 2.3 shows a graph representation of a circuit and the corresponding hierarchy-based clustered representation. In the clustered representation, $M1 = \{LB1, LB2, n1, n12\}$, $M2 = \{LB3, LB5, LB6, n4, n5, n9\}$, and $M3 = \{LB4, LB7, LB8, n8, n11\}$. From a placement perspective, if $M1$, $M2$ and $M3$ are already solved, i.e. their blocks primitives placed, a solution to the netlist consists of placing three blocks instead of eight, as in the original representation. The same reasoning applies for routing. If $M1$, $M2$ and $M3$ are solved, i.e. their blocks primitives routed, the input to the router is reduced from twelve to five.



(a) A graph representation of a circuit

(b) A hierarchy-based clustering of the circuit representation

Figure 2.3: A graph representation of a circuit and corresponding clustered version

2.4 Approach

Placement and routing belong to the NP-Hard class of problems [27] [33]. NP-Hard problems are problems whose solution can be verified in a polynomial time, but it takes a polynomial time to generate one on a non-deterministic machine. An exact solution to such problems is computationally demanding, and oftentimes a solution is approximated using *heuristics*, techniques that use rule of thumb to find a good enough solution for the problem in consideration. Two basic strategies exist for heuristics: a divide-and-conquer and an iterative approach [36]. In a divide-and-conquer approach, a problem is further smaller problem of similar or related type, until the small problems are easy enough to be solved. The solutions to the sub-problems are then stitched together to form the final solution to the original problem. An iterative approach on the other hand, starts from a current solution to a problem and iteratively tries to converge to an optimal or near optimal solution by

exploring other neighboring solutions.

The proposed work reduces the input size to placement and routing by following a hybrid approach: a divide-and-conquer step divides the problem into smaller problems, guided by the design hierarchy. The small problems are solved using an iterative approach. This step corresponds to the Offline Processing column in Table 2.1. The last step of stitching of the final solution is solved at run-time using an iterative approach. This step corresponds to the run-time Processing column in Table 2.1.

While applying heuristics is one way of approaching NP-Hard problems, another way of coping with such problems is by investigating the association of instances of such problems with parameters that can affect their complexity. When a parameter of a problem instance is small, an efficient algorithm (polynomial time) may solve the problem instance, making the problem instance tractable. Such technique is referred to as Fixed-Parameter Tractability (FPT) and is investigated by Downian et. al. in [23]. Average case analysis, approximation and randomization are yet other approaches for dealing with NP-Hard problems.

For example, in [60], Sulimma et al. show that grid based channel routing is a fixed-parameter tractable problem. By fixing the channel width (number of tracks), the authors were able to find a linear time solution for the channel routing problem, using Multi-Valued Decision Diagramss (MDDs) [32].

2.5 Run-time Complexity

In [44], Marquardt et al. present a simulated annealing based timing-driven placement algorithm for FPGAs, with an average complexity of $O(n^{4/3})$ in the input size n. Further analysis in this section will be based on that complexity model.

Assuming a hierarchical design of size S with M modules ($M \ll S$), and s_i the size of module m_i ($M < s_1 \leq s_2 \leq s_3 \ldots \leq s_M$), the run-time complexity of placing the design is in the order of Equation 2.1

$$O(S^{4/3}) \tag{2.1}$$

The run-time complexity of incrementally placing the design is in the order of Equation 2.2

$$O(\sum_{i=1}^{M} s_i^{4/3}) = O(s_1^{4/3} + s_2^{4/3} + s_3^{4/3} + \ldots + s_M^{4/3}), \tag{2.2}$$

Using Newton's binomial theorem,

$$s_1^{4/3} + s_2^{4/3} + s_3^{4/3} + \ldots + s_M^{4/3} < S^{4/3} \tag{2.3}$$

Therefore,

$$O(s_1^{4/3} + s_2^{4/3} + s_3^{4/3} + \ldots + s_M^{4/3}) < O(S^{4/3}) \tag{2.4}$$

Equation 2.4 states that an incremental placement of a design yields faster run-time.

Offline processing highlighted in Table 2.1 has also an effect on reducing placement run-time. In fact, offline processing of M modules further reduces the placement problem run-time order from

$$O(s_1^{4/3} + s_2^{4/3} + s_3^{4/3} + \ldots + s_M^{4/3}) \tag{2.5}$$

to

$$O(M^{4/3}) \tag{2.6}$$

where

$$O(M^{4/3}) < O(s_1^{4/3} + s_2^{4/3} + s_3^{4/3} + \ldots + s_M^{4/3}) \tag{2.7}$$

Therefore, with hierarchical decomposition and offline processing, the run-time complexity of placing a design of size S with M modules ($M \ll S$) can be reduced from $O(S^{4/3})$ to $O(M^{4/3})$, yielding faster placement. The same reasoning can be applied to routing, therefore won't be repeated in this work.

Chapter 3

Background & Related Work

This chapter starts by putting the work in perspective. It then summarizes the different stages of the FPGA design process and draws a high-level view of the similarities and differences between ASIC and FPGA design flows. The chapter concludes by surveying related work on FPGA back-end acceleration.

3.1 It's All About Perspective

Transforming a circuit representation to a FPGA device configuration is a lengthy process, and typical FPGA implementation tools take tens of minutes, if not hours and days to finish the job [49]. Such implementation tools can be perceived from two different angles: the ASIC engineer angle or the software engineer angle. Based on their angle of perception, an FPGA designer will have a different set of expectations.

The ASIC fabrication process is a lengthy process. The process of progressing from a circuit concept to a physical implementation is unforgiving to design errors and it takes months before a hardware engineer can verify the design on hardware. An off-the-shelf FPGA capable of emulating a design in hours, even days, comes handy for design testing and verification. No matter how long the FPGA implementation tools take to converge to a solution, they are still way faster and forgiving compared to the rigid and lengthy ASIC process.

From a software engineer point of view, it's a completely different story. Software engineers are used to build times in the order of seconds, and tens of compilations per day are expected for anyone developing software. FPGA tools suffer to keep up in such environments due to their long compilation times. It is from the *software engineering angle* that this work is to be perceived.

3.2 FPGA Design Flow

A typical FPGA design flow consists of three main steps: design entry and synthesis, design implementation, and design verification. Using design entry tools and/or languages, a designer models the problem at hand. The earliest of the design verification stages, functional simulation, is performed during design entry. The model is verified for expected operation, then synthesized into a structural netlist. During the implementation step, the structural netlist is then mapped to the underlying hardware resources or primitives, placed and routed. Two design verification tests can take place post-place and route: timing simulation and static timing analysis. Static timing analysis is best for timing checks of a design after it

is placed and routed, and determines path delays in the design. Timing simulation on the other hand, verifies that the mapped, placed and routed design runs at the desired speed for the device. Finally, the configuration bitstream is generated and downloaded to the actual device, where the design can be verified for desired operation. Details about the three flow steps follow.

3.2.1 Design Entry & Synthesis

During design entry, a designer models and simulates their problems before hardware implementation, using tools built to describe hardware systems, i.e., the front-end tools. Today, there exist two classes of front-end tools: textual and graphical tools.

Textual Tools Textual front-end tools in turn can be split into two sub-classes: HDL and High-Level Synthesis (HLS) tools. A HDL is a modeling language used to describe a digital circuit at the structural level, i.e., the collection of interconnected gates performing a computation, the behavioral level, i.e., a description of how the design behaves, or a combination of both. The design model has the ability of being simulated for functional correctness. JHDL [13], Verilog and VHDL are all examples of HDLs.

HLS is a process during which a structural view of a circuit is generated out of its architectural model [47]. The *architectural model* reveals a circuit as a set of operations, such as data computation or transfer, whereas the *structural view* presents the circuit as an interconnection of components. The architectural model is usually specified in C/C++/SystemC and the resulting structural view is specified in Register Transfer Language (RTL) Verilog or VHDL.

Graphical Tools Graphical modeling is yet another method of describing a digital circuit. In a graphical environment, a designer drags and drops components on a computer screen to model a design. Components usually consist of gates, thus, a structural model, or the description of a computation with a well defined interface, thus a behavioral model, or a mix of the two. An integrated simulator simulates the behavior of the model for functional correctness before generating an HDL representation of the model. DataIO Azido [8] and National Instruments LabVIEW FPGA are two examples of graphical modeling tools.

Synthesis Synthesis, or logic synthesis, transforms the HDL model generated by the design entry tools into a structural netlist of logic gates. The resulting netlist can be optimized for speed, area, and/or power.

3.2.2 Design Implementation

During the design implementation phase, the technology mapped netlist is placed and routed against the FPGA architecture at hand. Details about each of the three processes follow.

Technology Mapping The structural netlist is first technology mapped to the device. During mapping, the netlist is represented as a function of the underlying device primitives, versus a netlist of generic boolean gates. The mapped netlist also undergoes a series of optimization, including logic folding and fanout optimization.

Placement Given a netlist of primitives, a placer finds a suitable location for each element in the netlist while addressing optimization objectives, such as the total wirelength, number

of cut nets, wire congestion and signals delays. Placement belongs to the NP-Hard class of problems, whose solution can be verified in a polynomial time but it takes a polynomial time to generate one on a non-deterministic machine. Placement algorithms are computationally expensive, and placement problems are often solved using heuristics. Placement has significant impact on the final solution, and a bad placement will significantly hamper design speed and/or routability.

Existing placement algorithms can be classified into two classes: constructive algorithms and iterative algorithms. Constructive placement algorithms start with a completely unplaced netlist and constructively build a solution. Partitioning-based placement is an example of constructive placement [43]. Iterative placement algorithms begin with an initial netlist placement and seek to incrementally improve the current solution by exploring neighboring solutions. Simulated Annealing (SA) [36] is an iterative heuristic commonly used in vendor and research tools placement such as Xilinx Placer, Altera Placer and the Versatile Placement and Routing (VPR) package. When placing designs, a hybrid placement approach is oftentimes used; an iterative algorithm improves an initial solution generated by a constructive placement algorithm.

Routing Given a placed netlist, a router connects all elements in a placed design, guided by the connectivity information of that placed netlist, while also addressing optimization objectives, such as the total wirelength and critical signals delays. Connecting two elements is done by computing the list of FPGA wire segments from the source pin to the destination pin. Similar to placement, routing is yet another NP-Hard hard problem, therefore heuristics are also used to solve routing problems.

Routing is usually performed in two stages: a global routing stage followed by a detailed routed stage. During the global routing stage, each net of the design is assigned a set of channels to go through, while minimizing objectives such as routing congestion and satisfying timing constraints of critical nets. PathFinder [46] is an example of FPGA global router. Each net is then assigned a unique set of routing segments during the detailed routing stage, such that there is no resource conflict given any two nets. SEGA [40] is an example of a detailed router. Oftentimes a combined global and detailed routing approach is sought during routing for a better quality routing, with the VPR router being an example router that combines both routing techniques. For an extensive survey on FPGA design automation, including placement and routing algorithms, the reader is referred to [17].

3.2.3 Design Verification

Four verification methods are generally used in FPGA design flows: behavioral simulation, static timing analysis, timing simulation, and in-device verification. Details about each of the methods follow.

Behavioral Simulation Behavioral simulation is performed on the design model generated by the design entry tools, and before the model is synthesized into a structural netlist. During behavioral simulation, the design functionality is verified without any specifics about the implementation. Running at a high level of abstraction, behavioral simulation is design timing agnostic and runs the fastest compared to timing simulation. Most of the bugs in the design are detected and fixed during behavioral simulation.

Static Timing Analysis Static timing analysis is performed on a placed and routed design at the end of implementation phase. The implemented design is checked against timing violations such as hold time and setup time violations, incurred by designer specified design constraints.

Timing Simulation Once a design is placed and routed, logic and routing delays can be accurately estimated based on the device speed grade. Delay values are then used to build a timing-accurate HDL model, on which a timing simulation is performed. Timing simulation is the closest a simulation model can get to an actual design running on the device, however it has the slowest runtime when compared to higher level simulation methods.

In-Device Verification After a configuration bitstream is generated, the design can be tested on the device itself. Verification at this stage is done through monitoring of the device I/O signals using external testing and measuring equipment, internal device probing through an integrated analyzer, or a combination of both. In-device verification is the fastest and most robust among verification methods.

3.3 FPGA versus ASIC Design Flows

Configurable computing tools have always been considered as hardware development tools. In fact, almost all stages in the FPGA design flow have matches in the ASIC design flow. An ASIC flow also incorporates a design entry stage, synthesis, behavioral simulation, place and route, static timing analysis, timing simulation and in-circuit testing. A unique aspect of the ASIC design flow is the extra testing and verification steps during post-synthesis and

post-place and route, where techniques such as equivalence checking and second and third order effects verification are exercised. Figure 3.6 shows the different steps in both FPGA and ASIC design flows.

ASIC tools work to transform a functional description of a design into circuits with maximum performance and minimal area, disregarding the time it takes to converge to a solution. FPGA tools however, are solving the same problem but are not bounded by the same set of constraints as ASIC tools all the time. For low volume applications where cost does not justify an ASIC, an FPGA is used as an ASIC replacement. FPGAs are also leveraged for rapid prototyping, adaptive systems, in-system customization, and multi-model computation [1], and a one-size-fit-all design flow is not the right answer. In a rapid prototyping environment for example, a larger than needed device is assumed and design area is of less priority when compared to compilation speed [1]. In multi-model computation and adaptive systems, compilation speed and flexible design assembly are of high importance and desirable.

During design implementation, FPGA tools strive to satisfy three conflicting constraints: high clock speed, high density, and fast assembly. The high clock speed constraint pushes the tools to increase the running design clock speed, whereas the high density constraint guides the tools into fitting more logic given a device. The fast assembly constraint compels the tools into lower compilation times. In an ideal development environment, the tools do not compromise and optimize for all three constraints equally. Unfortunately, that is not the case and the tools oftentimes sacrifice a constraint when optimizing for another, thus the weight of each of the three constraints is use-model dependent. Figures 3.1, 3.2, 3.3, 3.4,

[1]Throughout this work, compilation speed and assembly speed are used interchangeably

and 3.5 illustrate a weight combination of all three aforementioned constraints per FPGA use model, using a three-value scale: very important, important, and less important.

The five radar plots convey a clear picture about FPGA tools requirements. As shown in Figures 3.2, 3.3 and 3.5, the stringent area and performance constraints are relaxed and the emphasis is on speed of assembly. Current tools do excel in addressing the stringent area and performance requirement with little effort done in compilation speed, which is the focus of this work in general, and the acceleration of the back-end process in particular.

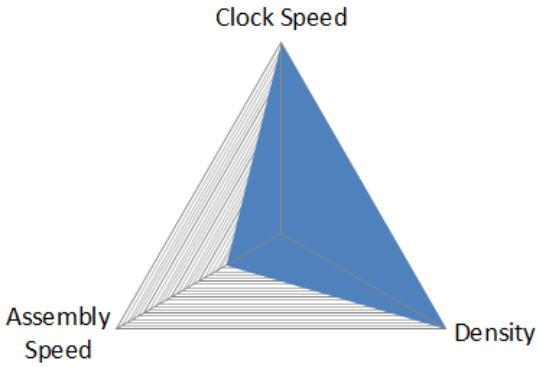


Figure 3.1: ASIC replacement FPGA use model

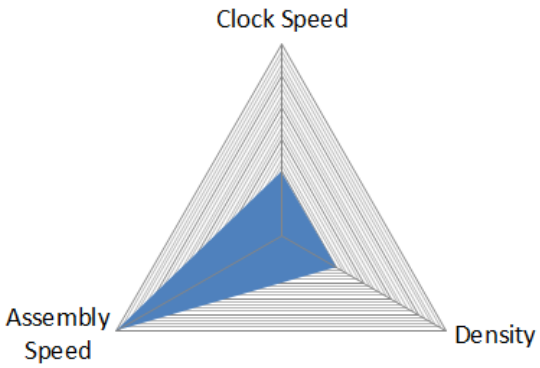


Figure 3.2: Rapid prototyping FPGA use model

3.4 Related Work

Back-end acceleration efforts can be broadly classified into two efforts: *speeding up place and route* and *reducing the size of placed and routed circuit*. Speeding up place and route can be achieved either by accepting a lower quality solution, or by parallelizing placement and routing algorithms and leveraging multi-core machines. Reducing the size of placed and routed circuit is oftentimes done through partitioning and/or re-use. Coarsening the granularity of the circuit primitives is yet another way of reducing the size placed and routed circuits.

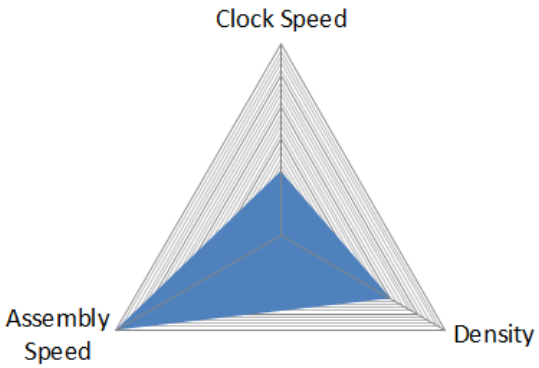


Figure 3.3: Adaptive systems FPGA use model

3.4.1 Speeding up Place and Route

The work in [54], [71], [62] and [42] speeds up placement and/or routing by leveraging algorithms and techniques that yield a lower solution quality. Parallelizing placement and routing algorithms is another approach to decrease place and route times [16]. A summary of the work follows.

Sankar et al. [54] implement a multi-level placement algorithm. A netlist of logic blocks is first clustered bottom-up based on the blocks connectivity. A constructive placement stage is then performed at each level of the hierarchy, followed by a SA iterative stage to refine placement. The work is parametrizable and allow the designer to control placement run-time versus quality of placement.

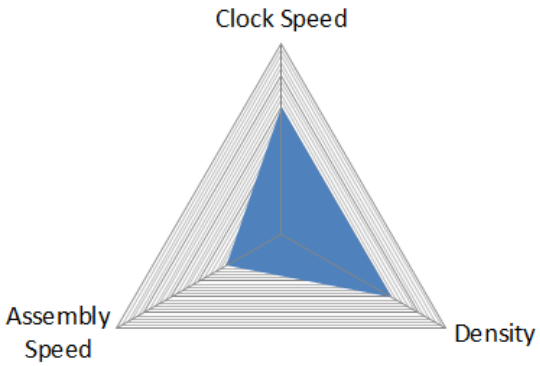


Figure 3.4: In-system customization FPGA use model

The authors of [71] present a quadratic placement tool for FPGAs. The tool incorporates three stages. During stage one, the tool builds and solves linear equations aiming to obtain a good initial placement. The second stage applies a quick placement refinement through node shuffling. A final refinement stage is done by applying a low temperature SA.

Swartz et al. [62] detail a routability-driven router for FPGAs. The router is capable of quickly identifying difficult or impossible routing problems, and quickly routing "low stress" routing designs, defined as designs where a routing channel has at least 10% more tracks than the minimum number of tracks needed to route a design. The routing algorithm is based on the PathFinder algorithm [46] with two speed enhancements: a depth-first search that directs the router faster towards a target, and a reduced routing expansion list for high-fanout nets.

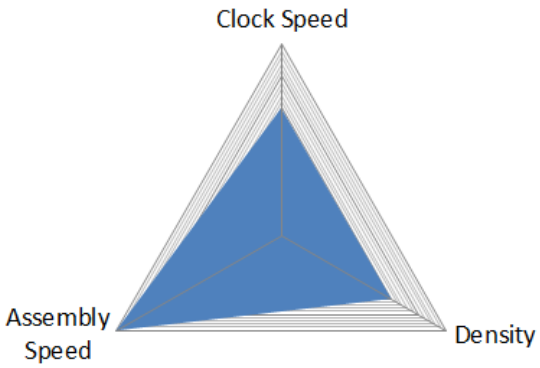


Figure 3.5: Multi-model computation FPGA use model

The authors of [42] presents Riverside On-Chip Router, a fast router for their JIT FPGA compiler. The router is based on VPR routability-driven router. The work assumes a simplified FPGA configurable logic fabric, and uses a smaller routing resource graph compared to that of VPR. The router is intended to run on processor running on the FPGA itself, thus the authors strive to speed-up the routing algorithm and minimize the corresponding memory usage.

3.4.2 Reduce Place and Route Circuit Size

The work in [64], [37], [21] and [7] incorporates re-use to reduce the amount of circuit to be placed and routed at run-time. Previously implemented circuits are cached as macros and re-used in future iterations of a design. Partial reconfiguration is another method of

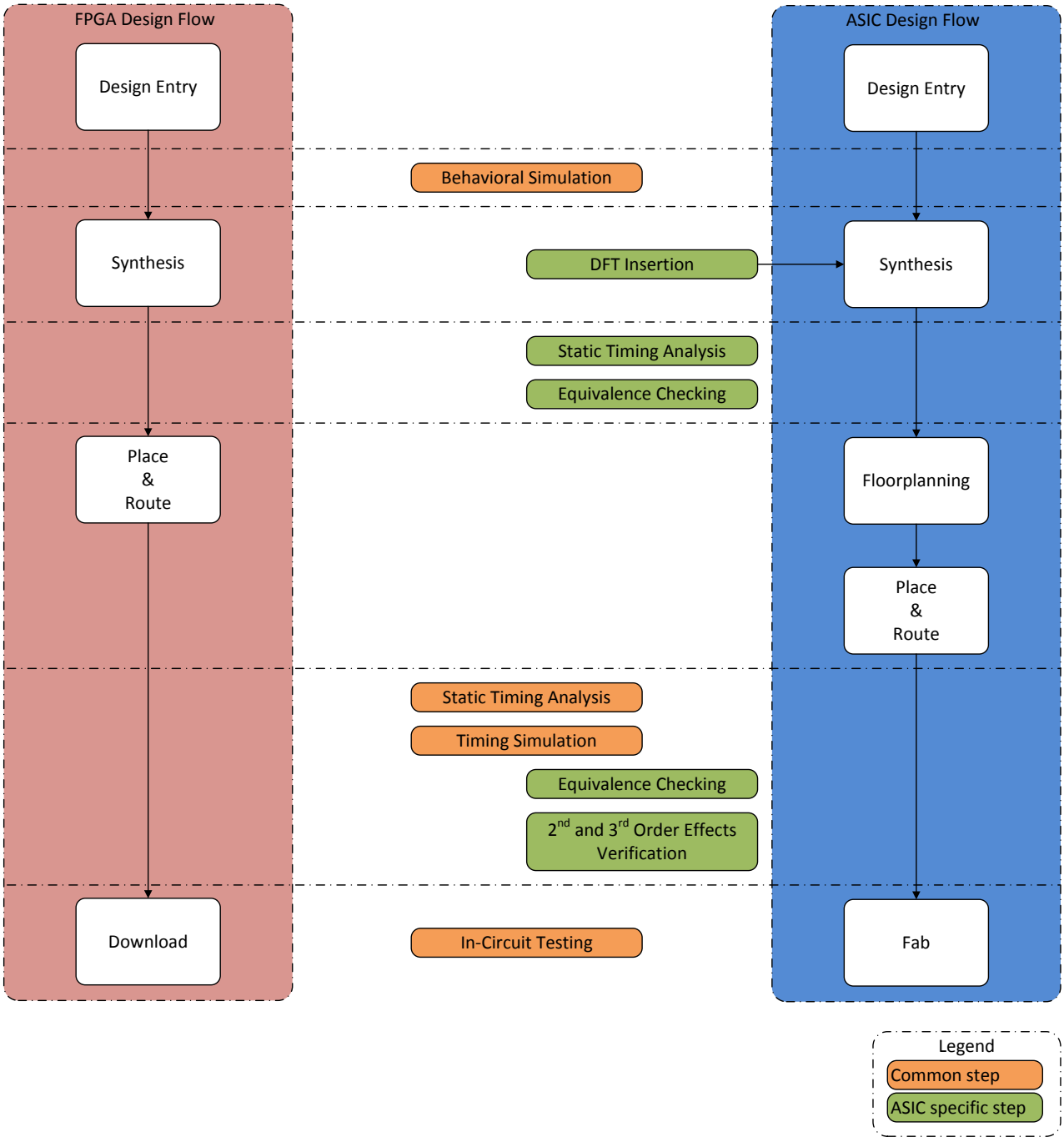


Figure 3.6: A high level view of FPGA and ASIC design flows

reducing place and route circuit size [57], [26]. A summary of the work follows.

The author in [64] presents *Frontier*, an FPGA placement tool that leverages floorplanning and precompiled soft and hard macros to speed up placement. An initial floorplanning stage places the macro blocks, followed by a low temperature SA placement refinement.

RapidSmith is a Java-based software library to manipulate XDL designs developed at Brigham Young University's Configurable Computing Laboratory [38]. Developed in the same lab, Hard Macro Flow (HMFlow) is an FPGA design flow based on hard macros that builds on RapidSmith [37]. The flow implements a custom mapper, placer, and router and leverages hard macros and caching to speed up the FPGA compilation process. HMFlow supports the Xilinx Virtex-4 and Virtex-5 architectures.

Coole et al. [21] introduce Block Place and Route (BPR), a tool to speed up the back-end process through re-use. Placement and routing of design cores are precomputed and stored in the form of relocated macro blocks. An initial placement relocates the macro blocks. The resulting placement is refined through a SA based iterations. The inter-blocks connectivity is then achieved through a PathFinder-based router with A* search.

Wires on Demand (WoD) [7] is a run-time reconfiguration framework for Xilinx FPGAs. The framework allocates a sandbox on the device where precompiled modules fetched from a library can be placed and routed at run-time. A software layer abstracts the low level reconfiguration details away from the designer.

OpenPR is an open-source, slot-based, partial-reconfiguration toolkit for Xilinx FPGAs [57]. The toolkit enables partial modifications on a design by loading partial configuration(s), while other parts of the design remain intact.

The authors of [26] presents a Partial module-producing, Automatic, Timing-aware, Incremental, Speculative (PATIS) floorplanner that leverages partial reconfiguration to improve implementation and debug turnaround. The tools incorporates incremental and speculative modes lead to a faster design convergence and configuration generation.

3.4.3 Reduce Place and Route Circuit Size: Domain-Specific Architectures

Implementing for domain specific architectures is another method of speeding-up the implementation process. The efforts in [55], [12] and [20] fall into this category, where a coarse-grain domain specific architecture is overlayed on a FPGA. The implementation problem is transformed into mapping the problem at hand to the intermediate architecture.

Two major weaknesses govern most of the work listed in sections 3.4.1, 3.4.2 and 3.4.3: architecture dependence and limited application scope. An architecture dependent implementation flow is oftentimes optimized for that architecture yielding considerable amount of speedups for that architecture at the expense of porting to different architectures. The work in [21], [7] and [57] is device architecture dependent. [26] depends on partial reconfiguration and thus also architecture dependent. The work in [54], [71], [62] and [42] is application scope limited, due to the lower solution quality. For example, HPC applications are excluded in

such case. A domain specific solution is also scope limited by design [55] [12] [20].

Chapter 4

Assembly Paradigm

This chapter proposes an assembly paradigm for FPGA development. It starts by a high-level description of the assembly process. The process is then broken down into its four constituents stages, which in turn are described in detail.

4.1 Overview

FPGA implementation tools strive to generate optimal designs with (1) increased design clock speed, (2) and maximized device utilization. During the process, complex optimization techniques are applied yielding high-quality results that can take hours or even days to generate. While such compilation times are acceptable for an ASIC engineer, they come at a high-price for the configurable computing community, where software-like compile times are desirable [30].

A closer look at an FPGA design reveals an interesting property. Independent of the application, a design can be seen as the combination of two complementary logic sets: an evolving logic set interacting with the outside world through a mostly invariant logic set. A memory interface and a Gigabit Ethernet interface are two typical examples of mostly invariant logic. Moreover, as the design goes through development iterations, more of the evolving logic stabilizes and becomes less likely to change. A verified module is a good example of invariant logic that evolved as the design matures. A migration in the opposite direction is yet another possibility. Updating parameters of a verified module takes this module back to evolving logic land.

4.2 A Proposed Assembly Paradigm

This work proposes a design assembly paradigm for FPGA development that exploits logical hierarchy and variance. *Logic variance* expresses the likelihood of a logic to change over a design lifetime; a variant logic is more likely to be updated whereas an invariant logic will mostly remain static. The proposed paradigm partitions a design into two classes, an invariant set and an evolving set, incorporating different implementation techniques and frequency for the two classes. Following the proposed partitioning, the invariant set is less likely to change, thus less frequently re-implemented, whereas the evolving set is oftentimes updated and re-implemented. Through the application of incremental implementation techniques, the work strives to reduce the run-time overhead of the implementation tools, for a quicker convergence to a solution [18] [24] [53] [15] [19] [41].

The proposed flow addresses the two major weaknesses of existing flows through device architecture independence and flexible logic implementation. Contrary to current tool flows that follow a holistic approach when implementing a design, the proposed technique divides the implementation process into four phases:

1. Design Partitioning
2. Invariant Set Implementation
3. Evolving Set Implementation
4. Design Assembly

Figure 4.1 presents a high-level view of the assembly paradigm. The assembly paradigm follows an incremental approach that starts by partitioning the design into two sets based on logic variance. The two logic sets, variant and invariant, are separately implemented then incrementally stitched together to produce the final solution. Such incremental approach provides the ability to localize design updates to the corresponding modules and implement different parts of the same design with different optimization goals. Details about the different phases of the paradigm follow.

4.3 A Four-Phase Implementation Process

4.3.1 Design Partitioning Phase

The first step in the implementation process is the design partitioning phase. During this phase, a design is partitioned into invariant and evolving logic. Logic can move back and

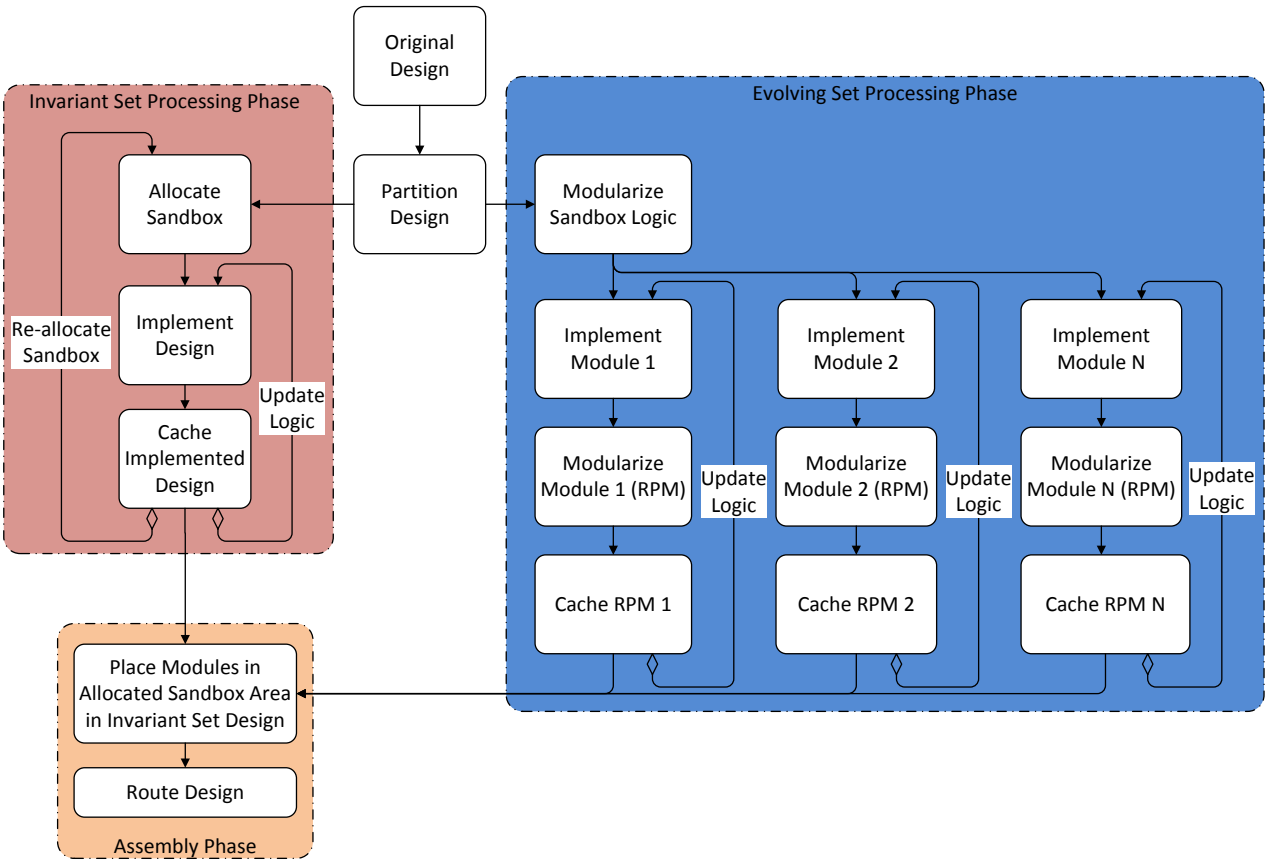


Figure 4.1: A high-level view of the proposed assembly paradigm

forth between these partitions over the course of development, yet the general concept is to cluster all the logic that is currently being refined into the evolving logic partition. Figure 4.2 shows an example partitioned design.

4.3.2 Invariant Set Implementation

The invariant set implementation process allocates a *sandbox*, a region on a device that excludes logic and routing resource utilization, that will eventually host the evolving logic. To yield a successful design assembly, the allocated sandbox area needs to satisfy the minimum

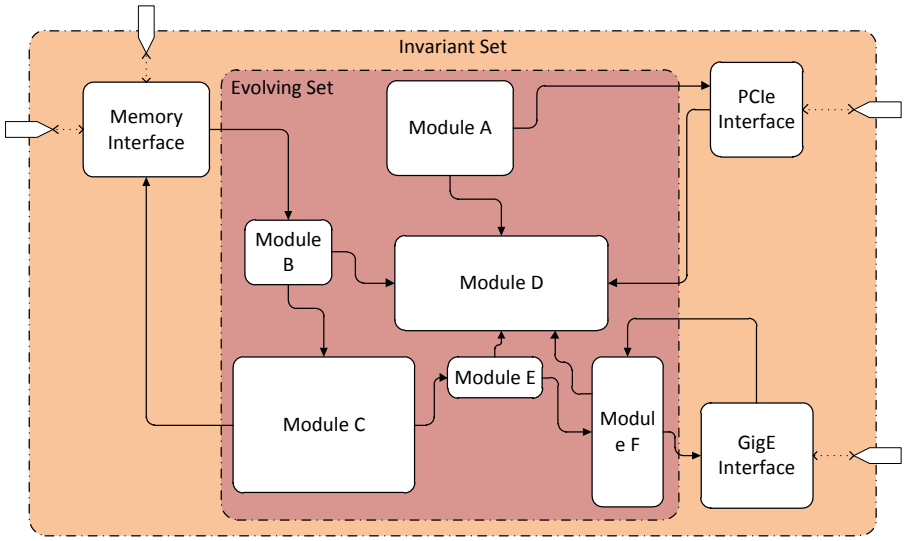


Figure 4.2: Partitioned Design

resource requirements for evolving logic. The vendor tools are then called to map, place and route the invariant logic. The outcome of this phase is a placed-and-routed design with dangling wires to the evolving set. Figure 4.4 illustrates an implemented invariant logic example with an allocated sandbox. The implemented invariant set netlist is then cached and retrieved during the design assembly phase, but never re-implemented until its corresponding logic changes, sparing the designer considerable amount of time. It is assumed that re-implementing the invariant set is much less frequent than re-implementing the evolving set.

4.3.3 Evolving Set Implementation

During the evolving set implementation process, every module of the evolving logic set, assumed to be hierarchical, is separately floorplanned, mapped, placed and routed. The

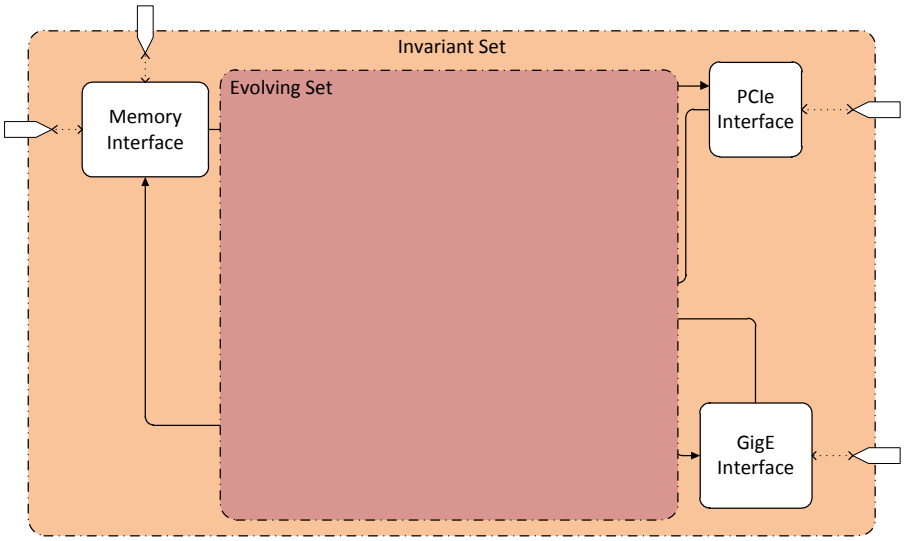


Figure 4.3: Invariant logic example with an allocated sandbox

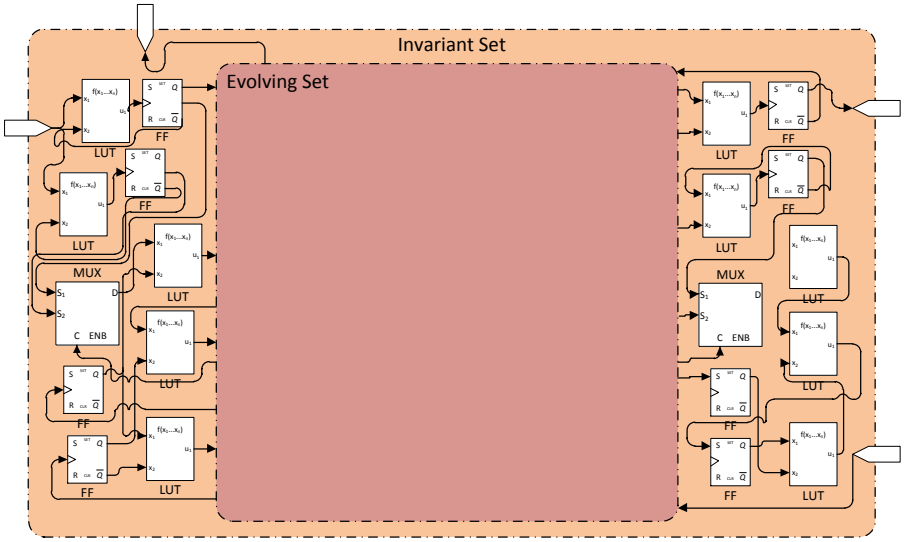


Figure 4.4: Invariant logic physical netlist

implementation result is represented as a *RPM*, a representation that provides structure to the design elements without the need to specify absolute placement location on the device [28]. An RPM representation of a module incorporates placement information, connectivity

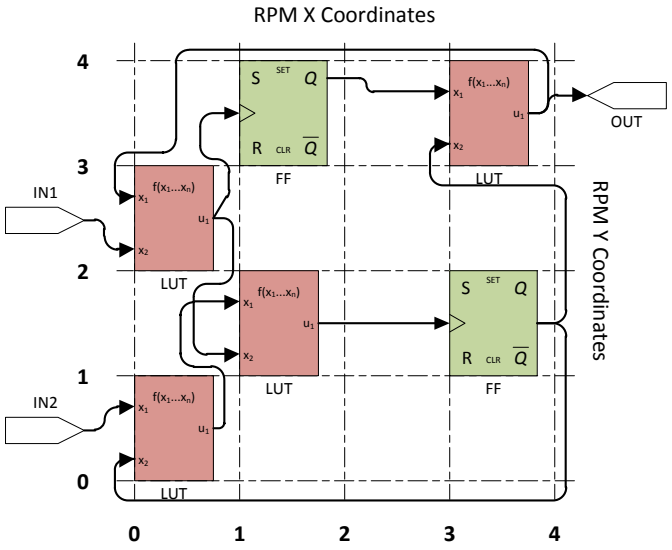


Figure 4.5: Evolving module implementation (RPM)

information as well as routing information about that module. Although the technique is originally meant to help implementation tools meet timing [28], the main purpose of using RPMs is their ability to easily relocate modules on the device. All module RPMs are then cached into a library for later retrieval, when needed during the design assembly phase, saving re-implementation time of their corresponding modules. With a hierarchical evolving set, the effect of a change to a module is local to the module itself and only the updated module is re-implemented. Figure 4.5 illustrates an example evolving module implementation in RPM representation.

4.3.4 Design Assembly

The final phase of the implementation process is the design assembly. During this phase, the implemented invariant set produced from section 4.3.2 is assembled with the implemented evolving set produced from section 4.3.3, by placing the corresponding evolving logic RPMs in the sandbox region reserved during the invariant set implementation phase. A routing stage finalizes the assembly and produces the final design. Details about placement and routing during the assembly phase follow.

Placer

Bounded by the sandbox resources and its shape, the evolving set RPMs required resources and their shapes, a placer places the evolving logic set by computing an absolute value for each module's RPM. The placement process runs until all modules are placed or until the placer fails to find a feasible solution. Several factors can cause the placer to fail such as insufficient sandbox resources or improper RPM floorplan. Different placement algorithms can also affect the feasibility, the quality as well as the convergence speed of the solution [49]. Figure 4.6 shows the first stage of the assembly with two evolving RPMs placed in the allocated sandbox.

Router

After a successfully placed design, a router traverses the evolving set connectivity list and routes all inter-module nets. The invariant set connectivity is already handled in section 4.3.2. The routing process runs until all nets are routed or until the router fails to find a feasible solution. The router fails mainly due to insufficient routing resources in congested

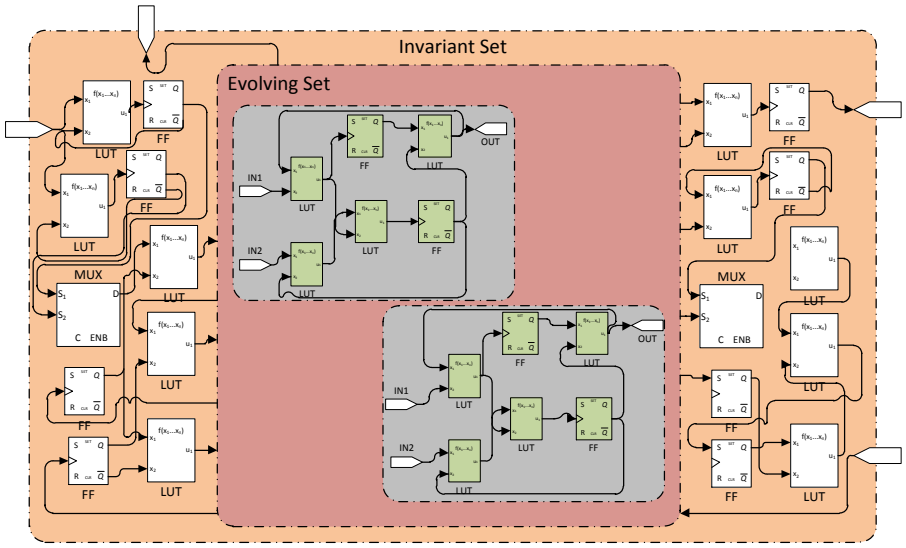


Figure 4.6: Two modules placed in the invariant logic implementation allocated sandbox

areas on the device. Different routing algorithms can also affect the feasibility, the quality as well as the convergence speed of the solution [49]. The routing phase ends by running a timing analysis step against all design timing constraints to verify that the assembled design meets timing. The resulting design is passed to the vendor tools to generate a configuration for the target device. Figure 4.7 shows the resulting assembled design.

4.4 Chapter Summary

This chapter describes an incremental implementation process that consists of four phases:

1. Design Partitioning
2. Invariant Set Implementation
3. Evolving Set Implementation

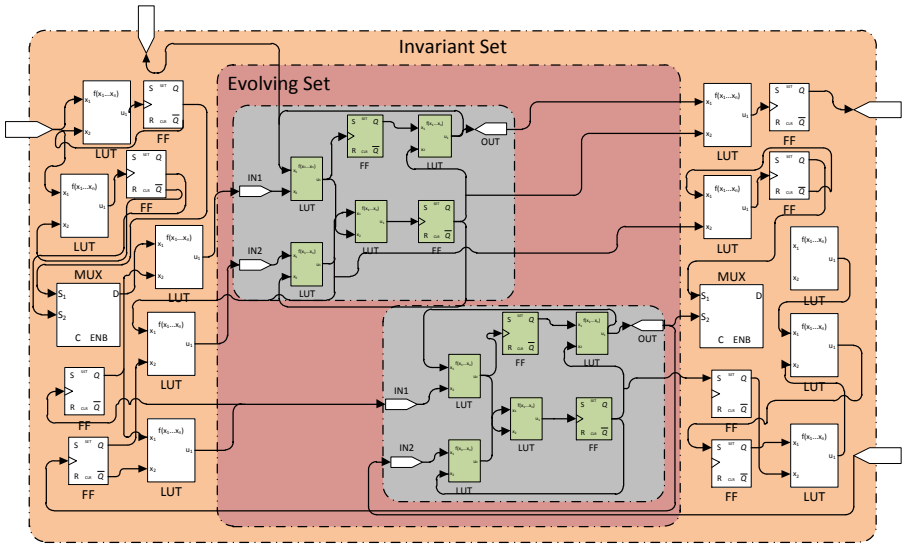


Figure 4.7: A fully placed and routed design

4. Design Assembly

Given a design, the process starts by (1) partitioning that design into two sets based on variance. The invariant set is first (2) implemented, and cached for re-use in future iterations of the design. The evolving (or variant) set is further partitioned into modules that are concurrently (3) implemented and cached into a library of relocatable modules. A final (4) assembly stage that consists of a module placer and a router combines the corresponding library modules and the implemented invariant set into the final solution.

Chapter 5

qFlow

This chapter presents qFlow, the implementation framework based on the assembly paradigm from Chapter 4. The chapter first introduces the reconfigurable computing platform TORC on which qFlow builds. The implementation details of the different parts of the framework are then presented along with a description of the resulting software packages.

5.1 Introduction

qFlow is a back-end productivity framework that implements the concepts presented in Chapter 4. Implemented in C++, qFlow is an FPGA back-end productivity framework that builds on TORC, an open-source infrastructure and tool set for reconfigurable computing [58]. The TORC tool-set has the capability to (1) read, write, and manipulate generic netlists, (2) read, write and manipulate physical netlists, (3) provide exhaustive wiring and logic information for commercial devices, and (4) read, write and manipulate bitstream packets [58]. qFlow extends TORC with physical netlist modularization techniques, RPM

grid generators, a coarse-grain placer and a metadata capturing framework. The entry point to qFlow is an Electronic Design Interchange Format (EDIF) netlist making it front-end tools agnostic; thus, capable of consuming input from graphical and textual tools. Currently, qFlow supports Xilinx Virtex-5 FPGA[1]. Details on the different component of the qFlow framework are described in the next sections. The reader is referred to Appendix C for a detailed description of all qFlow implementation and assembly stages.

5.2 Design Partitioning

During the partitioning phase, a design is partitioned into *invariant* and *evolving logic*. The design partitioning process can be done automatically, manually by the designer, or through a designer-guided process. qFlow leaves the process of partitioning the design to the designer. During design partitioning, the designer decides on the boundary separating the invariant and the evolving set, then creates a logical wrapper around the evolving set.

5.3 Design Jacket Generation

In qFlow nomenclature, the implemented invariant set is referred to as the *design jacket*. Generating the design jacket requires a two-step preparation phase. In Step 1, qFlow parses the logical wrapper interface (the wrapper wraps the evolving set logic) and generates anchor points for all connections going in and out of the wrapper, in the form of bus macros. A

[1]qFlow implements the RPM grid for Virtex-6, but does not have a placer for that architecture yet.

bus macro consists of a SLICE [2] on the FPGA device, with predefined input and output ports. Routing through bus macros is the way to instruct the vendor tools to create dangling wires that will be used later in the assembly phase to connect variant to evolving logic and vice versa. The wrapper along with the contained evolving logic is then replaced by a corresponding blackbox for implementation purposes. Step 2, qFlow allocates a *sandbox* using Xilinx *CONFIG PROHIBIT* constraints. The resulting design is implemented using the Xilinx tools and a fully placed and routed netlist (NCD format) is generated. The resulting netlist excludes any resources in the allocated *sandbox*. It is worth mentioning that vendor tools optimization techniques can be applied during design jacket implementation and that all timing information is reported back to the user. Figure 5.1 illustrates an FPGA Editor screenshot of an implemented design jacket. The orange colored rectangular region highlights the *sandbox* in the design.

5.4 Evolving Set Implementation

Assumed hierarchical, the *evolving set* consists of one or more interconnected modules. For implementation purposes, qFlow floorplans, maps and places each module as its own standalone design, allowing qFlow to handle all evolving modules in parallel. During the floorplan phase, qFlow estimates the resource utilization of every module and generates the corresponding AREA_GROUP constraints, confining a module's logic to a rectangular region on the device. During mapping and placement, qFlow calls the vendor tools on every module to achieve the task. The resulting physical netlist is then modularized, i.e., turned into a

[2] A SLICE is a grouping of two or more Xilinx Look-Up Tables (LUTs). A LUT is one of many primitive sites that reside on an FPGA device

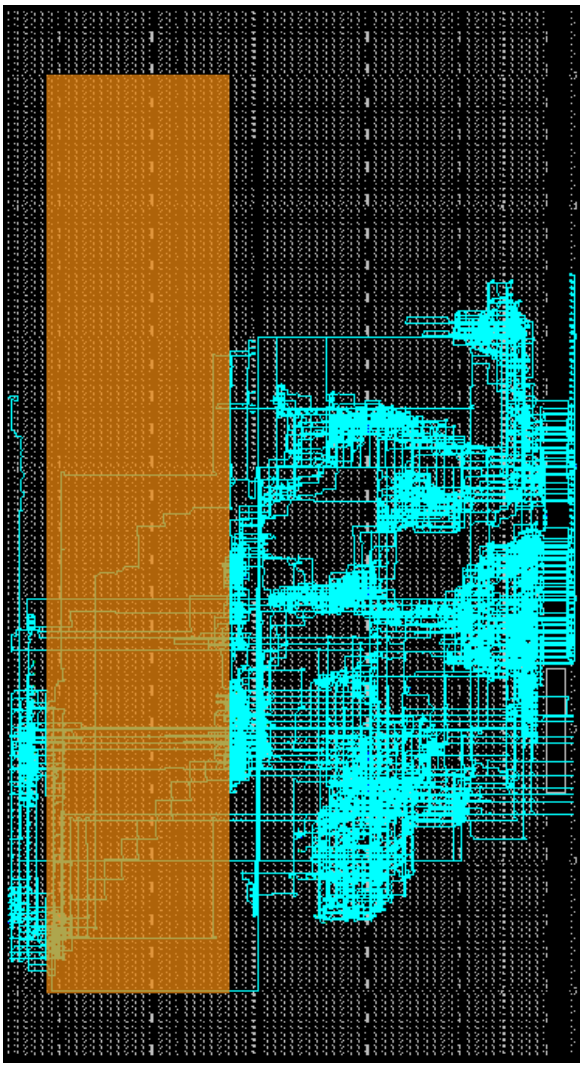


Figure 5.1: An implemented design jacket - FPGA Editor view

RPM (NMC format) easily relocatable on the device, and cached for retrieval during the assembly stage. As in section 5.3, vendor tools optimization techniques can be also applied to each module implementation. Figure 5.2 presents a XDL snippet of a mapped and placed adder, whereas Figure 5.3 shows the corresponding modularized form.

```
design "adder" xc5vlx110tff1136-1 v3.2 ,
inst "sum<2>" "SLICEL",placed CLBLL_X13Y48 SLICE_X22Y48  ,
  cfg " A5LUT::#OFF A6LUT:Madd_sum_addsub0000_lut<0>11:#LUT:O6=(A5@A6) ACY0::#OFF
    AFF:sum_0:#FF AFFINIT::INIT0 AFFMUX::O6 AFFSR::SRLOW AOUTMUX::#OFF
    AUSED::#OFF B5LUT::#OFF B6LUT:sum_addsub0000<1>1:#LUT:O6=((~A3*((~A4*(A5*A6))+
    (A4*(~A5+~A6))))+(A3*((~A4*(~A5+~A6))+(A4*(A5*A6))))) BCY0::#OFF BFF:sum_1:#FF
    BFFINIT::INIT0 BFFMUX::O6 BFFSR::SRLOW BOUTMUX::#OFF BUSED::#OFF C5LUT::#OFF
    C6LUT:Madd_sum_index00001:#LUT:O6=((~A3*(A4*A6))+(A3*((~A4*(A5*A6))+(A4*(A5+A6)))))
    CCY0::#OFF CEUSED::#OFF CFF:sum_2:#FF CFFINIT::INIT0 CFFMUX::O6 CFFSR::SRLOW
    CLKINV::CLK COUTMUX::#OFF COUTUSED::#OFF CUSED::#OFF D5LUT::#OFF D6LUT::#OFF
    DCY0::#OFF DFF::#OFF DFFINIT::#OFF DFFMUX::#OFF DFFSR::#OFF DOUTMUX::#OFF
    DUSED::#OFF PRECYINIT::#OFF REVUSED::#OFF SRUSED::0 SYNC_ATTR::ASYNC "
  ;
net "a<0>" ,
  inpin "sum<2>" A5 ,
  inpin "sum<2>" B5 ,
  inpin "sum<2>" C3 ,
  ...
```

Figure 5.2: A XDL snippet of a mapped and placed adder

5.5 Design Assembly

The assembly phase in turn is split into two phases, a placement phase and a routing phase, which stitch the design jacket and the evolving modules into the final design. Details about the placement and routing phases follow.

```
design "adder" xc5vlx110tff1136-1 v3.2;
module "adder_design" "sum<2>";
        port "a<1>_0" "sum<2>" "A5";
        port "a<0>_0" "sum<2>" "B3";
        port "b<1>_0" "sum<2>" "A6";
        port "b<0>_0" "sum<2>" "B4";
        port "clock<0>_0" "sum<2>" "CLK";
        port "reset<0>_0" "sum<2>" "SR";
        port "sum<2>_0" "sum<2>" "AQ";
        port "sum<1>_0" "sum<2>" "BQ";
        port "sum<0>_0" "sum<2>" "CQ";
        inst "sum<2>" "SLICEL", placed CLBLL_X13Y48 SLICE_X22Y48, cfg "A6LUT:
        Madd_sum_addsub0000_lut<0>11:#LUT:O6=(A5@A6) AFF:sum_0:#FF AFFINIT::INIT0 AFFMUX::O6
        AFFSR::SRLOW B6LUT:sum_addsub0000<1>1:#LUT:O6=((~A3*((~A4*(A5*A6))+(A4*(~A5+~A6))))
        +(A3*((~A4*(~A5+~A6))+(A4*(A5*A6))))) BFF:sum_1:#FF BFFINIT::INIT0 BFFMUX::O6 BFFSR::SRLOW
        C6LUT:Madd_sum_index00001:#LUT:O6=((~A3*(A4*A6))+(A3*((~A4*(A5*A6))+(A4*(A5+A6)))))
        CFF:sum_2:#FF CFFINIT::INIT0 CFFMUX::O6 CFFSR::SRLOW CLKINV::CLK SRUSED::0 SYNC_ATTR::ASYNC";
        net "$NET_0", inpin "sum<2>" A5;
        ...
endmodule "adder_design";
```

Figure 5.3: A XDL snippet of a modularized adder

5.5.1 Placement

The placer reads in the RPM representation of all modules and computes valid placement for each using a qFlow generated device *RPM grid*, a two-dimensional representation of all placeable sites on the device. RPM modules are placed by specifying the location of their

reference instance, the instance with respect to which all other instances in an RPM are placed. During placement, unmodified macros are retrieved from cache, whereas updated ones are re-implemented before placement proceeds. All inter-modules and modules to design jacket nets are also created. This section details the implementation details of a universal placement grid, called the RPM grid, and two placement algorithm: random and simulated-annealing placement.

RPM Grid Generation

Xilinx FPGAs include many resources, such as SLICE, Block RAM (BRAM) and DSP resources. A SLICE is a grouping of two or more Xilinx LUTs primitives, a BRAM is a primitive block RAM, and a DSP is a digital signal processing primitive. Each resource type is placed on a two-dimensional grid on the device; a Xilinx FPGA incorporates one grid per resource type, such as the SLICE grid, the BRAM grid and the DSP grid. Each grid has its own coordinate system and point of reference, and only captures placement information for the corresponding resource type. When trying to represent placement of *heterogeneous RPMs*, RPMs composed of more than one resource type, the task becomes quite challenging and the need for a universal coordinate system to represent all placeable resources emerges. For this purpose, Xilinx introduces the RPM grid with a universal coordinate system capturing all placeable resource types on a device. Currently, the only way of reconstructing the RPM grid is through FPGA Editor by manually clicking on each placeable site and reading the grid coordinates in the history window [70]. As the number of placeable sites on a device can go up to tens of thousand [3], the process of manually reconstructing the grid becomes

[3]The Xilinx University Program (XUP)-V5 platform used in this work hosts a Virtex-5 LX110T FPGA with more than $15,000$ placeable sites

tedious. Moreover, the RPM grid is device dependent and needs to be generated for every device separately.

qFlow automates the process of generating the RPM for Xilinx Virtex-5 and Virtex-6 FPGAs. The grid generation process consists of an architecture grammar and grid generator. The *architecture grammar* defines the coordinates, the offset rules traversing the different sites on a device and the point of reference. The *grid generator* generates the device RPM grid guided by the architecture grammar.

RPM Legal Placement

The RPM grid has an interesting property that turns out to be handy when generating legal RPM placements. For an FPGA device, the X value of the grid determines the resource type. A *legal placement* for an RPM module is a legal placement for the RPM reference instance that satisfies the resource requirements of all constituent instances of that RPM and does not violate its structure.

Computing legal placement for a RPM can be done by searching every placeable site on a FPGA device. Since the number of placeable sites on a device grows exponentially with the device size[4], an alternative approach that drastically prunes the search space is followed. In this approach, a module's legal RPM Y values are represented by a linear equation. A tile row scan of the device determines the module's legal RPM X values. The followed approach is independent of the number of placeable sites, but depends on a module's RPM size and

[4]The number of placeable sites on a device grows linearly as the device size grow. However the device size follows Moore's Law, thus growing exponentially every 18 months or so.

device tile column count, orders of magnitude smaller than placeable sites.

Computing all legal placement for a RPM module proceeds in the following manner. qFlow scans all instances of a RPM and picks the highest priority instance as the RPM reference point. Resource priority is based on resource scarcity, thus

$$Priority_{SLICE} \; < \; Priority_{BRAM} \; < \; Priority_{DSP} \tag{5.1}$$

All other instances in the RPM are represented relative to the highest priority instance in X and Y RPM offset. The highest priority instance type determines which of the following equations to use when computing the RPM Y range for the module:

$$RPM\_Y_{SLICE} \; = \; 2i \; , \; MIN_{SLICE} \; \leq \; 2i \; \leq \; MAX_{SLICE} \tag{5.2}$$

$$RPM\_Y_{BRAM} \; = \; 10j \; , \; MIN_{BRAM} \; \leq \; 10j \; \leq \; MAX_{BRAM} \tag{5.3}$$

$$RPM\_Y_{DSP} \; = \; 5k \; , \; MIN_{DSP} \; \leq \; 5k \; \leq \; MAX_{DSP} \tag{5.4}$$

MIN_{SLICE}, MIN_{BRAM}, MIN_{DSP} and MAX_{SLICE}, MAX_{BRAM}, MAX_{DSP} are the minimum and maximum, device dependent values that RPM_Y can take for each of the three resources. For example, a highest priority instance of type BRAM requires the use of Equation 5.3. Once the RPM Y range equation is fixed, qFlow scans the RPM instances one more time, and adjusts the MIN and MAX for the range equation taking into account the relative offset of all RPM module constituent instances. The resulting equation specifies the

range of legal RPM Y values for the module's reference instance. Finally, qFlow traverses each site in a device tile row, and saves the highest priority instance RPM X value for each legal placement. The combination of all saved RPM X values and RPM Y range equation list all possible legal RPM X and Y pair for the module's reference instance and thus legal placement for the module itself. It is worth mentioning that the above approach works because of the highly regular FPGA architecture.

Placement Algorithms

qFlow incorporates two placers: a random placer and a simulated annealing placer. Both placers are capable of placing device primitives such as SLICEs, BRAMs and DSPs, as well as coarse-grain RPM modules. They are also capable of avoiding rectangular regions on the device, specified as input parameters, during placement.

The random placer was first implemented during the proof-of-concept phase of the framework, and consists of two random number generators. When placing a module, the first random number generator randomly picks a legal RPM X value for the module's reference point from the list generated in section 5.5.1. The second random number generator randomly picks a legal RPM Y value for the module's reference point from the range equation in section 5.5.1. The resulting placement is verified against collision with any already placed logic before a legal placement is returned.

Random placement achieved speed of placement at the expense of solution quality. A SA placer was introduced to overcome the limitations of the random placer. The SA placer uses random placement as an initial placement and randomly shuffles modules. The wire length

is used a cost function. A placement that reduces the overall wire length is accepted, and a placement that increases the wire length is accepted based on a probability.

5.5.2 Routing

A successful placement initiates the router stage of the design. Due to its superior quality of results and timing awareness, qFlow leverages Xilinx PAR to perform all its routing, making qFlow a timing-aware framework with timing reporting capabilities. The re-entrant mode of PAR is used to handle the job. Figure 4.7 shows the resulting assembled design.

5.6 Metadata

qFlow generates considerable amount of logical and physical information when implementing a design. Therefore, qFlow leverages the work from [63] and [22], that mainly implement a management framework to capture design specific data in a reusable format, and extends it to capture information about each module's RPM, such as the reference instance and module's ports, the invariant logic anchor points during the design jacket generation, such as the pins location on the device, and the evolving logic connectivity, such as the nets going between the different evolving modules. Such information is mainly used during the assembly phase to generate the FPGA Editor script that will stitch the design together. The reader is referred to [63] and [22] for more information about the management framework.

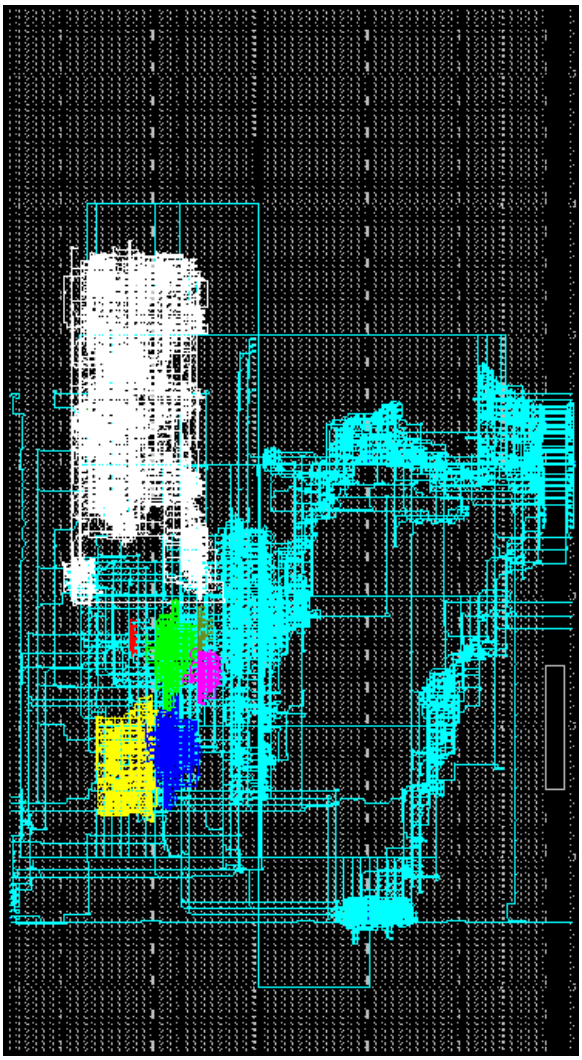


Figure 5.4: A fully assembled design - FPGA Editor view

5.7 Code Organization

qFlow's source code is organized into five packages[5]. Figure 5.5 shows qFlow's source code directory tree. Discussions regarding each package functionality, the constituent classes and their role in the framework, the qFlow-TORC integration and the qFlow process flow follow.

[5]qFlow is work in progress therefore the packages structure along with their classes are always evolving.

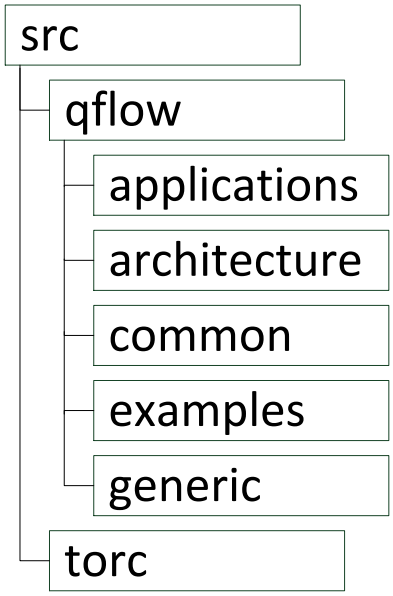


Figure 5.5: qFlow source code directory tree

5.7.1 Packages

applications The *application* package contains all of qFlow utilities classes. These classes consist of the following:

- **EdifToXML.cpp**: Extracts metadata information from an EDIF netlist.
- **HardMacroCreation.cpp**: Creates a hard-macro XDL representation of a physical netlist.
- **ModulePlacer_V5.cpp**: Places modules for Xilinx Virtex-5 architecture.
- **XMLPhysicalUpdater.cpp**: Updates a module metadata with physical netlist information.

- **XMLStaticUpdater.cpp**: Updates a design jacket metadata with physical netlist information.

- **XMLWrapperCombine.cpp**: Updates a sandbox metadata with physical netlist information.

architecture The *architecture* package hosts the device architecture information [6] relevant to placement along with the placement computer classes. These classes consist of the following:

- **PlacementComputer_V5.cpp / PlacementComputer_V5.hpp**: Computes placement for Virtex 5.

- **RPMGridGrammar_V5.cpp / RPMGridGrammar_V5.hpp**: Implements the RPM grid grammar for Virtex 5.

- **RPMGridGrammar_V6.cpp / RPMGridGrammar_V6.hpp**: Implements the RPM grid grammar for Virtex 6.

- **RPMGrid_V5.cpp / RPMGrid_V5.hpp**: Generates an RPM grid for Virtex 5.

- **RPMGrid_V6.cpp / RPMGrid_V6.hpp**: Generates an RPM grid for Virtex 6.

common & generic The *common* and *generic* packages provide two main functionality:

- Helper utilities called by the different utilities and classes of qFlow.

[6] As of the writing of this document, qFlow implements the RPM grid for Virtex 5 and Virtex 6. Placement, however, is only supported on Virtex 5.

- The metadata framework that extracts and stores logical and physical information needed by qFlow.

examples The *examples* package consists of example applications used to verify the correctness of the different qFlow utilities. These classes consist of the following:

- **RPMGridExample_V6.cpp**: RPM grid generation example for Virtex 6.
- **ModuleTransformerExample.cpp**: Modular transformer (flatten/modularize) example.
- **PlacementComputerExample_V5.cpp**: Placement computer example for Virtex 5.
- **RPMGridExample_V5.cpp**: RPM grid generation example for Virtex 5.

5.7.2 qFlow-TORC Integration

TORC is an object-oriented, open-source reconfigurable computing framework developed in C++ [58] [3]. The core functionality of the framework is facilitated by four Application Program Interfaces (APIs): a generic netlist API, a physical netlist API, a device architecture API, and a bitstream API.

qFlow builds on TORC and leverages many of TORC's capabilities. For example, the generic netlist API is called during metadata operations. The physical netlist is needed in netlist manipulation, whereas the device architecture API is necessary during design assembly.

qFlow also extends the framework APIs with new features. Specifically, the generic netlist API is extended with an export to XML feature. The physical netlist API is augmented with logic modularization and flattening capabilities. The device architecture API is enhanced with RPM grids generation code, and a module placer.

5.7.3 qFlow Process Flow

The qFlow framework consists of several software and hardware utilities coordinating to quickly implement a design. The process of generating a configuration using the framework requires design files to be processed through several stages. Figures 5.6 and 5.7 detail the whole implementation process flow. In particular, Figure 5.6 demonstrates the steps of generating a *design jacket*, implementing modules in the evolving set and extracting the connectivity of interest from the design. The same figure shows the different utilities used and the files stored in the framework library. Figure 5.7 highlights the assembly process, the utilities involved and the necessary files required for each stage of the assembly phase.

5.8 qFlow versus PR

FPGAs provide the flexibility of field programming, eliminating the need to go over the fabrication process every time a design change is required. PR further pushes FPGA flexibility by allowing the modification of parts of an FPGA while other parts are still running [2]. Although both flows incorporate invariant and variant logic (static and dynamic in PR nomenclature), qFlow and PR differ in flexibility and productivity.

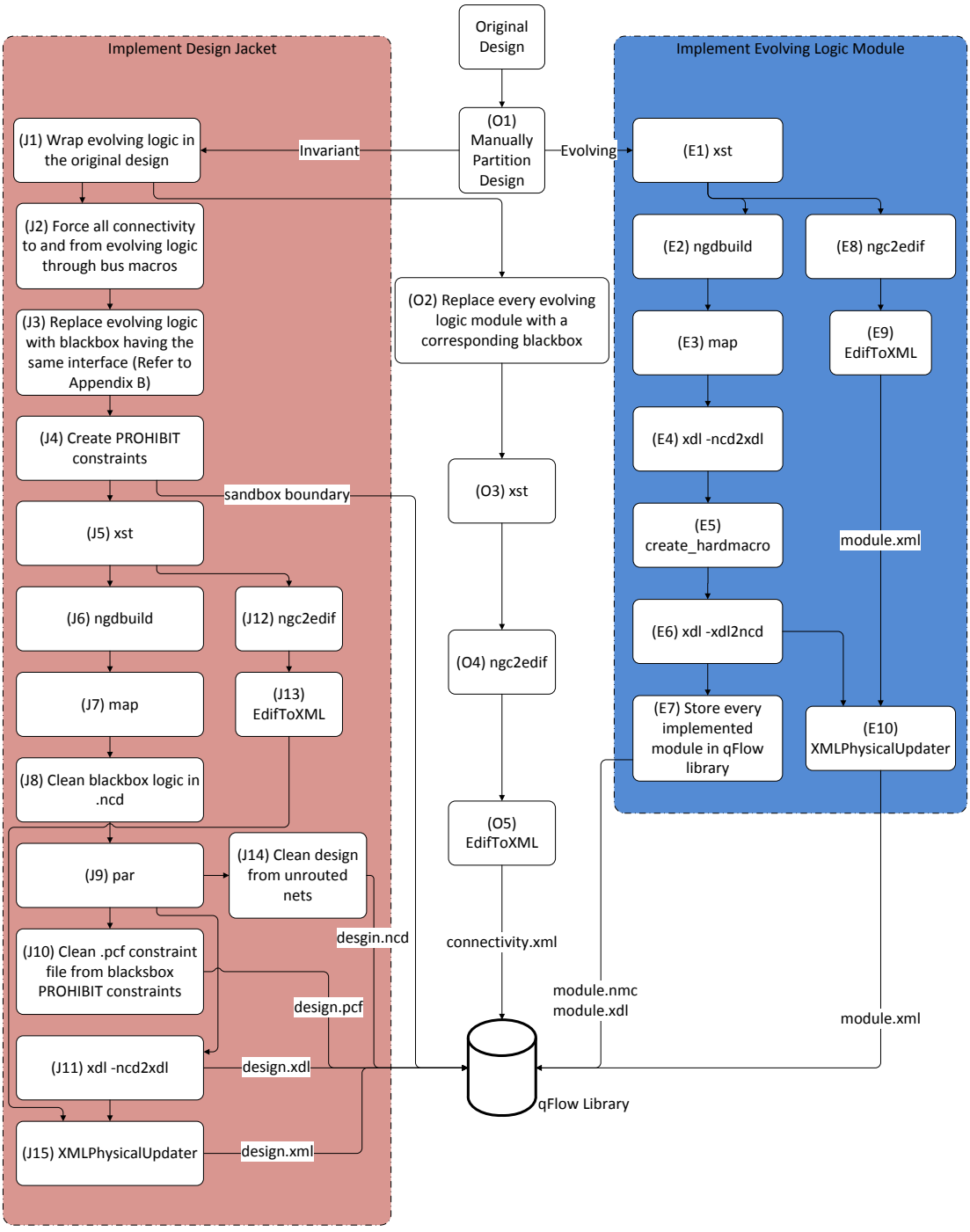


Figure 5.6: qFlow process flow for the design jacket and evolving logic implementation

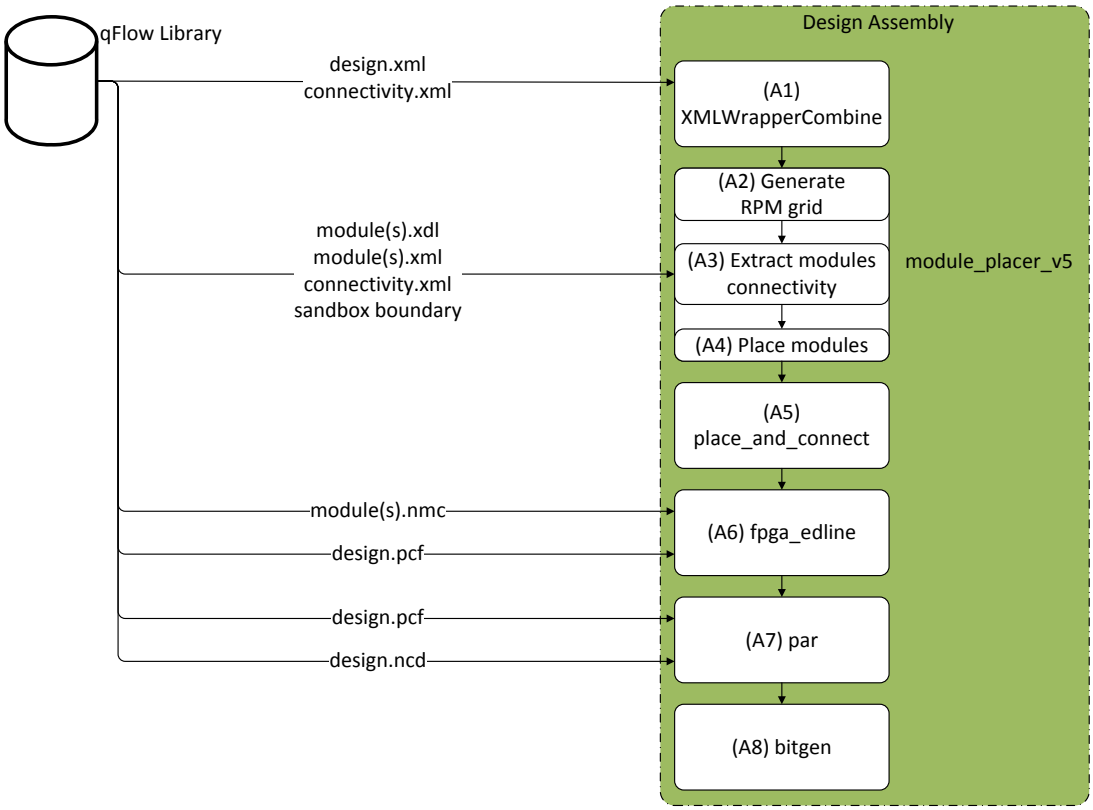


Figure 5.7: qFlow process flow for the design assembly phase

Xilinx PR flow is a slot-based reconfiguration flow, during which each dynamic region is assigned a slot on the device through floorplanning. A *dynamic region* is a region hosting a computation that can be updated while other parts of the device remain intact. The size of a dynamic region is set to be the largest of all hosted computations. The major weakness of PR lies in the process of preparing a hosted computation.

Assume a computation consisting of any combination of N modules, with a total number of 2^N possible computations. Preparing such computation for a PR dynamic region requires the implementation of 2^N computations. qFlow however, implements N modules then stitches

the final computation as needed at run-time, yielding higher flexibility and scalability as the number of modules increases. Furthermore, contrary to qFlow, PR flow does not leverage any offline processing causing its implementation times to be slower than qFlow.

5.9 Chapter Summary

This chapter details the implementation efforts of the qFlow framework. The framework consists of four phases. A manual design partitioning phase partitions the design into invariant and evolving logic. A jacket generation phase places and routes invariant logic. An evolving logic generation phase implements all evolving logic into relocatable logic. A final assembly stage stitches together the invariant netlist and the different relocatable logic using a custom placer and the vendor's router. The framework leverages TORC, an open-source framework for reconfigurable computing, and extends it with new features, such as a custom placer, grid generators, logic modularizer and flattener, and a XML exporter. The chapter concludes with a comparison between qFlow and PR flows.

Chapter 6

Benchmarks & Results

This chapter begins by briefly summarizing widely used FPGA benchmarks. The parameters of interest during the benchmark process are identified to be the implementation time, design clock speed, and resource utilization. Fifteen IWLS 2005 and six custom benchmark designs are then implemented using Xilinx ISE 13.4 and qFlow in ten experiments, and the results are thoroughly analyzed.

6.1 Benchmarks

6.1.1 Available FPGA Tools Benchmarks

FPGA benchmarks are produced by any of the following sources: conferences, industrial, or the open-source community. Following is a list of widely used and publicly available benchmarks:

The IWLS 2005 Benchmark Suite

IWLS 2005 benchmark suite includes designs from industry (Faraday Technology Corporation, Aeroflex Gaisler and Cadence), conferences (International Test Conference (ITC) 1999 and International Symposium on Circuits and Systems (ISCAS) 1989 and the open-source community (OpenCores). The suite is put together by Cadence Berkeley Laboratories [4] and contains 84 designs. The authors of [48] use the IWLS benchmark suite to evaluate new methods and tools on state encoding and combinational synthesis of sequential circuits. Mishchenko et al. [31] leverage the suite to test their preprocessing combinational logic technique.

The Microelectronics Center of North Carolina (MCNC) 1991 Benchmark Suite

Although it is distributed by MCNC, both academia and industry contributed to the MCNC 1991 benchmark suite. The suite consists of 35 designs that were assembled from earlier workshops and conferences. The benchmark suite is used in [42] to test a router for just-in-time FPGA compilation, in [39] when testing their fanin-based switch designs, and to evaluate placement techniques in [65].

The Golden 20 Benchmark Suite

The Golden 20 benchmark suite consists of 20 circuits extracted from the MCNC 1991 benchmark suite. This suite is commonly referred as the Toronto 20 benchmark suite [14]. Marquardt et al. [45] incorporate the Toronto 20 in their experiments, same as the authors of [59] to assess their architecture.

The Raw Architecture Workstation (RAW) Benchmark Suite

The RAW suite [11] is published by Massachusetts Institute of Technology reconfigurable architecture workstation project for performance evaluation of reconfigurable computing systems. The suite consists of 12 designs.

Other FPGA benchmarks do exist, yet they are either not freely available, or they concentrate on benchmarking System on Chip (SoC) using FPGAs and not FPGA tools. The reader is referred to [51] for an extensive survey on FPGA benchmarks.

6.1.2 What Are We Benchmarking?

The focus of the work is to accelerate the back-end process making implementation time and attained speedups the two parameters of interest. Nothing comes for free however, and the impact of achieved speedups on design clock speed and resource utilization is to be evaluated. In what follows, each benchmark design is implemented using Xilinx ISE 13.4 and qFlow and the back-end process *implementation time*, *design clock speed*, and *resource utilization* are reported for both flows. With the emphasis on the back-end process, the synthesis stage for both flows is excluded. However, even if the synthesis stage is included when measuring implementation times, qFlow is still expected to yield better results; qFlow synthesizes the invariant logic set, contrary to Xilinx ISE that does it every time the design is implemented. Moreover, qFlow reduces the input to synthesis through incremental and concurrent synthesis of all evolving logic modules, further reducing the synthesis time compared to Xilinx ISE.

6.2 Results

Two sets of benchmarks are used to evaluate qFlow. The first set consists of fifteen designs extracted from the IWLS 2005 benchmark suite [4], whereas the second is made of five larger designs, for a total of twenty benchmarks. Each design is implemented using Xilinx ISE 13.4 and qFlow for hundred runs. The average of compilation time, clock speed and device utilization is reported for all designs and both flows. Tables 6.1 and 6.3 briefly describe the selected benchmark designs. Tables 6.2 and 6.4 show the percentage of invariant logic and routing for the selected benchmark designs.

6.2.1 Experiment One

In the first experiment, the fifteen designs are implemented as *evolving logic*. Each benchmark design is implemented as one evolving logic module. All logic blocks in the module are placed and routed during the first run of the benchmark. In later runs, the stitching phase is only executed and the relative placement of the module's logic blocks is re-used. Table 6.5 presents the compilation times for Xilinx ISE 13.4, and qFlow, along with the speedup achieved using qFlow over Xilinx ISE 13.4. The table shows an average speedup of 1.2x using qFlow over Xilinx ISE 13.4. Throughout this chapter, speedup of flow one over flow two is defined in Equation 6.1.

$$Speedup = \frac{T_2}{T_1}, \tag{6.1}$$

where T_1 is the compilation time achieved by flow one, and T_2 is the compilation time achieved by flow two.

Table 6.1: Brief description of the fifteen IWLS 2005 designs

Design	**Platform**	**Description**
ac97_ctrl	Xilinx ML 509	WISHBONE AC 97 controller
aes_core	Xilinx ML 509	AES cipher
des_perf	Xilinx ML 509	DES optimized for performance
ethernet	Xilinx ML 509	Ethernet IP core
i2c	Xilinx ML 509	SHBONE revB.2 compliant I2C master controller
mem_ctrl	Xilinx ML 509	WISHBONE memory controller
pci_bridge32	Xilinx ML 509	PCI IP
sasc	Xilinx ML 509	Simple asynchronous serial controller
simple_spi	Xilinx ML 509	MC68HC11E based SPI interface
spi	Xilinx ML 509	SPI IP
tv80	Xilinx ML 509	TV80 8-bit microprocessor core
usb_funct	Xilinx ML 509	USB function core
usb_phy	Xilinx ML 509	USB 1.1 PHY
vga_lcd	Xilinx ML 509	WISHBONE rev.B2 compliant enhanced VGA/LCD controller
wb_dma	Xilinx ML 509	WISHBONE DMA/Bridge IP core

Three factors come together to cause the moderate speedup reported in this experiment. The first is a slow physical netlist interface. During the assembly stage, qFlow uses Xilinx *FPGA Edline*, the command line version of FPGA Editor, to merge the design jacket with the evolving logic RPM. However, FPGA Editor itself is a slow and outdated tool and a replacement is being developed by Xilinx [25]. Secondly, the speedup achieved using the

Table 6.2: Percentage of invariant logic and routing for the fifteen IWLS 2005 designs

Design	**% Invariant Logic**	**% Invariant Routing**
ac97_ctrl	27.12	7.11
aes_core	73.30	27.42
des_perf	23.42	5.17
ethernet	26.39	5.87
i2c	41.41	11.87
mem_ctrl	40.73	11.89
pci_bridge32	56.34	16.44
sasc	50.00	20.90
simple_spi	43.21	16.47
spi	42.96	12.69
tv80	13.94	2.95
usb_funct	37.59	9.36
usb_phy	48.84	19.41
vga_lcd	45.94	10.93
wb_dma	74.11	26.80

proposed framework is canceled out by the associated framework overhead given such small designs. For small designs, the need for acceleration is redundant in the first place . Thirdly, by design, all designs assumed no design jacket, which in turn reduces the speedup attained.

Table 6.3: Target platforms for the five sample designs

Design	**Platform**	**Description**
Gaussian Blur	Xilinx ML 509	Gaussian blur filtered video stream
Edge Detection	Xilinx ML 509	Edge detection filtered video stream
ZigBee	Xilinx ML 509	ZigBee radio
Vector Addition	Convey HC-1	One 64-bit vector add operation
Smith-Waterman	Convey HC-1	Smith-Waterman sequence alignment

Table 6.4: Percentage of invariant logic and routing for the five sample designs

Design	**% Invariant Logic**	**% Invariant Routing**
Gaussian Blur	6.43	2.64
Edge Detection	4.44	1.68
ZigBee	49.63	43.15
Vector Addition	90.66	88.92
Smith-Waterman	88.01	84.95

6.2.2 Experiment Two

In the second experiment, results of the runs from section 6.2.1 are reported with the assumption of an *ideal physical netlist interface*, an interface that takes insignificant time to merge two implemented designs [1], assuming the two designs have no resource conflicts. Table

[1] Although there is no resource conflict when merging the design jacket netlist (NCD format) with an evolving logic hard macro, it still takes FPGA Editor a considerable amount of time to complete the opera-

Table 6.5: Xilinx ISE 13.4 and qFlow compilation times for fifteen IWLS 2005 designs

Design	**ISE Impl. Time (s)**	**qFlow Impl. Time (s)**	**Speedup**
ac97_ctrl	146	113	1.3
aes_core	120	100	1.2
des_perf	123	120	1.0
ethernet	137	178	0.8
i2c	85	80	1.1
mem_ctrl	206	173	1.2
pci_bridge32	158	109	1.5
sasc	73	78	0.9
simple_spi	75	78	1.0
spi	115	92	1.3
tv80	168	100	1.7
usb_funct	123	130	1.0
usb_phy	80	78	1.0
vga_lcd	133	97	1.4
wb_dma	153	107	1.4

6.6 presents the compilation times for Xilinx ISE 13.4, and qFlow along with the average speedup achieved using qFlow over Xilinx ISE 13.4. The table shows an average speedup of 2x using qFlow over Xilinx ISE 13.4.

tion.

Table 6.6: Xilinx ISE 13.4 and qFlow (ideal physical netlist interface) compilation times for fifteen IWLS 2005 designs

Design	**ISE Impl. Time (s)**	**qFlow Impl. Time (s)**	**Speedup**
ac97_ctrl	146	73	2.0
aes_core	120	54	2.2
des_perf	123	72	1.7
ethernet	137	127	1.1
i2c	85	47	1.8
mem_ctrl	206	126	1.6
pci_bridge32	158	60	2.6
sasc	73	46	1.6
simple_spi	75	46	1.6
spi	115	56	2.1
tv80	168	64	2.6
usb_funct	123	83	1.5
usb_phy	80	46	1.7
vga_lcd	133	55	2.4
wb_dma	153	57	2.7

6.2.3 Experiment Three

Placement and routing in both sections 6.2.1 and 6.2.2 are timing driven. However, the clock speed was not constraint to any specific value, putting the implementation tools in

Performance Evaluation Mode, a mode during which the tools will automatically constrain detected design clock (or clocks) to the highest achievable speed. In experiment three, the runs from experiment one are repeated and the average achieved clock speeds for Xilinx ISE 13.4 are presented in Table 6.7.

Table 6.7: Xilinx ISE 13.4 and qFlow achieved clock speeds for fifteen IWLS 2005 designs

Design	**ISE Clock Speed**	**qFlow Clock Speed**	**Slowdown**
ac97_ctrl	340 MHz	267 MHz	1.27
aes_core	257 MHz	206 MHz	1.24
des_perf	294 MHz	223 MHz	1.31
ethernet	143 MHz	108 MHz	1.32
i2c	346 MHz	322 MHz	1.07
mem_ctrl	240 MHz	154 MHz	1.56
pci_bridge32	224 MHz	156 MHz	1.44
sasc	384 MHz	362 MHz	1.06
simple_spi	349 MHz	339 MHz	1.03
spi	126 MHz	128 MHz	0.98
tv80	106 MHz	110 MHz	0.96
usb_funct	221 MHz	177 MHz	1.25
usb_phy	499 MHz	400 MHz	1.25
vga_lcd	252 MHz	218 MHz	1.16
wb_dma	279 MHz	225 MHz	1.24

Two major factors come together to cause the clock slowdown when using qFlow. The first is due to bad bus macro placement during the jacket creation, yielding longer combinational paths. When creating a jacket, the tools are not instructed to lay the bus macros around the sandbox boundary, therefore will place these macros all over the device. The second contributor to the clock slowdown is the qFlow placer itself. With placement and routing not the emphasis of the work but a mean to an end, little effort was put in qFlow placer. A better placer is expected to yield higher clock speeds, but will likely incur higher runtimes.

6.2.4 Experiment Four

In experiment four, the clock constraint for Xilinx ISE tools on every design are set to match the ones achieved by qFlow from Table 6.7. The goal is to measure attained speedups for matching clock speeds. All fifteen benchmarks are once again implemented using Xilinx ISE 13.4. Table 6.8 presents the corresponding compilation times for Xilinx ISE, and qFlow along with the average speedup achieved using qFlow over Xilinx ISE 13.4. The table shows an average speedup of 1.7x using qFlow over Xilinx ISE 13.4, given a specific clock speed per design.

6.2.5 Experiment Five

In experiment five, the resource utilization of all fifteen benchmarks implemented using Xilinx ISE and qFlow are extracted, and the numbers are presented in Table 6.9 and Table 6.10.

Table 6.8: Xilinx ISE 13.4 (matching clock speed) and qFlow (ideal physical netlist interface) compilation times for fifteen IWLS 2005 designs

Design	**Clock Speed**	**ISE Impl. Time (s)**	**qFlow Impl. Time (s)**	**Speedup**
ac97_ctrl	267 MHz	117	73	1.6
aes_core	206 MHz	108	54	2.0
des_perf	223 MHz	109	72	1.5
ethernet	108 MHz	140	127	1.1
i2c	322 MHz	85	47	1.8
mem_ctrl	154 MHz	145	126	1.2
pci_bridge32	156 MHz	130	60	2.2
sasc	362 MHz	70	46	1.5
simple_spi	339 MHz	78	46	1.7
spi	128 MHz	116	56	2.1
tv80	110 MHz	128	64	2.0
usb_funct	177 MHz	110	83	1.3
usb_phy	400 MHz	70	46	1.5
vga_lcd	218 MHz	120	55	2.2
wb_dma	225 MHz	111	57	1.9

6.2.6 Experiment Six

In experiment six, one of the fifteen designs, *sasc*, is incrementally implemented using Xilinx ISE 13.4 and qFlow. The build starts with a design incorporating one *sasc* module.

Table 6.9: Resource utilization for fifteen IWLS 2005 designs implemented by Xilinx ISE 13.4

Design	**Registers**	**LUTs**	**BRAM**	**DSP**
ac97_ctrl	1031	1221	0	0
aes_core	530	442	10	0
des_perf	2512	1904	0	0
ethernet	2334	1904	3	0
i2c	133	174	0	0
mem_ctrl	977	1191	0	0
pci_bridge32	1085	918	2	0
sasc	52	67	0	0
simple_spi	71	101	0	0
spi	247	552	0	0
tv80	237	1374	0	0
usb_funct	1557	1724	0	0
usb_phy	97	84	0	0
vga_lcd	788	934	2	0
wb_dma	543	703	0	0

The number of *sasc* modules is gradually increased to six. All designs in this experiment are pushed through qFlow as evolving logic. Figures 6.1 and 6.2 presents the compilation times for Xilinx ISE 13.4, and qFlow.

Table 6.10: Resource utilization for fifteen IWLS 2005 designs implemented by qFlow

Design	**Registers**	**LUTs**	**BRAM**	**DSP**
ac97_ctrl	1031	1301	0	0
aes_core	530	843	5	0
des_perf	2512	2100	0	0
ethernet	2335	2135	2	0
i2c	134	218	0	0
mem_ctrl	1094	1468	0	0
pci_bridge32	1086	1304	2	0
sasc	52	104	0	0
simple_spi	71	138	0	0
spi	247	656	0	0
tv80	237	1450	0	0
usb_funct	1593	1984	0	0
usb_phy	97	129	0	0
vga_lcd	814	1146	2	0
wb_dma	543	1147	0	0

Figure 6.1 shows the compilation numbers with a linear trend line for each of the three flows, whereas Figure 6.2 displays the same compilation numbers but with exponential trend lines instead. The linear trend line is expected as a lower bound on placement and routing tools, whereas the exponential trend is treated as an upper bound on the same tools. The two figures show a noticeable decrease in compilation times using qFlow for incremental

builds over ISE, for all three cases, lower bound trend, upper bound trend and experimental values.

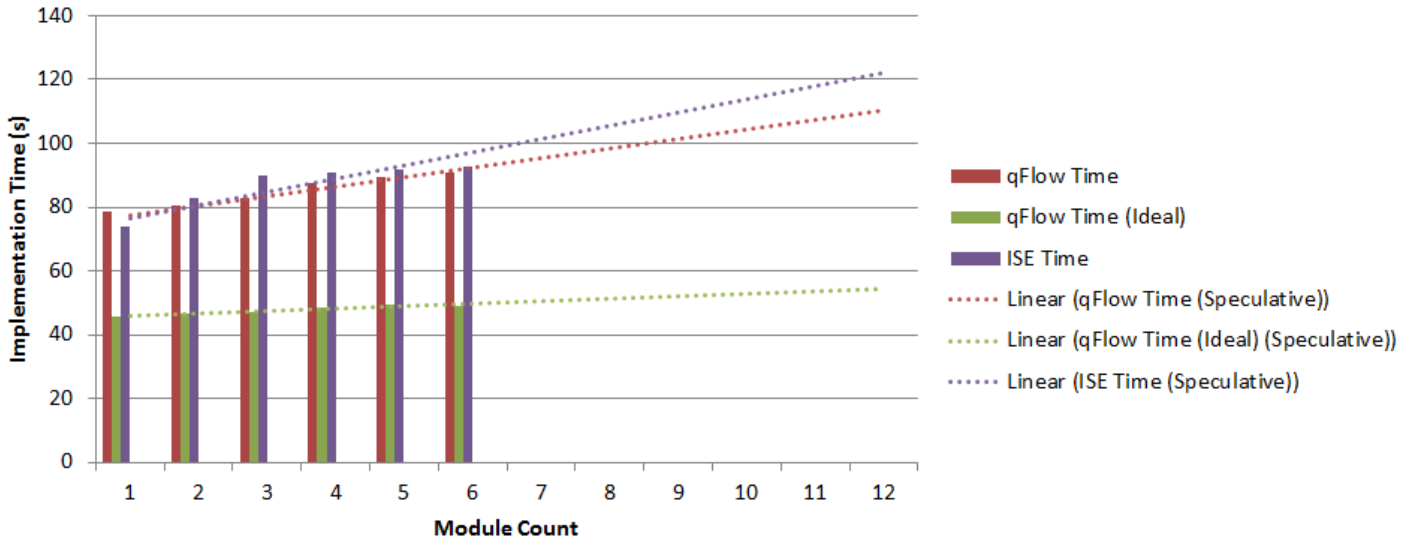


Figure 6.1: Xilinx ISE 13.4, qFlow, and qFlow (ideal physical netlist interface) compilation times for incremental builds of *sasc* with linear trend lines

6.3 Custom Benchmarks

The real speedup that qFlow offers is for larger designs that incorporate both evolving and invariant logic. Table 6.3 describes five designs along with the target platform of each of the designs, extracted from real designs used in video processing, software defined radio and high-performance computing projects. Each design is implemented using Xilinx ISE 13.4 and qFlow for hundred runs. The average of compilation time, clock speed and device utilization is reported for all designs and both flows.

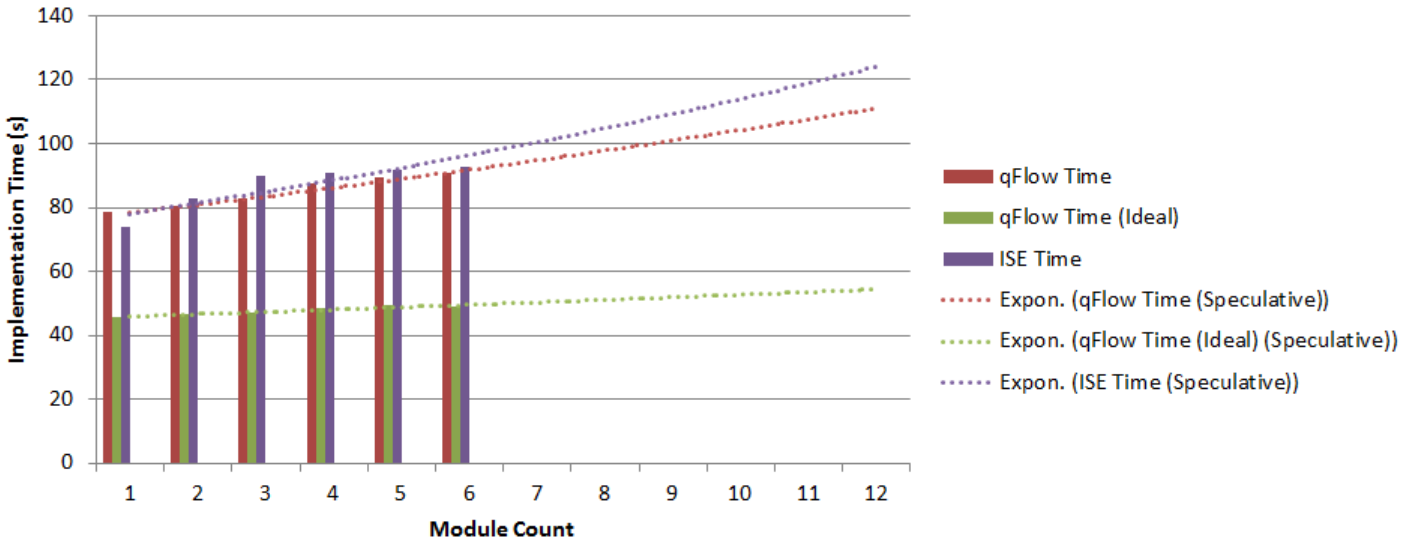


Figure 6.2: Xilinx ISE 13.4, qFlow, and qFlow (ideal physical netlist interface) compilation times for incremental builds of *sasc* with exponential trend lines

6.3.1 Experiment Seven

In experiment seven, the five designs are manually partitioned first into evolving and invariant logic. Table 6.11 describes the content of both partitions for all five designs. All designs are implemented using Xilinx ISE 13.4 and qFlow and the results are reported in Table 6.12. Each of these designs consists of a design jacket and one or more evolving modules. A design jacket is generated once per design, and re-used through the benchmarking process of that design. An average speedup of 2.94x is attained compared to Xilinx ISE 13.4 for the five designs, and an average 5.2x speedup for the large Convey HC-1 designs.

[1]Convey HC-1 framework includes includes eight memory controllers, a dispatch logic interface, a management interfaces, glue logic and more

Table 6.11: Logic partitioning for the five benchmark designs

Design	**Invariant Logic Partition**	**Evolving Logic Partition**
Gaussian Blur	VGA and DVI interfaces	Video Filter
Edge Detection	VGA and DVI interfaces	Video Filter
ZigBee	GigE and signal decoding logic	ZigBee radio components
Vector Addition	Convey HC-1 framework [1]	Vector add computation
Smith-Waterman	Convey HC-1 framework	Smith-Waterman computation

Table 6.12: Xilinx ISE 13.4 and qFlow compilation times for five sample designs

Design	**ISE Impl. Time (s)**	**qFlow Impl. Time (s)**	**Speedup**
Gaussian Blur	159	139	1.15
Edge Detection	185	150	1.24
ZigBee	246	130	1.89
Vector Addition	4522	716	6.31
Smith-Waterman	7027	1703	4.12

6.3.2 Experiment Eight

In experiment eight, results of the runs from section 6.3.1 are reported with the assumption of an ideal physical netlist interface. Table 6.13 presents the compilation times for Xilinx ISE 13.4, and qFlow along with the average speedup achieved using qFlow over Xilinx ISE 13.4. The table shows an average speedup of 5.1x using qFlow over Xilinx ISE 13.4 for the

five design and an order of magnitude speedup for the large Convey HC-1 designs.

Table 6.13: Xilinx ISE 13.4 and qFlow (Ideal Phys. IF) compilation times for five sample designs

Design	**ISE Impl. Time (s)**	**qFlow Impl. Time (s)**	**Speedup**
Gaussian Blur	159	83	1.9
Edge Detection	185	90	2
ZigBee	246	81	3
Vector Addition	4522	448	10.1
Smith-Waterman	7027	799	8.8

6.3.3 Experiment Nine

Each of the five designs has specific clock speed requirements. In experiment nine, the runs from experiment six are repeated and clock speeds for runs from Xilinx ISE 13.4 and qFlow are collected. In all of the six experiments, qFlow succeeded in meeting the same timing requirements as the Xilinx ISE 13.4 tools. Results are presented in Table 6.14.

6.3.4 Experiment Ten

In experiment ten, the resource utilization of all five benchmarks implemented using Xilinx ISE and qFlow are extracted, and the numbers are presented in Table 6.15 and Table 6.16.

Table 6.14: Xilinx ISE 13.4 and qFlow achieved clock speeds for five sample designs

Design	**ISE Clock Speed**	**qFlow Clock Speed**
Gaussian Blur	100 MHz	100 MHz
Edge Detection	100 MHz	100 MHz
ZigBee	125 MHz	125 MHz
Vector Addition	150 MHz	150 MHz
Smith-Waterman	150 MHz	150 MHz

Table 6.15: Resource utilization for five sample designs implemented by Xilinx ISE 13.4

Design	**Registers**	**LUTs**	**BRAM**	**DSP**
Gaussian Blur	4547	3833	0	18
Edge Detection	8607	3000	0	14
ZigBee	2980	3244	8	10
Vector Addition	58033	53717	53	0
Smith-Waterman	66441	60245	52	0

6.4 Analysis of Results

Even with a slow physical netlist interface, fourteen out of the twenty designs showed a reduction in compilation time when using qFlow, at an average clock slowdown of 1.3 x. The clock slowdown however, was computed by putting both flows in *performance evalation mode*, thus pushing them to their limit. Therefore, the average clock slowdown exposes the

Table 6.16: Resource utilization for five sample designs implemented by qFlow

Design	**Registers**	**LUTs**	**BRAM**	**DSP**
Gaussian Blur	4545	3899	0	18
Edge Detection	8608	3077	0	14
ZigBee	3396	5350	9	10
Vector Addition	24316	32588	60	0
Smith-Waterman	70217	62736	52	0

superiority of the Xilinx ISE tools compared to qFlow when aiming for higher clock speeds, but says less about qFlow's ability to implement high-performance designs. In fact, all five minimum clock periods are met by qFlow when implementing the five custom benchmarks with clock speeds up to 150 MHz. Implementing regular designs such as systolic arrays (section 6.2.6) showed a reduction in compilation time too, mainly due to qFlow's caching mechanism where for an array of size N, only one cell will be implemented and cached, and the implementation of the remaining identical $N-1$ cells will be retrieved from cache.

The major weakness of qFlow is its dependence on FPGA Editor as the interface to the implemented netlist (NCD or binary format). To the knowledge of the author, FPGA Editor (or its command line version *fpga_edline*) is the only Xilinx interface to the NCD netlist, forming a bottleneck for qFlow. Alternative solutions can surpass such limitations by going to a textual representation of the netlist (XDL format). However, when a device bitstream is needed, a binary netlist is a requirement and conversion from a textual to a binary netlist turns out to be time consuming [37].

The other weakness of qFlow is a lower device utilization, induced by the use of hard macros. Hard macros are useful when it comes to implementation results preservation, but they have a fixed shape. Device fragmentation though, can become a limiting factor with a sub-optimal hard macro placement; Unused regions between hard macros that fail to host new macros due to shape and not to available resources will be wasted.

As for resource utilization, qFlow incurs LUTs and registers overhead. The major contributor to the LUT overhead are the anchor points between the design jacket and the evolving logic. As discussed in Chapter 5, all nets (Except clock nets) crossing the variant-invariant logic boundary are routed through bus-macros, with each bus-macro consisting of four LUTs on a Virtex-5 device. The major contributor to the registers overhead is the lack of constant pushing when implementing evolving logic modules. For example, a register in a module might have a constant input coming from an outside source, yet with the current implementation tools, it is not possible to extract such information when modules are implemented separately.

The average invariance percentage for the large Convey designs is 89.34% for logic and 86.94% for routing. Experiments show that the large Convey designs yield the highest speedups among the twenty benchmark designs. The video processing benchmarks (Gaussian Blur and Edge Detection) have a low average invariance percentage of 5.44% for logic and 2.16% for routing, yet experiments show that an average of 1.95x is attained. Such gain is mainly caused by the large size of the video processing filters (evolving modules), thus a large re-use for the evolving logic. Therefore, experiments from the Convey and the video processing benchmarks suggest that two sets of designs will benefit the most from a

framework such as qFlow: designs with a large invariant logic set and designs with large evolving modules sizes.

6.5 Ease of Migration to Newer Xilinx Architectures

As of the writing of this document, qFlow reduces design implementation time on Xilinx Virtex-5. However, two newer generations are commercially available from Xilinx, the Virtex-6 and the Virtex-7 FPGA family. A quick evaluation on the framework architecture dependence on the Virtex-5 architecture along with the ease of migration to newer generations is therefore worthwhile. An assessment of the implementation effort for each part of qFlow follows.

Design Partitioning The design partitioning step is performed at the logical level, and by definition is architecture independent. As of the writing of this document, it is a manual process performed by the designer.

Metadata qFlow leverages the management framework from [63] and [22] to capture logical and physical information about a design. The framework is built with the ease of extension in mind, and currently supports Xilinx Virtex-5, Virtex-6 and Virtex-7 Xilinx architectures.

Design Jacket Generation The design jacket generation completely depends on the corresponding vendor tools. Although vendor tools are device and architecture dependent, the burden of such migration is on the vendors themselves, and not on qFlow designers.

Evolving Set Implementation The evolving set implementation mostly depends on the corresponding vendor tools. Each module in the set is passed through the vendor tools first, then a modularization step processes the outcome. The vendor tools for every new architecture are assumed to be provided, and the sole required effort in this stage is the migration of the modularization step. During the development of qFlow, the developed modularization technique was pushed into TORC's source code, and currently works under all the Xilinx architectures supported by TORC.

Placement The most effort required to migrate the framework to a different architecture lies in migrating the placer itself. Aside from minor architecture dependent tweaks, placement algorithms remain intact during the migration process. The module placement grid however, is device and architecture dependent. Vendor tools do not provide an easy way to generate that grid, and it is up to the framework developer to implement the grid. The framework to generate a new grid for a new device and architecture is already embedded into the qFlow framework, and provides all the hooks to be easily extendable.

Routing qFlow uses Xilinx PAR as its router, another vendor tool. As in the evolving set implementation and design jacket generation, the burden of migrating the tool to newer architectures lies on the vendors themselves, and not on qFlow designers.

6.6 qFlow and Non-Xilinx Architectures

Several FPGA vendors exist today, however the choice to implement a solution for Xilinx FPGAs is not arbitrary. Xilinx provides insertion points and reveals valuable architecture

information that make solutions such as TORC and qFlow viable. Otherwise, the concepts proposed in Chapter 4 are not in any way Xilinx specific. In fact, given the following vendor tools capabilities and architecture data, qFlow could be implemented for any FPGA device:

1. The ability to intercept the implementation flow after synthesis, placement and routing.
2. The ability to parse a netlist after synthesis, placement and routing.
3. The ability to instruct the implementation tools to avoid regions on the device during placement. The ability to instruct the tools to avoid regions on the device during routing would be useful, but is not necessary for the realization of qFlow.
4. The ability to confine logic to a specific region on the device during placement and routing.
5. The ability to easily relocate a placed netlist on a device.
6. The ability to combine two or more placed and/or routed (or partially routed) netlists into a final netlist.
7. The ability to start routing from a partially routed netlist.
8. The ability to obtain enough device architecture data for placement. The ability to obtain device architecture data for routing would be useful, but is not necessary for the realization of qFlow

Chapter 7

Conclusions & Future Work

7.1 Conclusions

Despite their high-performance and modest power consumption, FPGAs are often associated with long place and route times due to their programming model. FPGA implementation tools strive to generate optimal designs, yielding high-quality results that can take hours or even days to generate. While such compilation times are acceptable for an ASIC engineer, they come at a high-price for the configurable computing community, where software-like compile times are desirable.

Place and route can take up to 90% of the FPGA compilation process [66]. Theoretical limits on solving such computationally demanding problems are identified and the compilation process is examined in the light of identifying aspects that will minimize the computational load for incremental compilation. Offline and run-time processing techniques for the placement and routing problems are highlighted, and a model for the run-time complexity of the

two problems in light of the aforementioned techniques is presented.

The work proposes a design assembly paradigm for FPGA development that leverages the concept of design re-use to speedup the configuration generation process. The paradigm partitions a design into two classes, an invariant set and an evolving set, and applies different implementation strategies for each of the classes. Through the application of incremental assembly techniques, the work strives to quickly converge the implementation tools to a solution.

A FPGA implementation framework is built based on the proposed assembly paradigm. The framework is benchmarked using fifteen IWLS 2005 and five custom designs, and up to 10x speedup in implementation time is achieved over vendor tools for an average 1.2x slow-down in clock speed. Moreover, designs with clock speeds of up to 400MHz are successfully implemented using qFlow.

There are a few ways of interpreting the results. One is that a designer has the ability to iterate over their design up to ten times more over a given period of time. Another is that with a greatly reduced back-end compilation time, a designer is free to explore more design alternatives.

7.2 Research Contributions

The realization of this work produced the following contributions:

1. An assembly paradigm for FPGA development that leverages logic hierarchy and vari-

ance to partition a design into two sets. Each set follows separate implementation strategies based on specific constraints and requirements (Chapter 4).

2. An implementation framework for FPGA development called qFlow. qFlow builds on TORC, an open-source infrastructure and tool set for reconfigurable computing, and is designed with the ease of migration to newer vendor architectures in mind. qFlow also extends TORC's APIs with new features. A distinguishing feature of qFlow is its timing awareness (Chapter 5).

3. An evaluation of the proposed design assembly in light of the developed framework. Fifteen benchmark designs from IWLS 2005 and six customs benchmark designs covering a multitude of design domains are evaluated using the vendor tools and qFlow, and the runtime, clock speed and resource utilization are examined and analyzed (Chapter 6).

7.3 Future Work

The qFlow framework provides a solution for the long back-end compilations problem. The solution however, does not reach its full potential yet, and the author suggests the following expansions to the framework:

- *Design Partitioning*: The design partitioning process is currently a manual process performed by the designer, as well as the *design jacket* generation. It would be valuable to explore and augment the framework with partitioning techniques that maximize reuse, thus decrease compilation times.

- *Design Timing*: qFlow is a timing-aware solution. All timing information is collected and reported back to the framework at the end of the routing phase. However, currently qFlow does not take any actions in case an assembly fails to meet timing and leaves it up to the designer to act accordingly. One way to get around this issue is by leveraging the multi cores on modern machines and simultaneously exploring several assembly trials.

- *Module Routing*: Evolving logic modules are implemented by qFlow as hard macros, that contain placement, connectivity but not routing information. Studying the effect of including routing information and the corresponding achieved speedup is worth looking at, specifically now that newer releases of the vendor tools claim better modular and out-of-context module implementation support.

- *Non-Xilinx FPGA Architectures*: qFlow is implemented and tested on a Xilinx architecture. However, as argued earlier in Chapter 6, the concepts should apply equally to non-Xilinx architectures. A close investigation of such architectures along with modular and incremental implementation support would be a valuable addition to this work.

- *Placement and Routing and FPT*: While this work leverages heuristics to solve place and route problems, approaching the problem from an FPT perspective, such as studying the effect of fixing the number of evolving modules or the size of an evolving module on the run-time complexity of placement and routing, provides a valuable insight into this work.

Bibliography

[1] *Introduction to Reconfigurable Computing*. Springer, 2007.

[2] Partial Reconfiguration User Guide. Technical report, Xilinx Inc., January 2012.

[3] TORC. `http://torc-isi.sourceforge.net/`.

[4] Christoph Albrecht. IWLS 2005 Benchmarks, 2005. `http://iwls.org/iwls2005/benchmarks.html`.

[5] Altera. `http://www.altera.com/`.

[6] Altera. End Markets. `http://www.altera.com/end-markets /end-index.html`.

[7] Peter M. Athanas, J. Bowen, T. Dunham, Cameron Patterson, J. Rice, Matthew Shelburne, Jorge Surís, Mark B. Bucciero, and Jonathan Graf. Wires On Demand: Run-Time Communication Synthesis for Reconfigurable Computing. In *FPL*, pages 513–516, 2007.

[8] Azido. `http://www.azido.net`.

[9] Azido. `http://www.xilinx.com/tools/sysgen.htm`.

[10] Azido. `http://www.altera.com/products/software/products/dsp/dsp-builder.html`.

[11] Jonathan Babb, Matthew Frank, Victor Lee, Elliot Waingold, Rajeev Barua, Michael Taylor, Jang Kim, Srikrishna Devabhaktuni, and Anant Agarwal. The RAW Benchmark Suite: Computation Structures for General Purpose Computing. In *In IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 161–171, 1997.

[12] J. Becker, T. Pionteck, C. Habermann, and M. Glesner. Design and Implementation of a Coarse-Grained Dynamically Reconfigurable Hardware Architecture. In *VLSI, 2001. Proceedings. IEEE Computer Society Workshop on*, pages 41 –46, may 2001.

[13] P. Bellows and B. Hutchings. JHDL - An HDL for Reconfigurable Systems. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, FCCM '98, pages 175–, Washington, DC, USA, 1998. IEEE Computer Society.

[14] Vaughn Betz. The FPGA Place-and-Route Challenge. `http://www.eecg.toronto.edu/~vaughn/challenge/challenge.html/`.

[15] Daniel Brand, Anthony Drumm, Sandip Kundu, and Prakash Narain. Incremental Synthesis. In *Proceedings of the 1994 IEEE/ACM international conference on Computer-aided design*, ICCAD '94, pages 14–18, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[16] Pak K. Chan and Martine D. F. Schlag. New Parallelization and Convergence Results for NC: a Negotiation-Based FPGA Router. In *Proceedings of the 2000 ACM/SIGDA eighth international symposium on Field programmable gate arrays*, FPGA '00, pages 165–174, New York, NY, USA, 2000. ACM.

[17] Deming Chen, Jason Cong, and Peichen Pan. FPGA Design Automation: A Survey. *Found. Trends Electron. Des. Autom.*, 1(3):139–169, January 2006.

[18] C.-S. Choy, T.-S. Cheung, and K.-K. Wong. Incremental Layout Placement Modification Algorithms. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 15(4):437 –445, apr 1996.

[19] Jason Cong and Hui Huang. Depth Optimal Incremental Mapping for Field-Programmable Gate Arrays. In *Proceedings of the 37th Annual Design Automation Conference*, DAC '00, pages 290–293, New York, NY, USA, 2000. ACM.

[20] J. Coole and G. Stitt. Intermediate Fabrics: Virtual Architectures for Circuit Portability and Fast Placement and Routing. In *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2010 IEEE/ACM/IFIP International Conference on*, pages 13 –22, oct. 2010.

[21] James Coole and Greg Stitt. BPR: Fast FPGA Placement and Routing using Macroblocks. In *Proc. of CODES/ISSS'12: IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, Tampere, Finland, October 2012.

[22] Jacob Couch. Applications of TORC: An Open Toolkit for Reconfigurable Computing. Master's thesis, Virginia Tech, 2011.

[23] R.G. Downey and M.R. Fellows. Fixed-parameter tractability and completeness. i. basic results. *SIAM Journal on Computing*, 24(4):873 – 921, 1995/08/. fixed-parameter tractability;completeness;polynomial time solution;k-dominating set;k-feedback vertex set;intractability;parameterized problems;class hier-

archy;natural complete problems;independent set;clique;irredundant set;hardness;W-hierarchy;parameterized complexity;t-normalized satisfiability;.

[24] J.M. Emmert and D. Bhatia. Incremental Routing in FPGAs. In *ASIC Conference 1998. Proceedings. Eleventh Annual IEEE International*, pages 217 –221, sep 1998.

[25] Xilinx Forums. Hard Macro External Pins Deleted When Deleting a Net, 2011. `http://forums.xilinx.com/t5/Implementation/Hard-macro-external-pins-deleted-when-deleting-a-net/td-p/120124`.

[26] Tannous Frangieh, Athira Chandrasekharan, Suresh Rajagopalan, Yousef Iskander, Stephen Craven, and Cameron Patterson. PATIS: Using Partial Configuration to Improve Static FPGA Design Productivity. In *17th Reconfigurable Architectures Workshop (RAW 2010)*, Atlanta, GA, 2010.

[27] M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some Simplified NP-Complete Problems. In *Proceedings of the sixth annual ACM symposium on Theory of computing*, STOC '74, pages 47–63, New York, NY, USA, 1974. ACM.

[28] Paul Glover and Steve Elzinga. Relationally Placed Macros. `http://www.xilinx.com/support/documentation/white\_papers/wp329.pdf`, 2008.

[29] Mentor Graphics. `http://www.mentor.com/`.

[30] A. Hac. Performance Analysis of the Make and Load Building Algorithms. In *Computer Software and Applications Conference, 1990. COMPSAC 90. Proceedings., Fourteenth Annual International*, pages 20 –25, oct-2 nov 1990.

[31] L. Józwiak, A. Chojnacki, and A. Slusarczyk. Fast and Compact Sequential Circuits through the Information-Driven Circuit Synthesis. In *Proceedings of the Euromicro Symposium on Digital Systems Design*, DSD '01, pages 46–, Washington, DC, USA, 2001. IEEE Computer Society.

[32] T.Y.K. Kam and R.K. Brayton. Multi-Valued Decision Diagrams. Technical report, Univiversity of California at Berkeley, 1990.

[33] R. Karp. Reducibility among combinatorial problems. In R. Miller and J. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[34] Victoria Kim. Japan damage could reach $235 billion, World Bank estimates. *Los Angeles Times*, March 2011.

[35] Rachael King. Can Supercomputers Help Japan Predict Earthquakes? *Bloomberg Businessweek*, April 2011.

[36] S. Kirkpatrick, C.D. Gelett, and M.P. Vecchi. Optimization by Simulated Annealing. *Science*, 1983.

[37] Christopher Lavin, Marc Padilla, Jaren Lamprecht, Philip Lundrigan, Brent Nelson, and Brad Hutchings. HMFlow: Accelerating FPGA Compilation with Hard Macros for Rapid Prototyping. In *Field-Programmable Custom Computing Machines (FCCM), 2011 19th IEEE Annual International Symposium on*, pages 117–124, May 2011.

[38] Christopher Lavin, Marc Padilla, Jaren Lamprecht, Philip Lundrigan, Brent Nelson, and Brad Hutchings. RapidSmith: Do-It-Yourself CAD Tools for Xilinx FPGAs. In *Proceedings of the 21th International Workshop on Field-Programmable Logic and Applications (FPL'11)*, September 2011.

[39] Guy Lemieux and David Lewis. Circuit Design of Routing Switches. In *Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays*, FPGA '02, pages 19–28, New York, NY, USA, 2002. ACM.

[40] Guy G. Lemieux and Stephen D. Brown. A Detailed Routing Algorithm for Allocating Wire Segments in Field-Programmable Gate Arrays. In *in Proc. ACM/SIGDA Physical Design Workshop, Lake Arrowhead, CA*, pages 215–226, 1993.

[41] Li-Yi Lin, Hsin-Chang Lin, and Shih-Arn Hwang. Incremental Physical Design Method for Flat SoC Design. In *VLSI Design, Automation and Test, 2009. VLSI-DAT '09. International Symposium on*, pages 351 –354, april 2009.

[42] Roman Lysecky, Frank Vahid, and Sheldon X.-D. Tan. Dynamic FPGA Routing for Just-in-Time FPGA Compilation. In *Proceedings of the 41st annual Design Automation Conference*, DAC '04, pages 954–959, New York, NY, USA, 2004. ACM.

[43] Pongstorn Maidee, Cristinel Ababei, and Kia Bazargan. Timing-Driven Partitioning-Based Placement for Island Style FPGAs. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, March 2005.

[44] Alexander Marquardt, Vaughn Betz, and Jonathan Rose. Timing-Driven Placement for FPGAs. In *Proceedings of the 2000 ACM/SIGDA eighth international symposium on Field programmable gate arrays*, FPGA '00, pages 203–213, New York, NY, USA, 2000. ACM.

[45] Alexander (Sandy) Marquardt, Vaughn Betz, and Jonathan Rose. Using Cluster-Based Logic Blocks and Timing-Driven Packing to Improve FPGA Speed and Density. In

Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays, FPGA '99, pages 37–46, New York, NY, USA, 1999. ACM.

[46] Larry Mcmurchie and Carl Ebeling. PathFinder: A Negotiation-Based Performance-Driven Router for FPGAs. pages 111–117, 1995.

[47] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Higher Education, 1st edition, 1994.

[48] Alan Mishchenko, Satrajit Chatterjee, and Robert Brayton. DAG-Aware AIG Rewriting a Fresh Look at Combinational Logic Synthesis. In *Proceedings of the 43rd annual Design Automation Conference*, DAC '06, pages 532–535, New York, NY, USA, 2006. ACM.

[49] Chandra Mulpuri and Scott Hauck. Runtime and Quality Tradeoffs in FPGA Placement and Routing. In *Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field programmable gate arrays*, FPGA '01, pages 29–36, New York, NY, USA, 2001. ACM.

[50] Brent Nelson, Michael Wirthlin, Brad Hutchings, Peter Athanas, and Shawn Bohner. Design Productivity for Congurable Computing. In *ERSA*, page 5766, 2008.

[51] Raphael Njuguna. A Survey of FPGA Benchmarks, 2008. `http://www.cse.wustl.edu/ jain/cse567-08/ftp/fpga/`.

[52] Bryan T. Preas and Patrick G. Karger. Automatic Placement: A Review of Current Techniques (tutorial session). In *Proceedings of the 23rd ACM/IEEE Design Automation Conference*, DAC '86, pages 622–629, Piscataway, NJ, USA, 1986. IEEE Press.

[53] S. Raman, C.L. Liu, and L.G. Jones. A Timing-Constrained Incremental Routing Algorithm for Symmetrical FPGAs. In *European Design and Test Conference, 1996. ED TC 96. Proceedings*, pages 170 –174, mar 1996.

[54] Yaska Sankar and Jonathan Rose. Trading Quality for Compile Time: Ultra-Fast Placement for FPGAs. In *Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, FPGA '99, pages 157–166, New York, NY, USA, 1999. ACM.

[55] S. Shukla, N.W. Bergmann, and J. Becker. QUKU: A Two-Level Reconfigurable Architecture. In *Emerging VLSI Technologies and Architectures, 2006. IEEE Computer Society Annual Symposium on*, volume 00, page 6 pp., march 2006.

[56] Simulink. `http://www.mathworks.com/products/simulink/`.

[57] Ali Asgar Sohanghpurwala, Peter Athanas, Tannous Frangieh, and Aaron Wood. OpenPR: An Open-Source Partial-Reconfiguration Toolkit for Xilinx FPGAs. In *18th Reconfigurable Architectures Workshop (RAW 2011)*, Anchorage, AK, 2011.

[58] Neil Steiner, Aaron Wood, Hamid Shojaei, Jacob Couch, Peter Athanas, and Matthew French. Torc: Towards an Open-Source Tool Flow. In John Wawrzynek and Katherine Compton, editors, *Proceedings of the ACM/SIGDA 19th International Symposium on Field Programmable Gate Arrays, FPGA 2011, Monterey, California, USA, February 27, March 1, 2011*, pages 41–44. ACM, 2011.

[59] Dmitri B. Strukov and Konstantin K. Likharev. A Reconfigurable Architecture for Hybrid CMOS/Nanodevice Circuits. In *Proceedings of the 2006 ACM/SIGDA 14th*

international symposium on Field programmable gate arrays, FPGA '06, pages 131–140, New York, NY, USA, 2006. ACM.

[60] Kolja Sulimma and Wolfgang Kunz. An Exact Algorithm for Solving Difficult Detailed Routing Problems. In *Proceedings of the 2001 international symposium on Physical design*, ISPD '01, pages 198–203, New York, NY, USA, 2001. ACM.

[61] Wern-Jieh Sun and C. Sechen. Efficient and Effective Placement for Very Large Circuits. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 14(3):349 –359, mar 1995.

[62] Jordan S. Swartz, Vaughn Betz, and Jonathan Rose. A Fast Routability-Driven Router for FPGAs. In *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*, FPGA '98, pages 140–149, New York, NY, USA, 1998. ACM.

[63] Abhay Tavaragiri. A Management Paradigm for FPGA Design Flow Acceleration: Creating a Framework for Vendor Independent Representation of FPGA Designs. Master's thesis, Virginia Tech, 2011.

[64] Russell Tessier. Fast Placement Approaches for FPGAs. *ACM Trans. Des. Autom. Electron. Syst.*, 7(2):284–305, April 2002.

[65] Mak Wai-Kei and Li Hao. Placement for Modern FPGAs. In *In IEEE Emerging Information Technology Conference*, pages 92–95, 2005.

[66] Stephen A. Ward, Arthur C. Smith, and Russel G. Tessier. Fast Place and Route Approaches for FPGAs. Technical report, 1998.

[67] Yu-Liang Wu, S. Tsukiyama, and M. Marek-Sadowska. Graph Based Analysis of 2-D FPGA routing. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 15(1):33 –44, jan 1996.

[68] Xilinx. `http://www.xilinx.com/`.

[69] Xilinx. Applications. `http://www.xilinx.com/applications /index.htm`.

[70] Xilinx. Using an RPM Grid Macro to Control Block RAM-to-FF Timing. `http://www.xilinx.com/support/documentation/application_notes/xapp416.pdf`.

[71] Yonghong Xu and Mohammed A. S. Khalid. QPF: Efficient Quadratic Placement for FPGAs. In *FPL'05*, pages 555–558, 2005.

[72] Masamichi Yoshitsubaki. Great Eastern Japan Earthquake: Three Weeks Later. *NHK World*, April 2011.

Appendix A

Statistical Analysis

This appendix will list all the statistical analysis of the experiments from Chapter 6 on the fifteen IWLS 2005 and the five custom benchmark designs.

A.1 Xilinx ISE 13.4 Implementation Results

Tables A.1 to A.19 present the *minimum*, *maximum*, *mean*, and *standard deviation* runtime for *ngdbuild*, *map* and *par* utilities for all twenty benchmark designs.

Table A.1: ac97 benchmark statistics

Utility	**Minimum (s)**	**Maximum (s)**	**Mean (s)**	**Std. Dev.**
ngdbuild	8.16	8.66	8.37	0.11
map	62.28	63.15	62.66	0.18
par	76.57	78.21	76.97	0.23
Total Time	147.41	149.01	148.01	0.31

Table A.2: des benchmark statistics

Utility	**Minimum (s)**	**Maximum (s)**	**Mean (s)**	**Std. Dev.**
ngdbuild	8.64	9.8	8.88	0.17
map	65.74	72.44	66.3	0.67
par	49.49	50.39	49.86	0.18
Total Time	124.25	131.43	125.6	0.72

Table A.3: ethernet benchmark statistics

Utility	**Minimum (s)**	**Maximum (s)**	**Mean (s)**	**Std. Dev.**
ngdbuild	8.89	9.67	9.19	0.14
map	77.26	82.66	77.81	0.67
par	51.80	52.77	52.19	0.19
Total Time	138.33	144.31	139.19	0.73

Table A.4: i2c benchmark statistics

Utility	**Minimum (s)**	**Maximum (s)**	**Mean (s)**	**Std. Dev.**
ngdbuild	6.90	7.90	7.27	0.14
map	36.85	37.73	37.21	0.16
par	41.80	42.63	42.22	0.16
Total Time	86.13	87.28	86.69	0.25

Table A.5: mem_ctrl benchmark statistics

Utility	**Minimum (s)**	**Maximum (s)**	**Mean (s)**	**Std. Dev.**
ngdbuild	8.14	9.20	8.40	0.15
map	89.57	91.04	90.19	0.29
par	107.72	109.25	108.44	0.29
Total Time	205.93	208.34	207.02	0.44

A.2 qFlow Implementation Results

Tables A.20 to A.38 present the *minimum*, *maximum*, *mean*, and *standard deviation* runtime for *metadata_processor*, *module_placer*, the *fpga_edline* and *par* utilities for all twenty benchmark designs.

Table A.6: pci benchmark statistics

Utility	**Minimum (s)**	**Maximum (s)**	**Mean (s)**	**Std. Dev.**
ngdbuild	8.21	8.82	8.43	0.13
map	63.91	65.30	64.34	0.24
par	86.46	87.96	87.04	0.28
Total Time	158.99	161.00	159.81	0.40

Table A.7: sasc benchmark statistics

Utility	**Minimum (s)**	**Maximum (s)**	**Mean (s)**	**Std. Dev.**
ngdbuild	7.05	7.98	7.24	0.14
map	32.11	32.81	32.41	0.16
par	34.86	35.53	35.22	0.14
Total Time	74.28	75.95	74.87	0.27

Table A.8: simple_spi benchmark statistics

Utility	**Minimum (s)**	**Maximum (s)**	**Mean (s)**	**Std. Dev.**
ngdbuild	7.08	8.18	7.27	0.17
map	34.75	36.02	35.15	0.17
par	33.76	34.62	34.14	0.15
Total Time	75.93	77.41	76.56	0.24

Table A.9: spi benchmark statistics

Utility	**Minimum (s)**	**Maximum (s)**	**Mean (s)**	**Std. Dev.**
ngdbuild	7.36	8.44	7.55	0.18
map	70.82	72.76	71.39	0.32
par	37.38	38.47	37.79	0.19
Total Time	115.86	118.36	116.73	0.46

Table A.10: tv80 benchmark statistics

Utility	**Minimum (s)**	**Maximum (s)**	**Mean (s)**	**Std. Dev.**
ngdbuild	7.80	8.66	8.02	0.14
map	84.11	86.08	84.86	0.36
par	76.09	78.46	76.77	0.35
Total Time	168.55	172.53	169.65	0.63

Table A.11: usb_funct benchmark statistics

Utility	**Minimum (s)**	**Maximum (s)**	**Mean (s)**	**Std. Dev.**
ngdbuild	8.58	9.90	9.01	0.16
map	68.18	70.23	68.63	0.33
par	96.53	98.20	97.26	0.32
Total Time	173.87	176.44	174.90	0.49

Table A.12: usb_phy benchmark statistics

Utility	**Minimum (s)**	**Maximum (s)**	**Mean (s)**	**Std. Dev.**
ngdbuild	6.80	8.45	7.32	0.24
map	33.59	34.53	33.91	0.18
par	40.38	42.31	41.44	0.21
Total Time	81.32	83.91	82.67	0.37

Table A.13: vga_lcd benchmark statistics

Utility	**Minimum (s)**	**Maximum (s)**	**Mean (s)**	**Std. Dev.**
ngdbuild	8.01	8.78	8.22	0.14
map	64.06	65.40	64.69	0.23
par	61.67	62.67	62.15	0.19
Total Time	134.52	135.83	135.06	0.29

Table A.14: wb_dma benchmark statistics

Utility	**Minimum (s)**	**Maximum (s)**	**Mean (s)**	**Std. Dev.**
ngdbuild	7.76	8.87	8.01	0.20
map	68.87	70.15	69.28	0.23
par	76.15	77.44	76.71	0.24
Total Time	153.14	155.10	154.00	0.40

Table A.15: Edge benchmark statistics

Utility	**Minimum (s)**	**Maximum (s)**	**Mean (s)**	**Std. Dev.**
ngdbuild	18.08	21.00	18.72	0.68
map	90.34	138.83	95.64	6.58
par	68.65	77.70	71.11	2.43
Total Time	177.22	231.03	185.47	8.40

Table A.16: Gaussian benchmark statistics

Utility	**Minimum (s)**	**Maximum (s)**	**Mean (s)**	**Std. Dev.**
ngdbuild	17.00	19.08	17.65	0.65
map	71.99	99.85	75.69	3.70
par	62.63	71.04	65.31	2.28
Total Time	151.85	179.98	158.65	5.83

Table A.17: ZigBee benchmark statistics

Utility	**Minimum (s)**	**Maximum (s)**	**Mean (s)**	**Std. Dev.**
ngdbuild	13.05	14.80	13.63	0.59
map	164.46	189.93	169.70	5.26
par	60.95	67.72	62.91	2.04
Total Time	238.73	265.40	246.24	7.07

Table A.18: Vector Add benchmark statistics

Utility	**Minimum (s)**	**Maximum (s)**	**Mean (s)**	**Std. Dev.**
ngdbuild	150.40	168.06	156.21	7.09
map	3661.36	3819.73	3721.10	52.75
par	638.99	659.34	644.80	7.52
Total Time	4471.10	4632.48	4522.11	57.08

Table A.19: Smith-Waterman benchmark statistics

Utility	**Minimum (s)**	**Maximum (s)**	**Mean (s)**	**Std. Dev.**
ngdbuild	176.46	190.08	180.71	4.72
map	4964.89	5070.83	5003.94	32.06
par	1775.03	1898.72	1841.98	33.43
Total Time	6937.36	7095.69	7026.64	45.69

Table A.20: ac97 benchmark statistics

Utility	**Minimum (s)**	**Maximum (s)**	**Mean (s)**	**Std. Dev.**
metadata_processor	1.86	2.08	1.94	0.04
module_placer	3.27	3.52	3.36	0.05
fpga_edline	39.79	40.41	40.10	0.12
par	98.28	177.79	109.11	13.51

Table A.21: des benchmark statistics

Utility	**Minimum (s)**	**Maximum (s)**	**Mean (s)**	**Std. Dev.**
metadata_processor	2.34	2.61	2.41	0.04
module_placer	3.64	4.03	3.84	0.06
fpga_edline	48.85	49.86	49.20	0.19
par	113.82	373.50	133.96	39.05

Table A.22: ethernet benchmark statistics

Utility	**Minimum (s)**	**Maximum (s)**	**Mean (s)**	**Std. Dev.**
metadata_processor	2.64	2.96	2.74	0.05
module_placer	3.98	4.48	4.18	0.11
fpga_edline	52.08	53.55	52.78	0.44
par	123.08	453.67	162.73	95.40

Table A.23: i2c benchmark statistics

Utility	**Minimum (s)**	**Maximum (s)**	**Mean (s)**	**Std. Dev.**
metadata_processor	1.20	1.32	1.25	0.03
module_placer	2.98	3.19	3.09	0.04
fpga_edline	32.91	33.35	33.09	0.09
par	79.83	81.28	80.53	0.27

Table A.24: mem_ctrl benchmark statistics

Utility	**Minimum (s)**	**Maximum (s)**	**Mean (s)**	**Std. Dev.**
metadata_processor	2.37	2.68	2.44	0.04
module_placer	3.46	3.70	3.56	0.05
fpga_edline	46.94	47.61	47.23	0.13
par	108.11	764.08	175.38	131.06

Table A.25: pci benchmark statistics

Utility	**Minimum (s)**	**Maximum (s)**	**Mean (s)**	**Std. Dev.**
metadata_processor	2.79	2.94	2.87	0.04
module_placer	3.17	3.42	3.27	0.05
fpga_edline	49.23	49.87	49.51	0.13
par	109.91	113.41	110.97	0.70

Table A.26: sasc benchmark statistics

Utility	**Minimum (s)**	**Maximum (s)**	**Mean (s)**	**Std. Dev.**
metadata_processor	1.16	1.35	1.22	0.03
module_placer	2.62	2.92	2.71	0.05
fpga_edline	32.57	33.21	32.77	0.11
par	78.60	80.10	79.12	0.27

Table A.27: simple_spi benchmark statistics

Utility	**Minimum (s)**	**Maximum (s)**	**Mean (s)**	**Std. Dev.**
metadata_processor	1.15	1.36	1.23	0.04
module_placer	2.56	2.89	2.73	0.07
fpga_edline	32.45	33.28	32.84	0.16
par	78.05	79.82	79.10	0.37

Table A.28: spi benchmark statistics

Utility	**Minimum (s)**	**Maximum (s)**	**Mean (s)**	**Std. Dev.**
metadata_processor	1.47	1.65	1.55	0.03
module_placer	3.51	3.82	3.64	0.05
fpga_edline	36.58	37.21	36.88	0.13
par	88.61	295.55	96.53	20.52

Table A.29: tv80 benchmark statistics

Utility	**Minimum (s)**	**Maximum (s)**	**Mean (s)**	**Std. Dev.**
metadata_processor	1.54	1.85	1.63	0.04
module_placer	2.87	3.13	2.98	0.04
fpga_edline	37.21	37.95	37.46	0.12
par	96.03	127.34	105.42	7.26

Table A.30: usb_funct benchmark statistics

Utility	**Minimum (s)**	**Maximum (s)**	**Mean (s)**	**Std. Dev.**
metadata_processor	2.55	2.75	2.63	0.04
module_placer	3.34	3.64	3.46	0.05
fpga_edline	46.97	47.77	47.20	0.13
par	112.44	448.08	154.89	80.44

Table A.31: usb_phy benchmark statistics

Utility	**Minimum (s)**	**Maximum (s)**	**Mean (s)**	**Std. Dev.**
metadata_processor	1.17	1.42	1.24	0.03
module_placer	2.60	2.97	2.74	0.05
fpga_edline	32.91	33.45	33.15	0.11
par	79.36	80.77	80.00	0.27

Table A.32: vga_lcd benchmark statistics

Utility	**Minimum (s)**	**Maximum (s)**	**Mean (s)**	**Std. Dev.**
metadata_processor	2.01	2.16	2.08	0.03
module_placer	3.04	3.40	3.14	0.06
fpga_edline	42.59	43.42	42.92	0.15
par	98.10	101.10	99.21	0.52

Table A.33: wb_dma benchmark statistics

Utility	**Minimum (s)**	**Maximum (s)**	**Mean (s)**	**Std. Dev.**
metadata_processor	2.95	3.20	3.04	0.04
module_placer	3.39	3.71	3.50	0.06
fpga_edline	50.09	51.38	50.42	0.15
par	106.10	280.41	113.17	23.87

Table A.34: Edge benchmark statistics

Utility	**Minimum (s)**	**Maximum (s)**	**Mean (s)**	**Std. Dev.**
metadata_processor	4.18	49.57	12.94	8.82
module_placer	60.13	66.53	61.54	1.82
fpga_edline	69.09	79.20	71.02	2.15
par	79.42	124.89	89.91	9.44

Table A.35: Gaussian benchmark statistics

Utility	**Minimum (s)**	**Maximum (s)**	**Mean (s)**	**Std. Dev.**
metadata_processor	3.30	44.04	10.70	7.41
module_placer	53.72	60.11	55.49	1.87
fpga_edline	65.55	73.02	67.83	2.31
par	73.97	114.89	83.78	7.67

Table A.36: ZigBee benchmark statistics

Utility	**Minimum (s)**	**Maximum (s)**	**Mean (s)**	**Std. Dev.**
metadata_processor	2.15	2.52	2.25	0.10
module_placer	48.51	49.46	48.76	0.25
fpga_edline	68.46	72.83	69.22	1.32
par	79.94	84.41	80.81	1.33

Table A.37: Vector Add benchmark statistics

Utility	**Minimum (s)**	**Maximum (s)**	**Mean (s)**	**Std. Dev.**
metadata_processor	184.39	189.22	186.04	1.67
module_placer	5.52	6.61	5.86	0.40
fpga_edline	260.27	280.57	268.62	9.17
par	702.96	734.89	716.35	13.88

Table A.38: Smith-Waterman benchmark statistics

Utility	**Minimum (s)**	**Maximum (s)**	**Mean (s)**	**Std. Dev.**
metadata_processor	9.46	10.82	10.27	0.47
module_placer	856.00	964.36	903.83	39.65
fpga_edline	497.77	523.50	506.84	9.33
par	787.64	816.97	799.30	9.03

Appendix B

Using qFlow

This appendix presents step-by-step instructions on how to assemble a design using qFlow. It walks a designer throughout the 3 steps involved when using qFlow: design jacket generation, computational modules generation and computation sandbox connectivity generation.

B.1 Environment Setup & Project Tree

qFlow makefiles assume the following variables are set in your *.bashrc* file:

```
# qFlow executables directory
export QFLOW_EXEC=/qflow/trunk/src/
# Add executables to PATH
export PATH=$QFLOW_EXEC:$PATH
```

Do not forget to source the Xilinx tools. qFlow is tested on ISE 13.4 and no extensive tests were done on newer versions of the tools. qFlow template projects can be found under qflow/trunk/templates/ Figure B.1 shows the qFlow project directory tree.

B.2 Preparing the Design Jacket

In order to create a qFlow design jacket, a designer needs the design Verilog source files, the Perl script *parse.pl* (qflow/trunk/res/perl/), the bus macro Verilog interface *busmacro_xc5v_async.v* (qflow/
trunk/res/verilog/) , and the bus macro hard macro *busmacro_xc5v_async.nmc* (qflow/trunk/res/nmc/). A designer will perform the next 11 steps to generate the design jacket's physical netlist. The designer can then reuse the design jacket's implementation over and over again until they need to update the size or the interface of the sandbox.

Step 1: Prepare the computation sandbox

Prepare your computation sandbox as a Verilog file. In case your computation sandbox consists of more than one module, create a wrapper around all the computation modules. A Perl script script processes the computation sandbox Verilog file and converts it to a blackbox wrapped by bus macros. Verilog allows couple formats when specifying a module's interface. However, the Perl script *parse.pl* assumes the following format for the computation sandbox Verilog:

```
module your_computation_sandbox(
  clk,
```

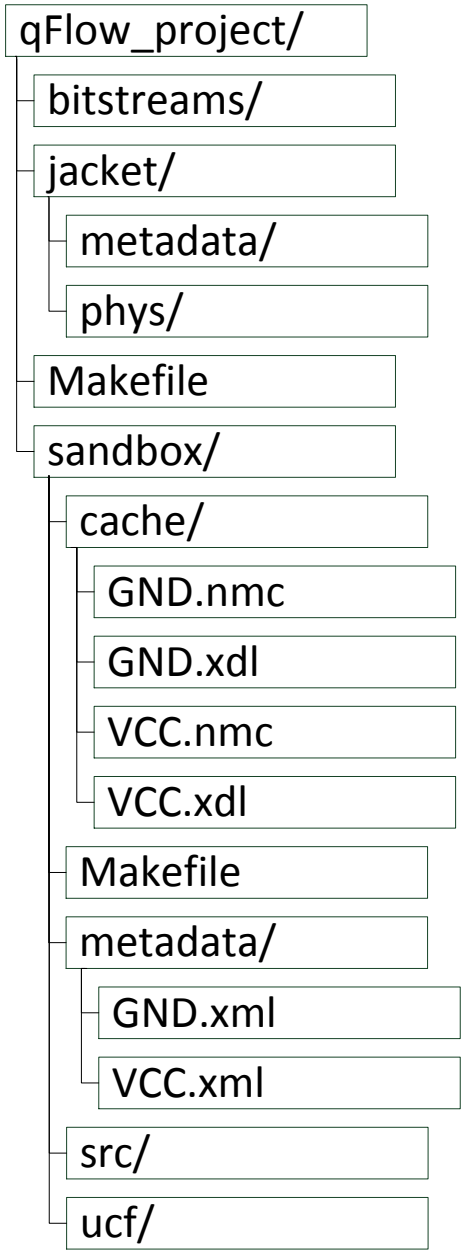


Figure B.1: qFlow project directory tree

```
reset,

in1,
```

```
  in2,
  out1,
  ...
  outn);
  input clk;
  input reset;
  input [x:0] in1;
  input [y:0] in2;
  output [z:0] out1;
  ...
  output outn;
  ...
endmodule;
```

In this above format, every port declaration is on a separate line, so declarations like the following will cause the script to generate erroneous code:

```
  input clk,reset;
  output [x:0] out1,out2;
```

Step 2: Generate a blackbox out of your computation sandbox

Run the script *parse.pl* on the computation sandbox Verilog file using the following command:

```
Perl parse.pl your_computation_sandbox.v
```

When done, the script should display a "done!" message. The script generates three files:

- your_computation_sandbox_bt.v: This is the blackbox corresponding to your computation sandbox. It consists of a matching interface along with bus macros driving all output signals.

- top.v: This file instances *your_computation_sandbox_bt.v* and wraps it with bus macros, and declares all the necessary wires connecting bus macros to *your_computation_sandbox_bt.v* and vice versa.

- top.cf: This file is an example user constraints file showing how to prohibit a region that will host all modules in *your_computation_sandbox_bt.v*. Data in this file is only meant for demonstration purposes and varies from design/device to another.

Step 3: Add the blackbox to your top level Verilog design

Open *your_design_top_level.v* and the Perl script generated *top.v*. From *your_design_top_level.v* comment out the code that instances *your_computation_sandbox.v*. From *top.v* copy the code section between */* Start copying from here into your design top module */* and */* End copying here into your design top module */*, then paste it in *your_design_top_level.v*. Copy the files *your_computation_sandbox_bt.v*, *busmacro_xc5v_async.v* and *busmacro_xc5v_async.nmc* to your working directory. Make sure to include all *.v* files in your project path.

Step 4: Update the user constraints file

Update the user constraint file of your project to prohibit a region on your device. The region should be large enough to host all your computation modules. PlanAhead can help

you decide on a prohibit region and generate the corresponding prohibit constraints.

Step 5: Synthesize and map your design

Synthesize and map your design, using the default ISE 13.4 options.

Step 6: Cleanup your mapped design

Open the design in FPGA Editor (or FPGA Edline) and delete all bus macros that have the name **BT**, using the following commands (make sure you are in read/write mode):

```
unselect -all
select hm *BT*
delete
```

Save and exit.

Step 7: Route your design

Route your design using the default ISE 13.4 options.

Step 8: Generate XDL of your routed design

Generate XDL for your routed design using the following command:

```
xdl -ncd2xdl your_design_top_level.ncd
```

Copy *your_design_top_level.xdl* to qflow_project/jacket/phys/

Step 9: Generate metadata for the design

Generate metadata for your design using the following commands:

```
ngc2edif your_design_top_level.ngc
EdifToXML your_design_top_level.ndf
```

Copy *your_design_top_level.xml* to qflow_project/jacket/metadata/

Step 10: Cleanup the routed design

Open the routed design in FPGA Editor and flatten all bus macros that have the name **bm_**, using the following commands (make sure you are in read/write mode):

```
unselect -all
select hm *bm_*
unbind
```

A pop-up window will ask you if you want to proceed. Press Yes. Select all unrouted nets using FPGA Editor GUI then delete all of them. Save and exit. Copy *your_design_top_level.ncd* to qflow_project/jacket/phys/

Step 11: Cleanup the physical constraints file

Open the file *your_design_top_level.pcf* and delete all PROHIBIT constraints that match the prohibit constraints that you earlier specified in the user constraints file. Save and exit.

Copy *your_design_top_level.pcf* to qflow_project/jacket/phys/ Congratulations, you successfully generated your qFlow design jacket.

B.3 Preparing the Computation

The computation sandbox should be completely modular, i.e., no logic in the computation sandbox should be outside a module. This section will show you how to process the computation modules of your computation sandbox.

Step 1: Synthesize your computation module

Synthesize the computation module in your computation sandbox. Make sure to turn off the automatic I/O buffers insertion option. Assuming you are using XST (though Synplify was already tested, and proven to work), the output of the synthesis process is *NGC* netlist.

Step 2: Generate an EDIF netlist

Convert the *NGC* netlist to an *NDF* (EDIF) format using the following command:

```
ngc2edif ModuleX.ngc
```

Open the *NDF* netlist and delete the status section, usually found close to the start of the file. A sample status section follows:

```
(keywordMap (keywordLevel 0))
(status
```

```
    (written
      (timestamp 2012 6 30 16 5 14)
      (program "Xilinx ngc2edif" (version "O.87xd"))
      (author "Xilinx. Inc ")
      (comment "This EDIF netlist is to be used within supported synthesis tools")
      (comment "for determining resource/timing estimates of the design component")
      (comment "represented by this netlist.")
      (comment "Command line: ac97.ngc ")))
  (external UNISIMS
```

Save and exit. Copy the NDF netlist to qFlow_project/sandbox/src/

Step 3

Generate AREA_GROUP constraints for the computation module using PlanAhead or the qFlow utility pa_script. The utility *pa_script* is still under development and tends to underestimate a module's required resources. Therefore, manual tweaking of the resulting AREA_GROUP constraints might be needed. If you decide to use pa_script, the following set of commands will generate the required AREA_GROUP constraints given ModuleX.ndf netlist:

```
pa_script ${QFLOW_EXEC} ModuleX.ndf qflow xc5vlx110tff1136-1 0 4 20 24 139
make -f shape_module
```

Where 4 20 24 139 are the southwest and northeast Configuration Logic Block (CLB) coordinates of your prohibit region (The region that you prohibited while creating your design jacket). These coordinates need to be adjusted based on your prohibit region. Copy the resulting *ModuleX.ucf* to qFlow_project/sandbox/ucf/ Repeat Steps 1, 2 and 3 for every module in your computation sandbox.

B.4 Generating Computation Sandbox Connectivity Metadata

Generate connectivity information between the different modules inside your computation sandbox and between your computation sandbox and the design jacket. In order to achieve that, synthesize *your_computation_sandbox.v* with the keep hierarchy flag set. Every module inside *your_computation_sandbox.v* should be implemented as a blackbox. A module will be treated as a blackbox during synthesis when you delete all logic inside that module's verilog file and only keep the module interface (i.e., input/output ports). Make sure to turn off the automatic I/O buffers insertion option. Once done, generate the metadata connectivity information using the following commands:

```
ngc2edif your_computation_sandbox.ngc
EdifToXML your_computation_sandbox.ndf
```

Rename *your_computation_sandbox.xml* to *BT.xml* then copy it to qFlow_project/sandbox/metadata/ Congratulations, you are now ready to assemble your design using qFlow.

B.5 Running qFlow

Navigate to qFlow_project/ then type *make*. Sit back and relax. The first time qFlow runs, it implements and caches all modules in your computation sandbox before assembling your design and generating a bitstream. For any future run, qFlow will use any cached modules that were not modified since the last run. Updated modules will be implemented and cached for future re-use.

Appendix C

qFlow Detailed Stages

This appendix details all of qFlow stages illustrated in Figure C.1 and Figure C.2. Each stage of relevance in Figure C.1 and Figure C.2 has its own stage number shown between parentheses.

C.1 Design Partitioning

The following section details the partitioning stage (stage O1) in Figure C.1.

Stage O1: The designer decides what goes into the design jacket and what is treated as evolving logic. GigE, Memory, PCIe, Video and Audio interfaces usually go into the design jacket. Computing modules on the other hand usually go into evolving logic.

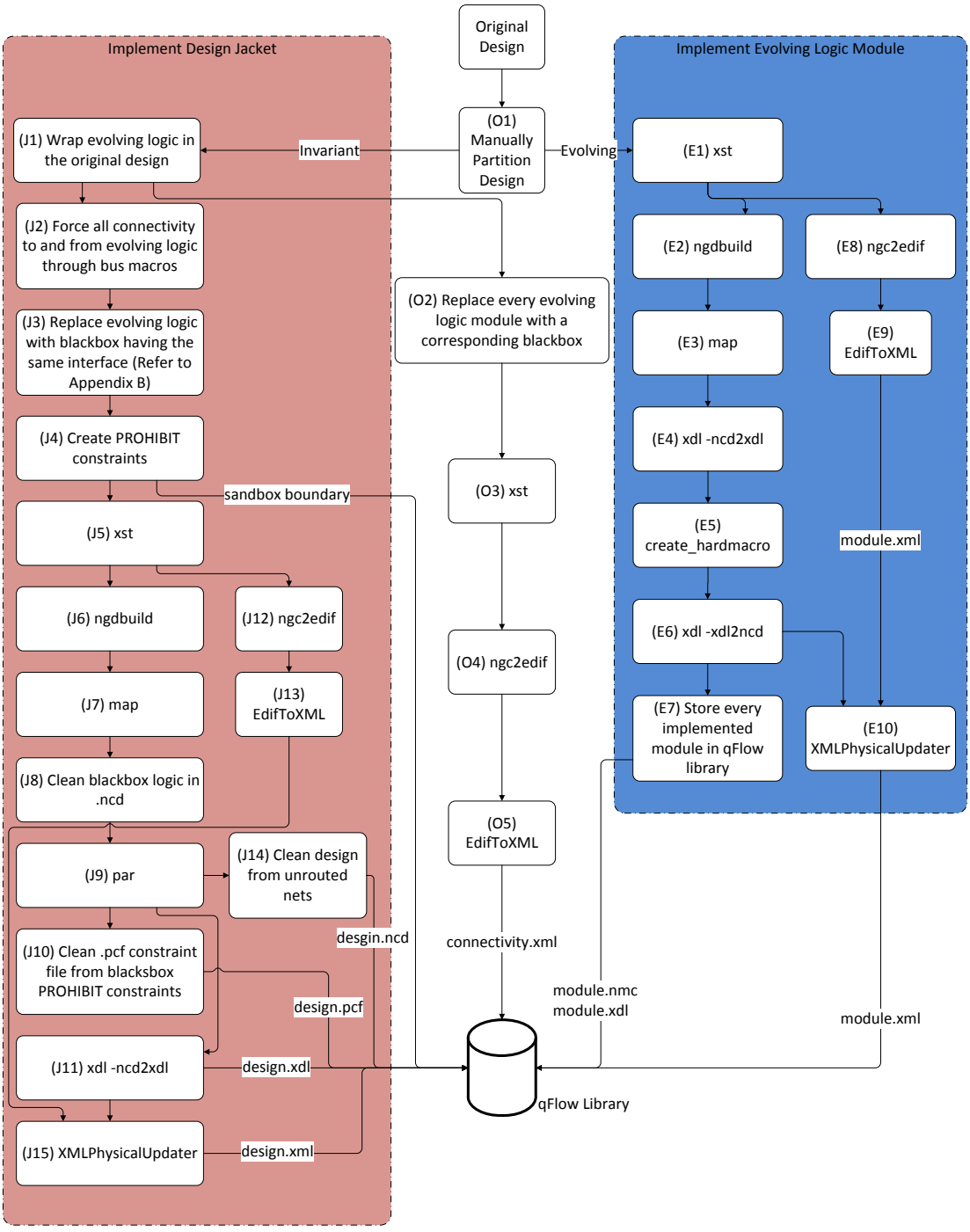


Figure C.1: qFlow process flow for the design jacket and evolving logic implementation

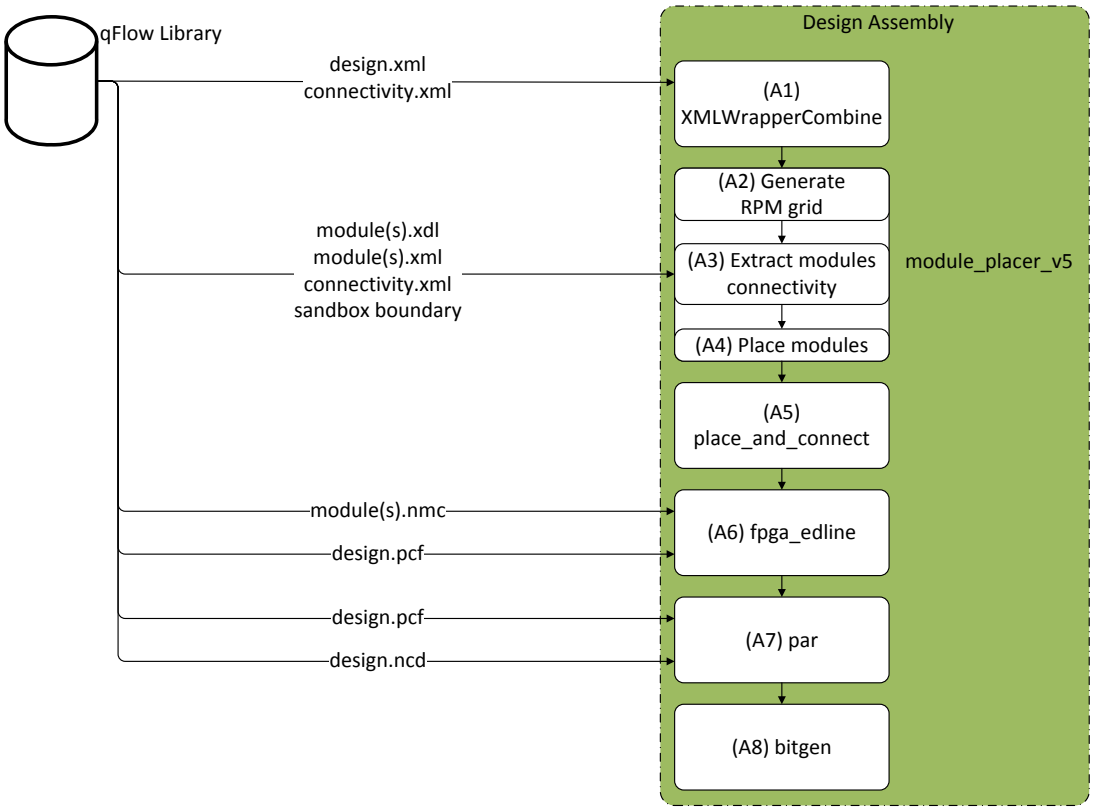


Figure C.2: qFlow process flow for the design assembly phase

C.2 Design Jacket Implementation

The following section details the fifteen steps of the design jacket implementation, stages J1 to J15 in Figure C.1.

Stage J1: Starting with the original HDL design, all evolving modules are wrapped using a Verilog or VHDL wrapper. The wrapper is instanced in the top level design, therefore all evolving logic modules are pushed one level of hierarchy below the design top level, as well as all intra evolving logic modules connectivity.

Stage J2: All nets connecting the wrapper to the rest of the design (invariant logic) are passed through bus macros, creating anchor points for all connectivity between evolving and invariant logic.

Stage J3: A Verilog or VHDL blackbox module is created with the same interface as the wrapper. The wrapper is then removed from the top level design, and replaced by the blackbox instance. The goal is to implement the design jacket excluding evolving logic modules.

A blackbox module usually consists of an interface definition only. However, when pushed through the Xilinx tools, such blackbox was optimized out along with many dangling wires in the design jacket needed during the assembly stage to connect to and from evolving logic modules. To prevent such undesired optimization, the attribute (*S = "TRUE"*) is applied to all input ports of the blackbox, whereas all output ports of the blackbox are driven by bus macros. It is recommended to follow a naming convention for the bus macros since they need to be deleted in a later stage.

Stage J4: The design jacket needs to reserve logic on the device that will host evolving logic modules during the assembly stage. Reserving logic on the design is done by prohibiting regions on the device from being used by the design jacket. Prohibiting regions is done by including prohibit ranges in the user constraint file (UCF format) as shown below:

Chapter 4 names a region on a device that excludes logic and routing resource utilization during the invariant set implementation, a *sandbox*. For the purpose of qFlow, a *sandbox* excludes logic but not routing resources, and routes can still freely cross a sandbox region.

```
CONFIG PROHIBIT=SLICE\_X2Y4:SLICE\_X45Y155; // Prohibit a slice range
CONFIG PROHIBIT=DSP48\_X0Y2:DSP48\_X0Y61;   // Prohibit a dsp range
CONFIG PROHIBIT=RAMB36\_X0Y1:RAMB36\_X1Y30; // Prohibit a ramb range
```

Finally, specifying the prohibit constraints (thus the sandbox boundary) is a manual process.

Stages J5, J6 & J7: The design from stage J3 along with the constraints from J4 are passed to the Xilinx implementation tools for synthesis, mapping and placement. The output of the placement stage is a placed netlist (NCD format).

Stage J8: The placed netlist (NCD format) is updated using Xilinx FPGA Editor, by deleting all bus macros added in stage J3. This can be done by selecting a bus macro by its name then deleting it.

Stage J9: The placed netlist (NCD format) is passed to Xilinx PAR to route the design. The output of the route stage is routed netlist (NCD format).

Stage J10: The physical constraint file (PCF format) constructed by the Xilinx tools during the implementation stages is updated, by cleaning all prohibit constraints.

Stage J11: The routed netlist (NCD format) is converted into a textual representation (XDL format) using Xilinx *xdl* utility with *-ncd2xdl* flag set.

Stage J12: The synthesized netlist (NGC format) is converted to a textual synthesized netlist (EDIF format) using Xilinx *ngc2edif* utility.

Stage J13: Metadata information is extracted from the textual synthesized netlist (EDIF format). Information mainly includes the names of all nets connecting invariant to evolving logic, along with the driving/driven port names.

Stage J14: The routed netlist (NCD format) is updated using Xilinx FPGA Editor by cleaning any remaining unrouted nets. Unrouted nets may exist mainly due to actions taken in stage J3.

Stage J15: The metadata information from stage J13 is populated with physical information extract from the textual representation of the routed netlist (XDL format) from stage J11. Specifically, all physical locations (pins) for the nets crossing the invariant/evolving logic boundary are specified.

C.3 Evolving Logic Implementation

The following section details the ten steps of the evolving logic implementation, stages E1 to E10 in Figure C.1.

Stages E1, E2 & E3: An evolving module is passed to the Xilinx implementation tools for synthesis, mapping and placement. The output of the placement stage is a placed netlist (NCD format).

Stage E4: The placed netlist (NCD format) is converted into a textual representation (XDL format) using Xilinx *xdl* utility with *-ncd2xdl* flag set.

Stage E5: The textual representation of the placed netlist (XDL format) is transformed into a textual representation of a hard macro (XDL format), referred to as an RPM throughout this document. The process consists of creating input and output ports for the module, internally driving any constant signals, and setting a reference point for the hard macro. The resulting hard macro (XDL format) includes placement and connectivity but not routing information.

Stage E6: The hard macro textual representation (XDL format) is converted into a binary format (NMC format) using Xilinx *xdl* utility with *-xdl2ncd* flag set.

Stage E7: The hard macro (NMC format) is copied to the cache folder for re-use when needed.

Stage E8: The synthesized netlist (NGC format) is converted to a textual synthesized netlist (EDIF format) using Xilinx *ngc2edif* utility.

Stage E9: Metadata information is extracted from the textual synthesized netlist (EDIF format). Information mainly includes the names of all module's input and output ports.

Stage E10: The metadata information from stage E9 is populated with physical information extract from the textual representation (XDL format) of the hard macro. Specifically, all physical locations (pins) are specified for all input and output ports.

C.4 Sandbox Connectivity Extraction

The following section details the four steps of extracting the sandbox connectivity, stages O2 to O5 in Figure C.1. The sandbox connectivity consists of the connectivity between the design jacket and the evolving modules, and between the evolving modules themselves.

Stage O2: Every evolving logic module in the wrapper from stage J1 is replaced by a blackbox with an identical interface. The output of this stage is a Verilog or VHDL wrapper module.

Stage O3: The wrapper module is passed to the Xilinx *xst* utility with the *keep_hierarchy* flag set.

Stage O4: The synthesized netlist (NGC format) is converted to a textual synthesized netlist (EDIF format) using Xilinx *ngc2edif* utility.

Stage O5: Metadata information is extracted from the textual synthesized netlist (EDIF format). Information mainly includes the names of all instanced evolving logic modules, connectivity between the design jacket and the evolving modules, and between the evolving modules themselves

C.5 Design Assembly

The following section details the eight steps of design assembly, stages A1 to A8 in Figure C.2.

Stage A1: The metadata information from stage O5 is populated with physical information extracted from the metadata information from stage J15. Specifically, all physical locations (pins) for the nets crossing the invariant/evolving logic boundary are specified.

Stage A2: The RPM grid is generated. The RPM grid is a replica of the Xilinx RPM grid.

Stage A3: A weighted undirected graph data structure that reflects the connectivity between the design jacket and the evolving modules, and between the evolving modules themselves is created. The nodes in that graph are the evolving modules and the bus macro from J2. The edges represent the connectivity between the nodes, with an edge weight reflecting the number of wires between the corresponding two nodes.

Stage A4: An initial random placement computes the placement for all evolving modules. A SA placer refines the initial placement with the wire length is used a cost function.

Stage A5: The placement information from stage A4 along with the sandbox connectivity from stages O2 to O5 is represented in an FPGA Editor script. Specifically, the script specifies the location of every evolving module hard macro reference point based on stage A4, and creates all connectivity between the evolving modules hard macros.

Stage A6: The design jacket physical netlist (NCD format) is updated by opening it using FPGA Edline, then running the script generated in stage A5. The result of this stage is a design jacket (NCD format) that incorporates all evolving modules as hard macros (NMC format) and the corresponding connectivity information. Evolving modules hard macros are

still not routed at this stage, as well as the between the design jacket and the hard macros, and between the hard macros themselves. Therefore, the resulting design jacket netlist (NCD format) is partially routed.

Stage A7: The partially routed design jacket (NCD format) is passed to Xilinx PAR with the *-k* flag set. This will enable PAR to complete routing of a partially routed netlist (NCD format). The output of this stage is a fully routed netlist (NCD format).

Stage A8: The fully routed netlist (NCD format) from stage A7 is passed to Xilinx *bitgen* utility to generate the device configuration.

C.6 qFlow Running Scenarios

qFlow consists of four implementation phases: design partitioning, design jacket generation, evolving set implementation and design assembly. qFlow reacts differently by triggering one or more of the implementation phases when faced with a design update, based on the nature of the update itself. Three types of updates can occur: an update to the design jacket, an update one or more evolving logic modules, and an update to the connectivity between the design jacket and the evolving modules, and/or between the evolving modules themselves is created.

An update to the design jacket triggers stages J1 to J15, followed by stages A1 to A8 for design assembly. If the design jacket update involves an update to the interface between the invariant logic and the evolving logic, i.e. the interface of the wrapping module from

stage J1, stages O2 to O5 are triggered. As of now, all design jacket updates are handled manually.

When an update to an evolving module occurs, stages E1 to E10 are triggered for the updated module only, followed by stages A1 to A8 for design assembly. Any evolving logic that is not updated will be retrieved from the qFlow cache folder.

Finally, when an update to the connectivity between the design jacket and the evolving modules, and/or between the evolving modules themselves occurs, stages O2 to O5 are triggered, followed by stages A1 to A8 for design assembly.